UNIT-2

Function Overloading :-

→ Function Overloading is a feature in C++ where two or more functions can have the same name but different parameters.

→ Function overloading allows you to use the the same name for different functions to perform either same or different functions in the same class.

→ Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but with different number or types of arguments then you can simply overloaded the function.

→ Function can be overloaded in two types

* By changing number of Arguments
* By having different types of Arguments.

Different number of Arguments :-

→ In this type of function overloading we define two functions with same names but different numbers of parameter of the same type.

Ex

```
# include <iostream>
using namespace std;
class Sum
{
int sum (int x, int y){ cout << x+y; }
sum (int x, int y, int z)
{
cout << x+y+z;
}
int main ()
{
sum (10,20);
sum (10,20,30); }
```

Scanned by CamScanner

Different data type of arguments :-

In this type of overloading we define two or more function with same name and same no. of parameters but the type of parameter is different.

ex

```
int sum ( int x, int y)
  {
   cout << x+y;
  }
float sum ( float x, float y)
  {
   cout << x+y;
  }
int main()
  {
   sum (10,20);
   sum ( 10.25, 25.03);
  }
```

Functions with default arguments :-

When we mentioned a default value for a parameter while declaring the function, it is said to be as default argument. In this case eve if we make a call to the function without passing any value for that parameter the function will take the default value specified.

ex

```
# include <iostream>
using namespace std;
class Addition
  {
  int sum ( int x, int y)
    {
     return x+y;
    }
  int sum ( int x, int y, int z)
    {
     return x+y+z;
    }
  };
int main()
  {
  Addition A1;
  cout << "A1.sum (20,15) << endl;
  cout << " A1. sum (5,6,9);
  return 0;
  }
```

Advantages :-

→ To improve the code readability and allows code reusability.

* In Function overloading the function in parent class is called the overhidden function and function in child class is called overriding function.

Scanned by CamScanner

→ Function overloading happens in the same class when we declare same functions with different argrements in the same class.

→ In function overloading function signature should be different for all the overloaded functions.

→ In function overloading we can have any number of overloaded functions.

Operator Overlodding :-

→ It is a type of polymerphism in which an operator is overloaded to give user defined meaning to it.

→ overloaded operator is used to perform operation on user-defined data type for example '+' operator can be overloaded to perform addition on various data types, like for integer, string etc.

→ Almost any operator can be overloaded in C++. However there are few operator which can not be overloaded. Operator that are not overloaded are follows

- Scope operator - ::
- sizeof
- Member selector - .
- member pointer selector - *
- ternary operator - ?:

→ Operator overloading function can be a member function if the left operand is an object of that class but if the left operand is different then operator overloading function must be a non-member function.

→ Operator overloading function can be made friend function if it needs access to the private and protected member of class.

Restriction on overloading in C++

→ precedence and Associativity of an Operator cannot be changed.

→ Arity (no of operands) cannot be changed. Unary operator remains unary, binary remains binary.

→ Now new operators can be created only existing operators can be overloaded.

→ cannot redefine the meaning of a procedure. You can not change how integers are added.

Criteria to define the Operator function:

→ In case of a non-static function the binary operator should have only one argument and unary should not have an argument.

→ In case of a friend function the binary operator should have only two argument and unary should have only one argument.

→ All the class member object should be public if operator overloading is implemented.

→ Operator that cannot be overloaded are
.. * :: ?:

→ Operator cannot be used to overload when declaring that function as friend function
= () [] →.

Syntax

```
return type classname :: Operator op(arglist)
{
    // function body
}
```

Scanned by CamScanner

Types of Operator overloading :-

There are three types of operator overloading

- unary operator overloading
- Binary operator overloading
- overloading binary operator using friend function.

Unary Operator overloading :-

→ In unary operator function, no argument should be passed.

→ It works only with one class objects. It is a overloading of an operator operating on a single operand.

Ex

①
```
# include <iostream>
using namespace std;
class space
{
    int x,y,z;
public:
    void getdata (int a, int b, int c)
    {
    x = a;
    y = b;
    z = c;
    }
    void display ();
    void operator - ();
};

void space :: display()
{
    cout << x << " ";
    cout << y << " ";
    cout << z << "\n";
}
void space :: operator-()
{
    x = -x;
    y = -y;
    z = -z;
}
int main()
{
    space s
    s.getdata (10, -20, 30);
    cout << "s :";
    s.display ();
    -s;
    cout << "s :";
    s.display ();
    return 0;
}
```

②
```
# include <iostream>
using namespace std;
class A
{
    int val;
public:
    A (int z)
    {
    val = z;
    }
    void show()
    {
    cout << "The value is :" << val << endl;
    }
    operator + (class A)
};
```

Scanned by CamScanner

```
int main()
{
  A a1(10), a2(20)
  val = a1 + a2
  z.display();
}
```

```
A A :: operator+(const A)
{
  int temp;
  temp.val = val + A.val;
  return temp
}
```

Binary operator overloading :-

→ In binary operator overloading function, there should be one argument to be passed.

→ It is overloading of an operator operating on two operands.

→ In overloading of binary operators the left-hand operand is used to invoke the operator function and the right-hand operator is passed as an argument.

Ex

```
# include <iostream>
using namespace std;
class complex
{
    float x, y;
Public:
    complex() { }
    complex ( float real, float img)
      { x = real; y = img ;}
    complex operator+ (complex);
    void display();
};

complex :: operator+ (complex c)
{
    complex temp;
    temp.x = x + c.x;
    temp.y = y + c.y;
    return (temp);
}
void complex :: display()
{
  cout << x << " +j " << y << "\n";
}
int main()
{
  complex c1, c2, c3;
  c1 = complex (2.5, 3.5);
  c2 = complex (1.6, 2.7);
  c3 = c1 + c2;
  cout << "c1 = "; c1.display();
  cout << "c2 = "; c2.display();
  cout << "c3 = "; c3.display();
  return 0;
}
```

NOTE

* It receives only one complex type argument explicitly.
* It returns a complex type value.
* It is a member function of complex.

Overloading binary operator using Friend

→ The overloading operator function must precede with keyword friend, and declare a function class scope.

→ Friend operator function takes two parameter in a binary operator, varies one parameter in a unary operator.

→ All the working and implementation would same as binary operator function except this function will be implemented outside of the class scope.

Scanned by CamScanner

→ Friend function may be used in the place of member function for overloading a binary operator.

Difference being that a friend function requires two arguments to be explicity passed to it. While a member function requires only one.

→ An object need not be used to invoke a friend function but can be passed as an argument. Thus we can use a friend function with a built-in type data as the left-hand operand and an object as the right-hand operand.

ex

```
# include <iostream>
using namespace std;
class A;
class Time
{
int h,m,s;
Public:
Time(int x, int y, int z)
{ h=x; m=y; s=z; }
Time() { }
void showdata()
{
cout << "The value is :" << h << ":" << m << ":"
}
                << s << ":";
Friend time operator - (Time A)
};
class A
{
int val;
public:
A (int z) { val=z;}
Friend time operator - (Time A)
}
};
Time operator- (Time 1st, A sec)
{
Time = temp;
temp.h = 1st h - sec.val;
temp.m = 1st m - sec.val;
temp.s = 1st s - sec.val;
return temp;
}
int main()
{
Time t1 (5,20,30), t2
a.(20);
t2 = t1 - a1;
t2.showdata();
}
```

All

```
# include <iostream>
using namespace std;
Class A
{
float val;
public:
A() { }
A (int z) { val=z;}
```

Scanned by CamScanner

```
void showdata()
    {
    cout << " The value is : " << val << endl;
    }
void operator+ (A second)
    {
    int sum
    sum = val + second.val;
    cout << " The sum is : " << sum;
    }
A operator- (A second)
    {
    A temp;
    cout << " In 1st value is : " << val;
    cout << " In 2nd value is : " << ~~val~~ second.val;
    temp. val = val - second.val;
    return temp;
    }
A operator~ ()
    {
    val = val + 10.59;
    }
};
int main()
{
A a1(100), a2(50), a3;
a1.showdata();
~a1;
a1. showdata();
a3 = a1 - a2;
a3. showdata();
}
```

Templates

→ Template is the simple idea is to pass data type as a parameter so that we don't need to write same code for different data types.

→ C++ adds two new keywords to support templates 'template' and 'typename'. The second keyword can always be replaced by keyword 'class'.

→ Templates are expanded at compiler time. This is like macros. The difference is compiler does type checking before template expansion.

→ like normal parameters we can pass more than one data types as arguments to templates.

→ Both function overloading and templates are examples of polymorphism feature of OOP.

→ Templates are often used in larger codebase for the purpose of code reusability and flexibility of the program.

→ Templates are the foundation of generic programming which involves writing code in a way that is independent of any particular type.

→ A template is a blueprint or formula for creating a generic class or a function.

Function Templates

→ We write a generic function that can be used for different data types. example of function templates are sort(), max(), min(), printArray().

→ A function template works in a similar to a normal function with one key difference.

Scanned by CamScanner

→ A single function template can work with different data types at once but a single normal function can only work with one set of data types.

→ A function templates starts with the keyword template followed by template ~~followed by~~ parameter/s inside <> which is followed by function declaration.

```
# include <iostream>
# include <string>
using namespace std;
template <typename T>
inline T const & Max (T const& a, T const& b)
  {
    return a<b ? b:a;
  }
int main ()
  {
    int i = 39;
    int j = 20;
    cout << " Max (i,j): " << Max (i,j) << endl;
    double F1 = 13.5;
    double F2 = 20.7;
    cout << " Max (F1, F2): " << Max (F1,F2) << endl;
    string s1 = " Hello";
    string s2 = " World";
    cout << " Max (s1, s2): " << max (s1,s2) << endl;
    return 0;
  }
```

Class Template :-

Normally you would need to create a different class for each data type OR create different member variables and functions within a single class.

→ Inside the class body a member variable var and a member function someOperation() are both of type T.

→ To create a class template object you need to define the data type inside a <> when creation.

```
ClassName < data type> class object;
```

→ Class templates are useful when a class defines something that is independent of data type. can be useful for classes like linkedlist, Binary tree, stack, Queue, Array etc.

```
# include < iostream>
using namespace std;
template < class T, class u>
class complex
  {
    T real;
    u img;

  complex () { }
  void display ()
    {
      cout << real << " + " << img << " i ";
    }
  complex () { T r, u i }
    {
      real = r;
      img = i;
    }
  };
```

Scanned by CamScanner

```
int main()
{
    complex <int, float> c1 (3,4.5);
    c1.display();
}
```

Inheritance

→ Inheritance in object oriented programming can be described as a process of creating new classes from existing classes.

→ New classes inherit some of the properties and behaviour of the classes. An existing class that is "parent" of a new class is called a base class. New class that inherits properties of the base class is called a derived class.

→ Inheritance is a technique of code reuse. It also provides possibility to extend existing classes by creating derived classes.

Syntax

class Derivedclass : accessspecifier Baseclass

→ The inheritance mode is used mostly. In this the protected member of Base class becomes protected members of derived class and public becomes public.

→ The capability of a class to derive properties and characteristics from another class called inheritance.

→ sub class :- The class that inherits properties from another class called sub class or Derived class.

→ super class :- The class whose properties are inherites are inherited by sub class is called Base class or super class.

Base and derived classes :-

→ The mechanism of deriving a new class from an old class is called inheritance. The old class is refered as the base class and the new class is called derived class.

→ A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class.

→ A derived class inherit all base class methods with the following exceptions.

* constructors, destructors and copy constructors of the base class.
* overloaded operators of the base class.
* The friend function of the base class.

Access control

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions in the base class.

We can summarize the different access types according to who can access them.

Scanned by CamScanner

Acess	public	protected	private
Same class	Yes	Yes	Yes
Derived class	Yes	Yes	no
outside class	Yes	no	no

Inheritance Mode

We deriving a class from a base class the base class may be inherited through public, protected or private inheritance.

We hardly used protected or private inheritance but public inheritance is commonly used.

Public

When deriving a class from a public base class, public members of the base class become public member of the derived class and protected members of the ~~class~~ base class becomes protected members of derived class.

→ A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

Protected

When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

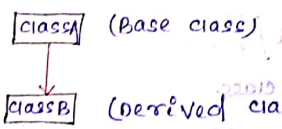
Private

When deriving from a private base class, public and protected members of the base class become private members of derived class.

Types of Inheritance

Single Inheritance

In single inheritance a class is allowed to inherit from only one class i.e one sub class is inherited by one base class only.



eg

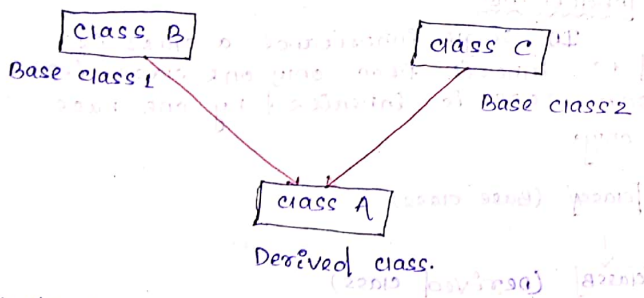
```
# Include <iostream>
using namespace std;
class vehicle {
public:
    Vehicle()
    {
    cout << " This is a Vehicle" <<endl;
    }
};

class car : public vehicle {
};
int main()
{
    car c1;
    return 0;
}
```

Output - This is a Vehicle

Scanned by CamScanner

Multiple Inheritance

→ In multiple inheritance a class can inherit from more than one classes. i.e one sub class is inherited from more than one base classes.



→ Number of base classes will be separated by a comma (,).

→ Ex

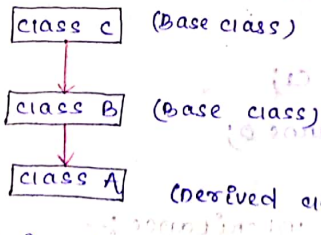
```
# include <iostream>
using namespace std;
class vehicle {
    public:
        vehicle()
        {
        cout << " This is a vehicle" << endl;
        }
    };
class FourWheeler {
    public:
    FourWheeler()
        {
        cout << " This is a 4 wheeler
                vehicle " << endl;
```

```
};
class car : public vehicle, public FourWheeler
        {
        };
int main()
        {
        car c1;
        return 0;
        }
```

Multilevel Inheritance :-

→ In multilevel inheritance a class can inherit from more than one classes. i.e one subclass is inherited from one subclass is inherited from more than one base classes.

→ In this type of inheritance a derived class is created from other derived class.



Ex

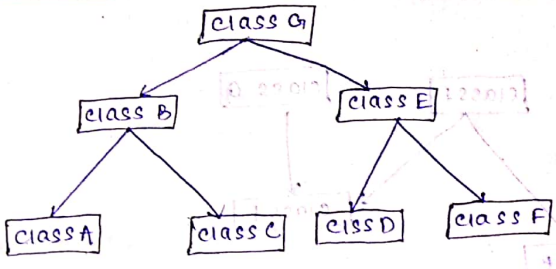
```
# include <iostream>
using namespace std;
class vehicle {
    public:
        vehicle()
        {
        cout << " This is a vehicle " << endl;
        }
    };
class FourWheeler : public vehicle
        {
```

Scanned by CamScanner

```
public :
    FourWheeler()
    {
    cout << " object with 4 wheels are vehicles" << endl;
    }
};
class car : public FourWheeler {
    public :
        car()
        {
        cout << " car has 4 wheels" << endl;
        }
};
int main()
{
    car c1;
    return 0;
}
```

Hierarchical inheritance :-

→ In this type of inheritance, more than one sub class is inherited from a single base class. is inherited from a single base class. i.e more one derived class is created from a single base class.



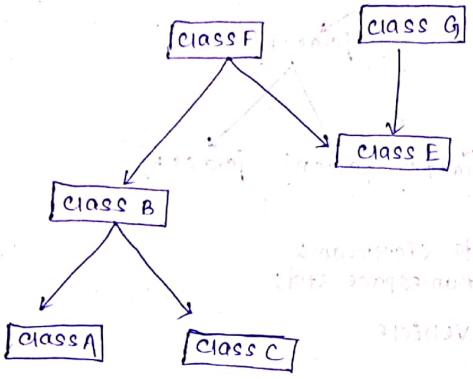
Ex

```
# include <iostream>
using namespace std;
class vehicle
{
 public:
   vehicle()
   {
   cout << " This is a vehicle " << endl;
   }
};
class car : public vehicle {
};
class Bus : public vehicle{
};
int main()
{
 car obj1;
 car obj2;
 return 0;
}
```

Hybrid Inheritance

Hybrid inheritance is implemented by combining more than one type of inheritance. for example : combining Hierachical inheritance and multiple inheritance.

Scanned by CamScanner



Ex

```
# include <iostream>
using name_space std;
class Vehicle
{
public:
  Vehicle()
  {
  cout << "This is a vehicle" << endl;
  }
};
class Fare
{
public:
Fare()
  {
  cout << "Fare of Vehicle\n";
  }
};
class car : public Vehicle
{
};
class BUS: Public Vehicle, public fare
int main()
{
B
};
```

```
int main()
{
Bus obj2;
return 0;
}
```

Pollymerphism:-

Polymorphism means more than one function with same name, with different working. polymerphism can be static or dynamic.

→ In static polymorphism memory will be allocated at compile-time. In dynamic pollymerphism memory will be allocated at run-time. Both function overloading and operator overloading are an examples of static polymerphism.

→ static polymerphism is also known as early binding and compile time polymerphism.

→ Dynamic polymerphism is also known as late binding and run time pollymerphism.

```
# include <iostream>
# include <conio.h>
using namespace std;
class A
{
public:
  Void virtual disp() = 0;
}
};
class B : public A
```

Scanned by CamScanner

```
{
 int val1;
public: A (int x) { val1 = x }
 Void show()
   {
   cout << " val1 in class B" << val1;
   }
 };
class c : class A
  {
  int val2;
 public:
      A ( int x) { val2 = x }
 Void show()
   {
   cout << "val2 in class c" << val2;
   }
 };
int main()
  {
   A *bp;
   B objb (100);
   c objc (200);
   bp = & obj b;
  bp → show();
  bp = & obj c
  bp → show();
  return 0;
}
```

Virtual function:-

→ It is a runtime polymorphism.

→ It is a member function which is declared with in the base class and redefined by derived class.

→ Virtual function is used in situation when we need to invoke derive class function using base class pointer.

→ Virtual function ensure that the correct function is called for an object, regardless of the type of reference used for function call.

→ functions are declared with a Virtual Key word in base class.

Rules:-

→ Virtual function must be member of same class.

→ They cannot be static member.

→ They are accessed by using object pointer.

→ Virtual function can be friend by other function.

```
# Include <iostream>
using namespace std;
class A
{
public:
    Void Virtual show()
  {
  cout << " hello base class";
  } };
```

Scanned by CamScanner

```
class B : public A
{
    public:
        void show()
    {
    cout << " hello derived class";
    }
};

int main()
{
    A aobjett();
    B bobj();
    A *bptr;

    bptr = &aobj();
    bptr -> show();
    bptr = &bobj();
    bptr -> show();
    return 0;
}
```

Scanned by CamScanner