OPERATOR OVERLOADING
&
DATA CONVERSION

OPERATOR OVERLOADING

- Similar to function overloading, operator overloading is another polymorphism feature in C++.
- In function, we use the same function name for different contexts. Likewise, operator overloading uses the same operators such as +, -, /, * in different scenarios.
- The operators in C++ such as +, -, /, * can operate on int, float double, etc. But they cannot operate on user-defined data types such as objects without extension; that is writing an additional piece of code.
- C++ permits us to make user-defined data types behave like built-in types by allowing the overloading of operators.
- Operator overloading is the method of giving additional meaning to the operators so that they can also work with user-defined variables. For adding two complex numbers, we have used the following statement:

  c3.add(c1 , c2); or c3 = c1.add(c2);

  Now, if we overload the + operator to add complex numbers, the above statement can be replaced by

  c3 = c1 + c2 // + is overloaded to act on objects

OVERLOADABLE OPERATORS

The significance of operator overloading is that user-defined data types behave like built-in data types, thus allowing users to extend the language and making the code more readable. C++ supports operator overloading, but at least the operand used with the operator should be the instance of class i.e object of a class.
In C++, all operators can be overloaded except the following:

- sizeof sizeof operator .
- Member operator .* Pointer to
- member operator :: Scope
- resolution operator ?:
- Conditional Operator

SYNTAX OF OPERATOR OVERLOADING

The operator function is defined with the keyword operator followed by the operator symbol. Like a function, the operator function has a return type and arguments. The operator function is in the following form:

```
return_type operator operator_symbol(arg_list) {

        //Body of the function
}
```

The operator function can be declared as a member function of a class or as a friend function of the class

```
class classname
{
        //.....
    public:
        return_type operator operator_symbol(arg_list);          //as a member function
        friend return_type operator operator_symbol(arg_list;    // as a friend function

};
```

RULES OF OPERATOR OVERLOADING

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. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
4. We cannot change the basic meaning of an operator i.e. we cannot redefine the plus(+) operator to subtract one value from another.
5. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
6. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

UNARY OPERATOR OVERLOADING

The operators which operate on a single operand(data) are called unary operators.

For member function of a class:

return_type classname:: operator operator_symbol()

{ ….//body of the function }

For friend function of a class:

friend return_type classname:: operator operator_symbol(arg1);

The unary operators in C++ are either prefix or postfix with the operand.

```
class class_name
{
        public: return_type operator operator_symbol() //for prefix
                        …// body of function        }

        { return_type operator operator_symbol(int); //for postfix
        {           …// body of function        }
};
```

Unary Operator Overloading

```
# include <iostream>
using namespace std;
class Stud
{
int a,b;
public:
void input(){
      cout<<"\n Enter the value of a: ";
      cin>>a; cout<<"\n Enter the value
      of b: "; cin>>b;

} void display()
{
       cout<<"\n The values of a and b are " << a <<" and "<<b<<
endl;
}
void operator -();
};
```

```
void Stud::operator -()
{
      a = -a;
      b = -b;
}
 int main()
{
      Stud S1,S2;
      S1.input();
      -S1;
      S1.display();
      return 0;
}
```

Output:

Enter the value of a: 1

Enter the value of b: -8

The value of a is: -1

The value of b is: 8

Unary Operator Overloading

Prefix operator overloading using friend function:

```
#include <iostream>
using namespace std;
class Complex
{
      int r, m;
 public:
      void input()
      {
          cout << "Enter real and imaginary part \n ";
          cin >> r >> m;
        }
        void display()
        {
          cout << r << "+" << m << "i" << endl;
         }
         void operator --(Complex &);
};
void operator --(Complex& c)
{
      c.r = c.r - 1;
      c.m =c.m - 1;
}
int main()
{
     Complex c1;
     c1.input();
     --c1;
     c1.display();
     return 0;
}
```

Output:

```
Enter real and imaginary part
4
5
3+4i
```

BINARY OPERATOR OVERLOADING

The operators which operate on two operands(data) are called binary operators.

```
class class_name
{
    public:
        return_type operator_symbol(class_name arg){....}
};
```

The binary operator can also be defined as a non-member function of the class. The binary operator defined as a non-member function has the following form:

```
return_type class_name::operator operator_symbol(class_name obj1, class_name obj2)
{
    //body of function
}
```

BINARY OPERATOR OVERLOADING

In the binary operator using the friend function we can specify the order of the left and right operand. The first argument implies left operand and second argument implies right operand.

Note: In binary operator overloading using friend function we can specify left hand operand as class object or built in data type. Same as in the right hand operand. But in binary operator overloading using member functions the left and right hand operand must be the object of the class which contains the operator function with the right hand operand of the class or built in data type as argument to it.

Binary Operator Overloading

```
#include <iostream>
using namespace std;
class Complex
{
    int r, m;
    public:
        void input()
        {
              cout << "Enter real and imaginary part " << endl;
              cin >> r >> m;
         }
         void display()

        {
              cout << r << "+" << m << "i" << endl;

       }
       Complex operator + (int);
};
```

```
Complex Complex::operator + (int a){
        Complex temp;
        temp.r = r+a;
        temp.m = m+a;
        return temp;
}
int main(){
        Complex c1, c2;
        c1.input();
        c2 = c1 + 2; // c1.operator+(2)
        c2.display();
        return 0;
}
```

Output:

```
Enter real and imaginary part
3
4
5+6i
```

Binary Operator Overloading

```
#include <iostream>
using namespace std;
class Complex
{
    int r, m;
    public:
        void input()
        {
            cout << "Enter real and imaginary part" << endl;
            cin >> r >> m;
        }
        void display()
        {
            cout << r << "+" << m << "i" << endl;
         }
        friend Complex operator + (Complex, int);
};
```

```
Complex operator + (Complex c, int a){
        Complex temp;
        temp.r = c.r+a;
        temp.m = c.m+a;
        return temp;
}
int main(){
        Complex c1, c2;
        c1.input();
        c2 = c1 + 2; // operator+(2)
        c2.display();
        return 0;
}
```

Output:

```
Enter real and imaginary part
4
5
6+7i
```

Binary Operator Overloading

WAP to compare the magnitude of a complex number by overloading <, > and ==

```
#include <iostream>
#include <math.h>
using namespace std;
enum Bool { FALSE, TRUE };
class Complex
{
   int r;
 int i;
public:
      void input(){
         cout << "Enter real and imaginary part" ;
         cin >> r >> i;
       }
       void display(){
          cout << r << "+" << i << "i" << endl;
       }
```

```
Bool operator < (Complex C){
      float m1 = sqrt(r*r + i*i);
      float m2 = sqrt(C.r*C.r + C.i*C.i);
      return (m1 < m2 ? TRUE : FALSE);
}
Bool operator > (Complex C){
      float m1 = sqrt(r*r + i*i);
      float m2 = sqrt(C.r*C.r + C.i*C.i);
      return (m1 > m2 ? TRUE : FALSE);
}
Bool operator == (Complex C){
      float m1 = sqrt(r*r + i*i);
      float m2 = sqrt(C.r*C.r + C.i*C.i);
      return (m1 == m2 ? TRUE : FALSE);
}
};
```

Binary Operator Overloading

```
int main()
{
    Complex c1, c2;
    c1.input();
    c2.input();
    if(c1<c2)
        cout << "1st complex number is less than 2nd complex number" << endl;
    else if(c1>c2)
        cout << "1st complex number is greater than 2nd complex number" << endl;
    else if(c1==c2)
        cout << "1st complex number is equal to 2nd complex number" << endl;
    c1.display();
    c2.display();
    return 0;
}
```

Output:

```
Enter real and imaginary part 4 5
Enter real and imaginary part 6 7
1st complex number is less than 2nd complex number
4+5i
6+7i
```

Multiple Choice questions on Operator Overloading

1. What is operator overloading in C++?
A) Using operators to change the value of variables.
B) Defining new operators in the language.
C) Providing a new implementation for an existing operator to work with user-defined data types.
D) Creating functions that have the same name but different parameters

2.Which of the following operators can be overloaded?
A) ++
B) &&
C) ?:
D) ::

3.What must be true for an operator function in C++?
A) It must return a void type.
B) It must be defined as a friend function.
C) It must have at least one operand of a user-defined type.
D) It must be a member function

4.Which operator cannot be overloaded in C++?
A) []
B) new
C) *
D) sizeof()

5.Which of the following is not a valid reason for using operator overloading?
A) To perform object-oriented programming.
B) To allow user-defined types to behave like fundamental types.
C) To increase the complexity of the code.
D) To provide intuitive syntax for operations on user-defined types.

6.What is the syntax of overloading operator + for class A?
a) A operator+(argument_list){}
b) A operator[+](argument_list){}
c) int +(argument_list){}
d) int [+](argument_list){}

7.How many approaches are used for operator overloading?
a) 1
b) 2
c) 3
d) 4

8. In the case of friend operator overloaded functions how many maximum object arguments a binary operator overloaded function can take?
a) 1
b) 2
c) 3
d) 0

9. What will be the output of the following C++ code?

```
#include <iostream>
#include <string>
using namespace std;
class complex
{
    int i;
    int j;
  public:
    complex(){}
    complex(int a, int b)
  {
        i = a;
        j = b;
    }
    complex operator+(complex c)
  {
        complex temp;
        temp.i = this->i + c.i;
        temp.j = this->j + c.j;
        return temp;
    }
    void show(){
cout<<"Complex Number: "<<i<<" + i"<<j<<endl;
    }
};

int main()
{
    complex c1(1,2);
    complex c2(3,4);
    complex c3 = c1 + c2;
    c3.show();
    return 0;
}
```

a) Complex Number: 4 + i6
b) Complex Number: 2 + i2
c) Error
d) Segmentation fault

10. Which is the correct statement about operator overloading?
a) Only arithmetic operators can be overloaded
b) Only non-arithmetic operators can be overloaded
c) Precedence of operators are changed after overlaoding
d) Associativity and precedence of operators does not change

11. What is the keyword for separting the pre and post increment/decrement operator for overloading operator in C++?

a) Void
b) Null
c) Float
d) int

DATA CONVERSION

- The = operator will assign a value from one variable to another in statements like

  int var1 = int var2;

  where int var1 and int var2 are integer variables.
- We may also have noticed that = assigns the value of one user-defined object to another, provided they are of the same type, in statements like

  dist3 = dist1 + dist2;

  where the result of the addition, which is type Distance, is assigned to another object of type Distance, dist3.
- Normally, when the value of one object is assigned to another of the same type, the values of all the member data items are simply copied into the new object. The compiler doesn’t need any special instructions to use = for the assignment of user-defined objects such as Distance objects.
- What if the assignment operator is used for different type i.e., float to int or int to user-defined. Conversion between user defined type and built in type cannot be performed implicitly by the compiler but C++ allows type conversion between them from after the rules for the type conversion have been defined.

DATA CONVERSION

Three types of situations arise in the data conversion between incompatible types:

- Conversion from basic type to user defined
- Conversion from user defined to basic type
- Conversion from one user defined to another user defined

Basic to User defined Conversion

A basic to class conversion can be performed through a constructor with arguments of basic type. The constructor must have only one argument.

```
constructor (basic type) {
    // conversion steps;
}
```

```
#include<iostream>
using namespace std;
class Complex
{
    int r;
    int m;
    public:
        Complex()
        {
            r=0;
            m=0;
        }
        Complex(int a) // for Basic to Class conversion
        {
            r=a;
            m=0;
        }
        void display()
        {
            cout<<"\nThe real is:"<<r;
            cout<<"\nThe imag is:"<<m;
        }
};
int main()
{
        Complex c1;
        c1=9;
        c1.display();
        return 0;
}
```

Output:

The real is:9

The imag is: 0

User defined to Basic Conversion

user defined to Basic type conversion can be achieved using operator function. C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type.

```
operator typename()
{
    // conversion statements
}
```

This function is defined inside a class. The object of this class is converted by the statements in the body of function and returns a variable of type name.

User defined to Basic Conversion

```
#include<iostream>
#include<cmath>
using namespace std;
class Complex
{
    int r;
    int m; public:
    void input()
    {
        cout<<"\nEnter the value of r & m";
        cin>>r>>m;
}
void display()
{
      cout<<"\nReal:"<<r;
      cout<<"\nImag:"<<m;
}
operator float()
{
      float m1;
      m1=sqrt(r*r+m*m);
      return m1;
}
 };
 int main()
{
      Complex c1;
      c1.input();
      float magnitude=c1;
      cout<<"\nThe magnitude is :"<<magnitude;
      return 0;
}
```

Output:

Enter the value of r & m 3 4

The magnitude is :5

User defined to User defined Conversion

User defined to user defined conversion requires identification of source class and destination class. The right hand operand of the assignment operator acts as destination class operand and left hand sided operand is source class operand.

For example, ***obj1 = obj2;***

If obj1 is an object of class A and obj2 is an object of class B, then class A is the destination class and class B is the source class.

User defined to user defined Conversion can be performed in two ways:

- using constructor
- using operator function

User defined to User defined Conversion

When using constructor, the constructor is defined inside the destination class and the object of source class type is the argument of the constructor.

```
#include<iostream>
using namespace std;
class Grade
{
      float d;
      public:
            void input() {
                  cout<<"\nEnter Grade:";
                  cin>>d;
            }
            float getGrade(){
                  return d;
            }
};
```

```
class Radian
{
      float r;
      public:
            Radian(){
                  r=0.0;
            }
            Radian(Grade);
            void display(){
                  cout<<"\nThe radian is:"<<r;
            }
};
Radian::Radian(Grade G){
      r=(G.getGrade()*3.14)/200;
}
```

```
int main()
{
      Grade     G1;
      Radian    R1;
      G1.input();
      R1=G1;
      R1.display();
      return 0;
}
```

Output:

Enter Grade:50

The radian is:0.785

User defined to User defined Conversion

When using operator function, the function is defined inside source class with a return of destination class object.

```
#include<iostream>
using namespace std;
class Radian;
class Grade{
    float d;
    public:
        void input() {
            cout<<"\nEnter Grade:";
            cin>>d;
        }
        Grade(){
            d=0.0;
        }
        operator Radian();
};
```

```
class Radian
{
    float r;
    public:
        Radian(){
            r=0.0;
        }
        void display(){
            cout<<"\nThe radian is:"<<r;
        }
        void setRadian(float r){
            this->r=r;
        }
};
```

User defined to User defined Conversion

```
Grade:: operator Radian(){
        Radian R;
        R.setRadian(d*3.14/200);
        return R;
}
int main()
{
        Grade G1;
        cout<<"\nGrade to Radian ";
        G1.input();
        Radian R1;
        R1=G1;
        R1.display();
        return 0;
}
```

Output:

Grade to Radian
Enter Grade:100
The radian is:1.57

EXPLICIT CONSTRUCTOR

- There may be situations where you don't want some type conversions to take place.
- It is easy to prevent conversion using a casting function, just don't define a casting function inside the class.
- However, preventing through constructors is not as easy, as you may need one argument constructor to initialize the data member of the class.
- Therefore, to prevent this implicit conversion, ANSI C++ standards have defined a keyword ***explicit***.

```
class XYZ
{
        int A;
       public:
       explicit XYZ(int m)
       {
            A = m;
      }   ….. //other members

};
```

Now, when an object of XYZ is declared as below:

XYZ obj(5); *//object can be created*

But,

XYZ obj1 = 45; *//This is not allowed and is illegal*

Multiple Choice questions on Data Conversion

1. What is the correct way to explicitly convert a float to an int in C++?

A) int x = (float)num;
B) int x = static_cast<int>(num);
C) int x = convert<int>(num);
D) int x = int(num);

2. What is the process called when a double is automatically converted to an int by the compiler?

A) Explicit conversion
B) Implicit conversion
C) Typecasting
D) Overloading

3. Which of the following is not a valid C++ type conversion?

A) double to int
B) char to int
C) int to double
D) string to int

4. Which of the following operators can be used for explicit type conversion in C++?

A) cast
B) convert
C) static_cast
D) reinterpret_cast

5. Which of the following is an example of a user-defined conversion function in C++?

a) A constructor that takes a single parameter
b) A destructor that takes no parameters
c) A member function that returns a value of a different type
d) None on the above

6.What is the syntax for the conversion operator in C++?

a) operator returnType() const
b) operator returnType() const()
c) returnType operator() const;
d) None of the above

7. Which of the following is not a valid data type conversion in C++?

a) Implicit conversion
b) Explicit conversion
c) Dynamic conversion
d) None

INHERITANCE

INHERITANCE

- Inheritance is the process of creating new classes, called derived classes, that inherit properties from existing or base classes.
- The derived class inherits all the capabilities of the base class but can add embellishments and refinements of its own.
- The base class is unchanged by this process.
- Inheritance is an essential part of object oriented programming. It provides code reusability.

Need for Inheritance

1. Capability of expressing the inheritance relationship that ensures closeness with the real
2. Allows reusability of code, i.e., addition of additional features to an existing class without modifying it.
3. Reusing existing code that has already been debugged saves time and money and increases a program’s reliability.

BASE AND DERIVED CLASS

A Derived class is the class which inherits the property of another class. The class from which the properties are inherited is known as base class. Derived class is also known as subclass or child class. Base class is also known as the parent class. The base class is unchanged by this inheritance process.

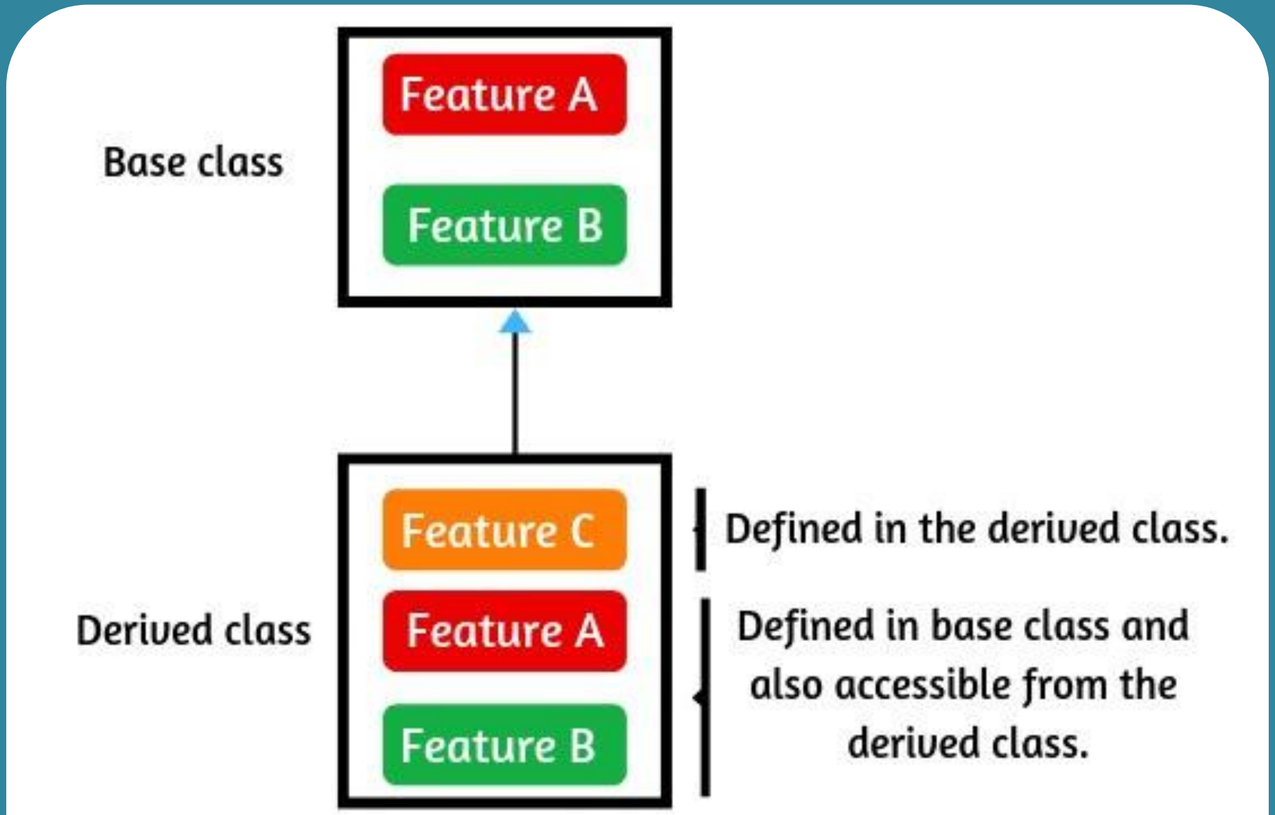


Fig: Base class and Derived class relationships

PROTECTED ACCESS SPECIFIER

Private access specifiers cannot be inherited and public members are inheritable but directly accessible through objects. Protected access specifier is used when the data members or member functions are required to be inheritable but inaccessible through objects. Like private members, they can be accessed only through functions.

So, If you are writing a class that you suspect might be used, at any point in the future, as a base class for other classes, then any data or functions that the derived classes might need to access should be made protected rather than private. This ensures that the class is “inheritance ready”.

DERIVED CLASS DECLARATION

```
class derived_class_name : visibility_mode base_class_name
{
    // members
};
```

The classname is followed by colon(:), visibility mode and base class name respectively. Visibility mode may be private, public, or protected.

Note: By default the visibility mode is private.

For inheritance from multiple classes, commas are used as shown below:

```
class derived_name : visibility base1_name, visibility base2_name
{
    // members
};
```

Visibility mode

Visibility Mode	Private Member of Base	Public Members of Base	Protected Members of Base
public	Not Inherited	public	protected
private	Not Inherited	private	private
protected	Not Inherited	protected	protected

- Private Members of Base class are never inherited.
- In public visibility mode, the public members are inherited as public members of derived class and protected members are inherited as protected members i.e. members are inherited in the same access specifier.
- Private visibility mode inherits both public and protected members in the private access section of the derived class.
- Protected visibility mode inherits both public and protected members into the protected section of the derived class.

```
/* single inheritance */
#include<iostream>
using namespace std;
class Base
{
    protected:
       int a;
    public:
       void inputBase()
      {
           cout<<"enter the value of a:";
           cin>>a;
       }
       void displayBase()
       {
           cout<<"the value of a is:"<< a;
           cout<<endl;
       }
};
```

```
class Derived: public Base
{
    int b;
    public:
    void inputDerived()
    {
        inputBase();
        cout<<"enter a value of b:";
        cin>>b;
    }
    void displayDerived()
    {
        displayBase();
        cout<<"the value of b is:"<< b;
        cout<<endl;
        cout <<"the sum ="<< a+b;
        cout<< endl;
    }
};
```

```
int main()
{
      Derived D1;
      D1.inputDerived();
      cout<<"Displaying the taken value:";
      cout<<endl;
      D1.displayBase();
      D1.displayDerived();
      return 0;
}
```

Output:

```
enter the value of a:3
enter a value of b:5
Displaying the taken value:
the value of a is:3
the value of a is:3
the value of b is:5
the sum =8
```

MEMBER FUNCTION OVERRIDING

- The process of creating members in the derived class with the same name as that of the visible members of the base class is known as **function overriding.**
- It is called overriding because the new name in the derived class overrides (hides or displaces) the old name inherited from the base class.
- If we define a function on a derived class in the same name as an overloaded function in a base class, even with a different parameter list, the base class functions are hidden.
- C++ has a mechanism to access those functions; the base member can be accessed with the base class name and scope resolution operator before the function name.
- Redeclaration of member functions in derived class which is already defined inside visible sections (private and public) of base is known as **function overriding**.

In the next example, ‘Derived’ class is inherited from ‘Base1’ and ‘Base2’. All the classes have input() and display() functions. This is function overriding as the derived class has the same functions as base class. In main(), when input() and display() are invoked, the functions of the derived class are called.

Function Overriding

Note: The overridden functions of a base class can be invoked in two ways:

1. *From the member function of the derived class*
2. *From the object of derived class by using the scope resolution operator.(eg: obj.base::display();)*

```
#include<iostream>
using namespace std;
class Base1
{
    protected:
        int a;
    public:
        void input(){
            cout<<"enter value to a of Base1: ";
            cin >> a;

        }
        void display(){

            cout <<"the a of Base 1 is:" << a << endl;
        }
};
```

```
class Base2
{
    protected:
        int a;
    public:
        void input(){
            cout<<"enter value of a for Base2: ";
            cin >> a;
        }
        void display(){

            cout<<"the a of Base2 is: " << a << endl;
        }
};
```

Function Overriding

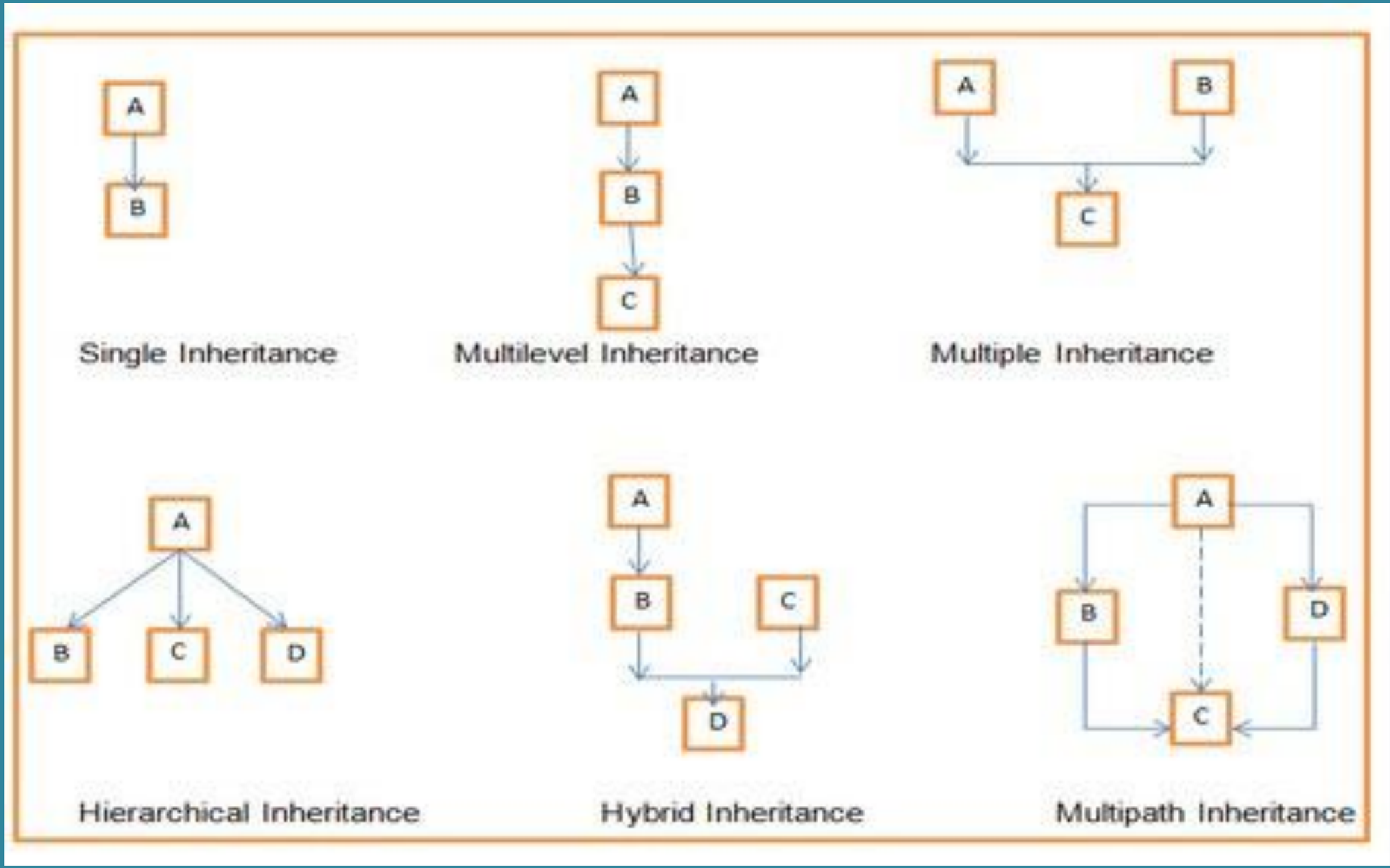
```
class Derived: public Base1, public Base2
{
    protected:
    int c;
    public:
        void input()
        {
            cout<<"enter the value of c: ";
            cin >> c;
        }
        void display()
        {
            Base1::display();
            Base2::display();
            cout<<"the value of c is:" << c << endl;
            cout<<"the sum is:" << Base1::a + c + Base2::a << endl;
        }
};
```

```
int main()
{
    Derived D1;
    D1.Base1::input();
    D1.Base2::input();
    D1.input();
    cout<<"displaying the taken value:" << endl;
    D1.display();
    return 0;
}
```

Output:

```
enter value to a of Base1: 3
enter value to a of Base2: 4
enter the value of c: 5
displaying the taken value:
the a of Base1 is:3
the a of Base2 is:4
the value of c is:5
the sum is:12
```

FORMS OF INHERITANCE



Single inheritance

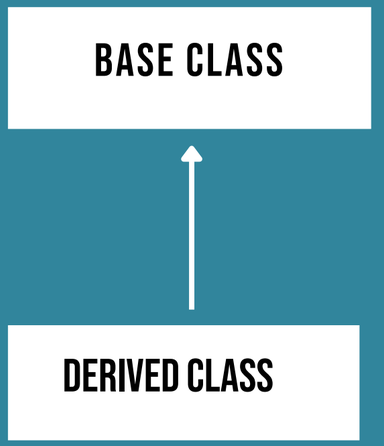
This is the simplest form of inheritance. One derived class is inherited from one base class.

Syntax:

class derived_class_name: visibility_mode base_class_name

{ ……

};




```
#include <iostream.h>
class Value
{
protected:
int val;
public:
void set_values (int a){
val=a;
}
};
```

```
class Cube: public Value
{
public:
int cube()
{
    return (val*val*val);
}
};
```

```
int main ()
{
Cube cub;
cub.set_values (5);
cout << "The Cube of 5 is::" << cub.cube() << endl;
return 0;
}
```

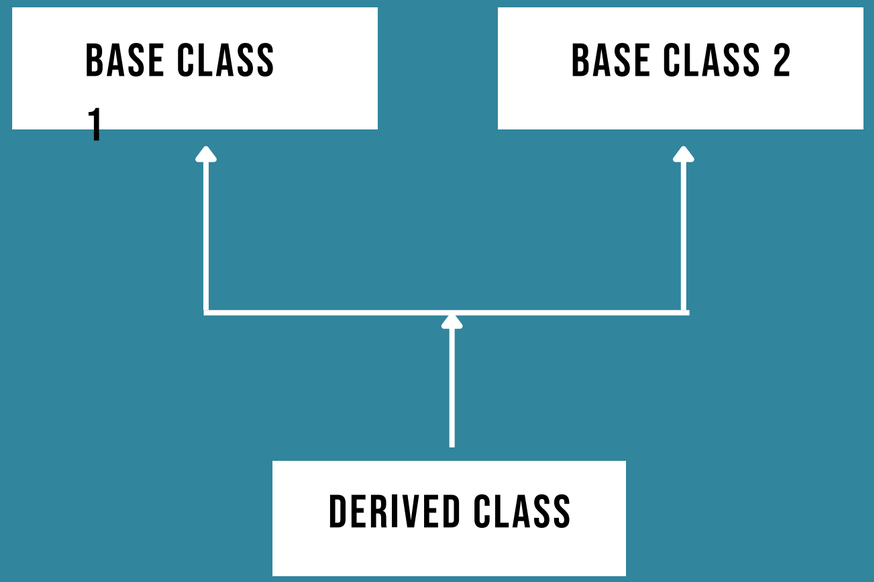
Multiple inheritance

When a derived class is inherited from two or more base classes it is known as multiple inheritance.

class derived_class_name : visibility_mode base_class1, visibility_mode base_class2, ... {

.....

};



Multiple inheritance

```
#include<iostream>
using namespace std;
class Base1
{
protected:
int a;
public:
void inputBase1() {
cout<<" enter the value of a:" << endl;
cin >> a;
}
void displayBase1()
{
cout<<"the value of a is:"<< a << endl;
}
};
class Base2
 {
protected:
int b;
public:
void inputBase2() {
cout<<"enter the value of b:" << endl;
cin >> b;
 }
void displayBase2() {
cout<<"the value of b is:" << b << endl;
}
};
```

Multiple inheritance

```
class Derived:public Base1, public Base2
{
int c;
public:
void inputDerived(){
cout<<"enter the value of c:" << endl;
 cin >> c;
inputBase1();
inputBase2();
}
void displayDerived()
{
displayBase1();
displayBase2();
cout<<"the value of c is:" << c << endl;
cout<<"the sum is:" << a+b+c << endl;
 }
 };
```

```
int main()
{
Derived D1;
D1.inputDerived();
cout<<"displaying the taken value: "<< endl;
D1.displayDerived();
return 0; }
```

Output:

```
enter the value of c: 3
enter the value of a: 1
enter the value of b: 2
displaying the taken value
the value of a is: 1
the value of b is: 2
the value of c is: 3
the sum is: 6
```

Ambiguity in multiple inheritance

When a derived class is inherited from multiple base classes,i.e., two or more base classes, and the base classes have the same functions, ambiguity arises. This ambiguity can be removed using the scope resolution operator (::).

In the next example, ‘Derived’ class is inherited from ‘Base1’ and ‘Base2’. Both ‘Base1’ and ‘Base2’ have the same functions input() and display(). When inherited into ‘Derived’, ambiguity arises as it is unclear whether the function of ‘Base1’ or ‘Base2’ is to be called. So, scope resolution is used to resolve this ambiguity.

Ambiguity in multiple inheritance

```
#include<iostream>
using namespace std;
class Base1
 {
      protected:
            int x;
      public:
            void input()
            {
                  cout<<"enter value to x of Base1: ";
                  cin >> x;
            } void
            display()
            {
                  cout <<"the x of Base 1 is:" << x << endl;
            }
};
```

```
class Base2
{
      protected:
            int x;
      public:
            void input()
            {
                  cout<<"enter value of x for Base2: ";
                  cin >> x;
            }
            void display()
            {
                  cout<<"the x of Base2 is: " << x << endl;
            }
};
```

Ambiguity in multiple inheritance

```
class Derived: public Base1, public Base2
{
    protected:
    int c;
    public:
        void inputDerived(){
            cout<<"enter the value of c: ";
            cin >> c;
            Base1::input();
            Base2::input();
        }
        void displayDerived(){
            Base1::display();
            Base2::display();
            cout<<"the value of c is:" << c << endl;
            cout<<"the sum is:" << Base1::x + c + Base2::x << endl;
        }
};
```

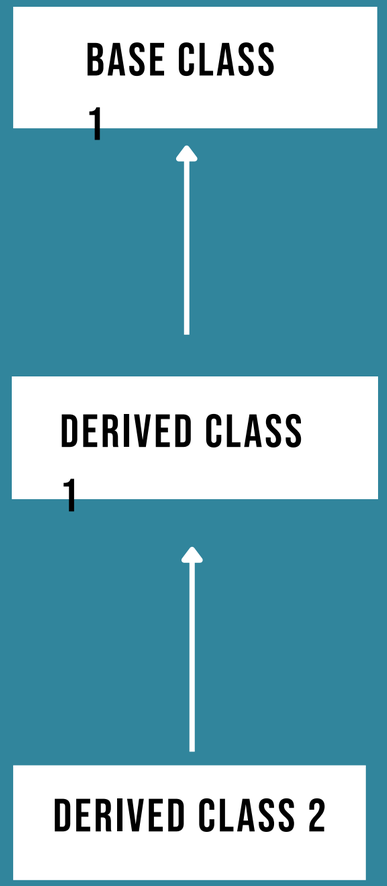
```
int main()
{
    Derived D1;
    D1.inputDerived();
    cout<<"displaying the taken value:" << endl;
    D1.displayDerived();
    return 0;
}
```

Output:

```
enter the value of c: 5
enter value to x of Base1: 4
enter value of x for Base2: 3
displaying the taken value:
the x of Base 1 is:4
the x of Base2 is: 3
the value of c is:5
the sum is:12
```

Multilevel inheritance

When a derived class acts as a base class for another class, it is known as multilevel inheritance.



Multilevel inheritance

```
#include <iostream>
using namespace std;
class A{
protected:
int a;
public:
void inputA() {
cout << "Enter the value of a:";
cin >> a;
}
void displayA() {
cout << "a = " << a << endl;
}
};
```

```
class B: public A
{
    protected: int b;
    public: void inputB()
    {
        inputA();
        cout << "Enter the value of
        b:"; cin >> b;
    }
    void displayB()
    {
        displayA();
        cout << "b = " << b << endl;
     }
 };
```

Multilevel inheritance

```
class C: public B
{
    int c;
    public:
    void inputC()
    {
        inputB();
        cout << "Enter the value
        of c:"; cin >> c;
    }
    void displayC()
    {
        displayB();
        cout << "c = " << c << endl;
    }
 };
```

```
int main() {
C obj;
obj.inputC();
obj.displayC();
return 0;
}
```

Output:

Enter the value of a:1

Enter the value of b:2

Enter the value of c:3

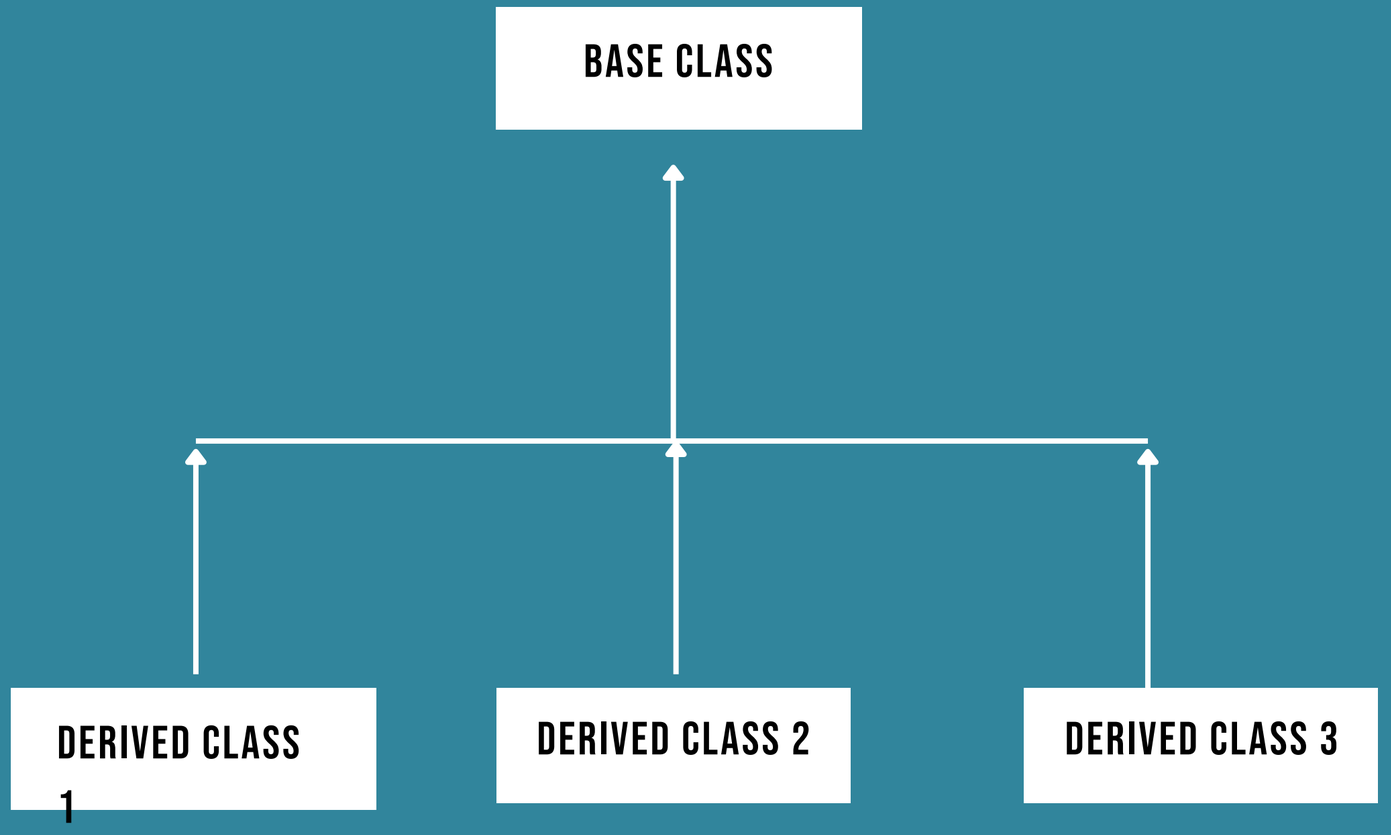
a = 1

b = 2

c = 3

Hierarchical inheritance

When two or more classes from one base class, it is known as hierarchical inheritance.



Hierarchical inheritance

```
#include <iostream>
using namespace std;
class A {
protected:
int a;
public:
void inputA() {
cout << "Enter the value of a:";
cin >> a;
}
void displayA() {
cout << "a = " << a << endl;
}
};
```

```
class B: public A
{
    protected:
        int b;
    public:
        void inputB()
    {
        inputA();
        cout << "Enter the value of b:";
        cin >> b;
    }
    void displayB()
    {
        displayA();
        cout << "b = " << b << endl;
    }
};
```

Hierarchical inheritance

```
class C: public A
{
     int c;
    public:
    void inputC()
    {
    inputA();
    cout << "Enter the value of c:";
    cin >> c;
    }
    void displayC()
    {
    displayA();
    cout << "c = " << c << endl;
    }
};
```

```
int main()
{
cout << "Class B" << endl;
B obj1;
obj1.inputB();
obj1.displayB();
cout << "Class C" << endl;
C obj2;
obj2.inputC();
obj2.displayC();
return 0; }
```

Output:

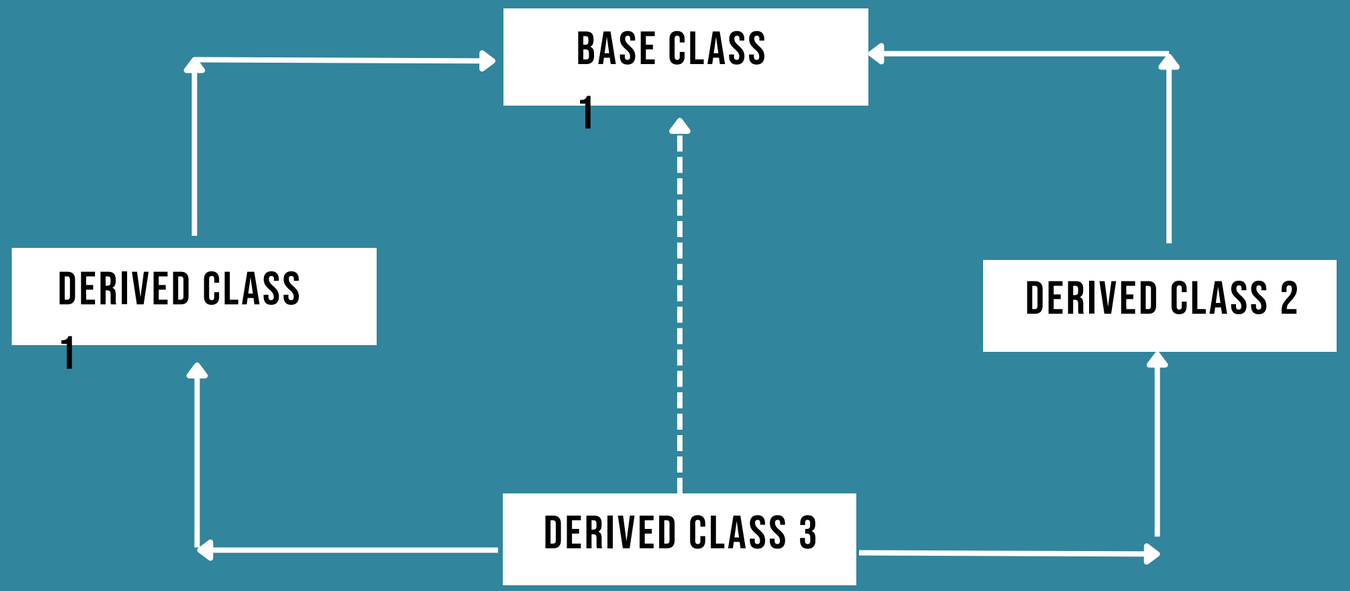
```
Class B
Enter the value of a:1
Enter the value of b:2
a = 1
b = 2
Class C
Enter the value of a:3
Enter the value of c:4
a = 3
c = 4
```

Hybrid inheritance

Hybrid Inheritance is a combination of more than one of the previous inheritance types(multiple, multilevel, hierarchical).

Multipath inheritance

When a base class is derived to two or more derived classes, and these derived classes are again combined as base class to another derived class, then this type of inheritance is known as multipath inheritance.



Multipath Inheritance and Virtual Base Class

Multipath inheritance can pose some problems in compilation. The public and protected members of grandparent are inherited into the child class twice, first, via parent 1 class and then via parent 2 class. Therefore, the child class would have duplicate sets of members of the grand- parent which leads to ambiguity during compilation and it should be avoided. It can be resolved adding virtual to the access specifier

```
class A //grandparent
{.....};
class B1 : virtual public A //parent 1
{.....};
class B1 : public virtual A //parent 2
{.....};
class C :public B1 , public B2 //child
{..... //only one copy of A will be inherited

.....
};
```

The keyword ***virtual*** and ***public*** or ***protected*** may be used in any order. After adding the keyword ***virtual*** while creating classes parent1 and parent2, it ensures that only one copy of the properties of class grandparent is inherited in the class child which is derived from classes parent1 and parent2.

Multipath Inheritance and Virtual Base Class

```
#include<iostream>
using namespace std;
 class Aclass
{
protected:
int a;
public:
void inputA()
{
cout<<"enter the value of a: ";
cin >> a;
}
};
```

```
class Bclass: virtual public Aclass
{
protected:
int b;
public:
void inputB()
{
cout<<"enter value of b: " ;
cin >> b;
}
};
```

```
class Cclass:public virtual Aclass
{
protected:
int c;
public:
void inputC()
{
cout<<"enter value of c: ";
cin >> c;
}
};
```

Multipath Inheritance and Virtual Base Class

```
class Derived: public Bclass, public Cclass
{
protected:
int d;
public:
void inputD()
{
cout<<"enter value of d:" << endl;
cin >> d;
}
void display(){
cout<<"the value of a is:" << a << endl;
cout<<"the value of b is:" << b << endl;
cout<<"the value of c is:" << c << endl;
cout<<"the value of d is:" << d << endl;
cout<<"the sum is:" << (a+b+c+d) << endl;
} };
```

```
int main(){
Derived D1;
D1.inputA();
D1.inputB();
D1.inputC();
D1.inputD();
D1.display();
return 0;
}
```

Output:

```
enter the value of a: 1
enter value of b: 3
enter value of c: 4
enter value of d: 5
the value of a is: 1
the value of b is: 3
the value of c is: 4
the value of d is: 5
the sum is: 13
```

Constructor Invocation in Single and Multiple Inheritances

Note: If a base class has a parameter constructor then there must be a constructor inside a derived class which passes a value to a base class parameters constructor through initialization list.

/*parameterized constructor through initialization list*/

```
#include<iostream>
using namespace std;
class Base1
{
int a;
public:
Base1(int a){
 cout<<"Base1 constructor:" << endl; this ->a = a;
}
~Base1(){
cout<<" Base1 destructor:" << a << endl;
}
};
```

```
class Base2
{
int c;
public:
Base2(int c)
{
this ->c = c;
cout<<"Base2 constructor:" << endl;
}
~Base2(){
 cout<<"Base2 destructor:" << c << endl;
}
};
```

Constructor Invocation in Single and Multiple Inheritances

```
class Derived: public Base2, public Base1
{
    int d;
    public:
      Derived(): Base1(10), Base2(20)
      {
          d=0;
          cout<<"Derived constructor:" << endl;
       }
      Derived( int x, int y, int z): Base1(x), Base2(y)
      {
          d=z; cout<<"Derived constructor:" << endl;

      }
      ~Derived()
      {
          cout<<"Derived destructor:" << d << endl;
      }
}
```

```
int main()
{
    Derived D1;
    Derived D2(1,2,3);
    return 0;
}
```

Output:

```
Base2     constructor:
Base1     constructor:
Derived   constructor:
Base2     constructor:
Base1     constructor:
Derived   constructor:
Derived   destructor:3
Base1     destructor:1
Base2     destructor:2
Derived  destructor:0
Base1   destructor:10
Base2 destructor:20
```

Constructor Invocation in Single and Multiple Inheritances

/* constructor in simple inheritance */

```
#include<iostream>
using namespace std;
class Base1{
int x;
public:
Base1(){
cout<<"Base constructor" << endl;
}
~Base1(){
cout<<"Base destructor" << endl; }
};
class Derived: public Base1{
int c;
public:
Derived(){
cout<<"Derived constructor" << endl;
}
~Derived(){
cout<<"Derived destructor" << endl;
}
};
int main() {
Derived D1;
return 0; }
```

Output:

Base constructor
Derived constructor
Derived destructor
Base destructor

Constructor Invocation in Single and Multiple Inheritances

/* access private of base class in derived class*/

```
#include<iostream>
using namespace std;
class Base1
{
    int x;
    public:
    void inputB(){
        cout<<"enter value to x of Base1:" << endl;
        cin >> x;
    }
    void display(){
        cout<<"the x of Base1 is:" << x;
    }
    int returnx(){
        return x;
} };
```

```
class Derived: public Base1
{
    int c;
    public:
    void inputD()
    {
        cout<<"enter the value of c:" << endl;
        cin >> c;
     }
     void displayD()
     {
        display();
        cout<<" the value of c is:" << c << endl;
        cout<<" the sum is:" << returnx() + c << endl;
      }
};
```

Constructor Invocation in Single and Multiple Inheritances

```
int main()
{
    Derived D1;
    cout<<"the num of data element inside derived class:" <<sizeof(D1)/sizeof(int) << endl;
    D1.inputB();
    D1.inputD();
    cout<<"Displaying the taken value:" << endl;
    D1.displayD();
    return 0;
}
```

Output:

```
enter value to x of Base1: 5
enter the value of c: 3
Displaying the taken value:
the x of Base1 is:5
the value of c is:3
the sum is:8
```

Destructor Invocation in Single and Multiple Inheritances

Destructor in single inheritance

When an object of derived class is created, first the base class constructor is invoked, followed by the derived class constructors. When an object of derived class expires, first the derived class destructor is invoked, followed by the base class destructor. Constructors and destructors of base class are not inherited by the derived class. Besides, whenever the derived class needs to invoke base class's constructor or destructor, it can invoke them through explicitly calling them.

Destructor in single inheritance

```
#include<iostream>
using namespace std;
class base
 {
 public:
 base()  {
cout<<"\nBase   Class   Constructor.";
}
~base()  {
cout<<"\nBase Class Destructor.";
} };
class derived : public base
{
 public:
 derived()  {
cout<<"\nDerived   Class   Constructor.";
}
~derived()  {
cout<<"\nDerived Class Destructor.";
} };
int main()
{
 Derived D1;
 return 0;
}
```

Output:

Base constructor
Derived constructor
Derived destructor
Base destructor

Destructor Invocation in Single and Multiple Inheritances

Destructor in multiple inheritance

In multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class. The destructor execute in reverse order to that of constructor i.e., destructor of the derived class is called and then the destructor of the base class which is last, in declaration of derived class and followed by base class in reverse order.

Destructor in multiple inheritance

```
#include<iostream>
using namespace std;
class base1
 {
public:  base1()
{
 cout<<”\nBase1 Class Constructor.”;
}
~base1()
{
cout<<”\nBase1 Class Destructor.”;
}
};
```

```
class base2
{
public:
base2()
{
cout<<”\nBase2 Class Constructor.”;
}
 ~base2()
{
cout<<”\nBase2 Class Destructor.”;
}
 };
```

Destructor in multiple inheritance

```
class derived : public base1, public base2
{
 public:
 derived()
{
 cout<<"\nDerived  Class  Constructor.";
}
~derived()
{
cout<<"\nDerived Class Destructor.";
}
};
```

```
int main()
{
 Derived D1;
 return 0;
}
```

Output:

```
Base1  Class  Constructor.
Base2  Class  Constructor.
Derived  Class  Constructor.
Derived  Class  Destructor.
Base2  Class  Destructor.
Base1 Class Destructor.
```

Multiple Choice questions on Inheritance

1. What is Inheritance in C++?

a) Wrapping of data into a single class
b) Deriving new classes from existing classes
c) Overloading of classes
d) Classes with same names

2. Which specifier makes all the data members and functions of base class inaccessible by the derived class?

a) private
b) protected
c) public
d) both private and protected

3. How many specifiers are used to derive a class?

a) 1
b) 2
c) 3
d) 4

4. If a class is derived privately from a base class then _____

a) no members of the base class is inherited
b) all members are accessible by the derived class
c) all the members are inherited by the class but are hidden and cannot be accessible
d) no derivation of the class gives an error

5. What is the order of Constructors call when the object of derived class B is declared, provided class B is derived from class A?

a) Constructor of A followed by B
b) Constructor of B followed by A
c) Constructor of A only
d) Constructor of B only

6. Which of the following can derived class inherit?

a) members
b) functions
c) both members & functions
d) classes

7. What is the order of Destructors call when the object of derived class B is declared, provided class B is derived from class A?

a) Destructor of A followed by B
b) Destructor of B followed by A
c) Destructor of A only
d) Destructor of B only

8. What is meant by multiple inheritance?

a) Deriving a base class from derived class
b) Deriving a derived class from base class
c) Deriving a derived class from more than one base class
d) Deriving a derived base class

9. Which of the following is the correct syntax for declaring a derived class in C++?

A) class Derived : public Base {};
B) class Derived inherits Base {};
C) class Derived extends Base {};
D) class Derived implements Base {};

10.Which constructor is called first in a multilevel inheritance hierarchy in C++?

a)The most derived class constructor
b) The base class constructor
c) Both are called at the same time
d) None of the above

11.Which of the following is true about hybrid inheritance in C++?

a) It combines two or more types of inheritance
b) It only allows for single inheritance
c) It only allows for multiple inheritance
d) None

12. In the case of virtual inheritance, which constructor is called first?
A) The constructor of the virtual base class.
B) The constructor of the derived class.
C) The constructor of the non-virtual base class.
D) The order of constructor calls is undefined.

13. Which of the following is true about the virtual keyword in C++?

a) It is used to create a virtual base class
b) It is used to create a virtual function
c) It is used to create a virtual object
d) None of the above

14.What is the syntax for calling a base class constructor from a derived class constructor in C++?

a) base::base();
b) super::super();
c) base::base(arguments);
d) None of the above

15.Which keyword is used to call the base class contructor in the derived class constructor?

a) this
b) super
c) base
d) none

16. What is the order of execution of constructors in a multilevel inheritance hierarchy in C++?

a)Constructors are executed in random order
b) From the most derived class to the base class
c) From the base class to the most derived class
d) None

THANK YOU