3.1 Operator overloading

3.1.1 Introduction

a. How to overload operators in C++
b. Overloading Unary operators
c. Overloading Binary operators

3.1.2 Operator overloading using Friend function

a. Overloading unary operator (-) using friend function
b. Overloading Binary operator (+) using friend function

3.1.3 Overloading special operators

a. Overloading [ ] operator
b. Overloading ( ) operator
c. Overloading -> operator
d. Overloading new and delete operator

3.1.4 Copy constructors

3.1.1 INTRODUCTION

Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. The meaning of operators are already defined and fixed for basic types like: int, float, double etc in C++ language. For example: If you want to add two integers then, + operator is used. But, for user-defined types(like: objects), you can define the meaning of operator, i.e, you can redefine the way that operator works. For example: If there are two objects of a class that contain string as its data member, you can use + operator to concatenate two strings. Suppose, instead of strings if that class contains integer data member, then you can use + operator to add integers. This feature in C++ programming that allows programmer to redefine the meaning of operator when they operate on class objects is known as operator overloading.

Overloaded operators are functions with special names the keyword 'operator' followed by the 'symbol' for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Syntax:

Operator operator_symbol

a. How to overload operators in c++?

```
class class_name
{
   ……..
public:
return_type operator sign (argument/s)
{
    …….
}
……
};
```

The return type comes first which is followed by keyword operator, followed by operator sign,i.e., the operator you want to overload like: +, <, ++ etc. and finally the arguments is passed. Then, inside the body of you want perform the task you want when this operator function is called.

This operator function is called when, the operator (sign) operates on the object of that class class_name.

Example:

```
/* Simple example to demonstrate the working of operator overloading*/
#include <iostream>
using namespace std;
class temp
{
  private:
    int count;
  public:
     temp():count(5){  }
     void operator ++() {
      count=count+1;
     }
     void Display() { cout<<"Count: "<<count; }
};
int main()
{
   temp t;
   ++t;       /* operator function void operator ++() is called */
   t.Display();
   return 0;
}
```

OUTPUT:

Count: 6

Explanation:

In this program, an operator function void operator ++ () is defined(inside class **temp**), which is invoked when ++ operator operates on the object of type **temp**. This function will increase the value of count by 1.

Note:

1. Operator overloading cannot be used to change the way operator works on built-in types. Operator overloading only allows to redefine the meaning of operator for user-defined types.
2. There are two operators assignment operator(=) and address operator(&) which does not need to be overloaded. Because these two operators are already overloaded in C++ library. For example: If **obj1** and **obj2** are two objects of same class then, you can use codeobj1=obj2; without overloading = operator. This code will copy the contents object of **obj2** to **obj1**. Similarly, you can use address operator directly without overloading which will return the address of object in memory.

3. Operator overloading cannot change the precedence of operators and associativity of operators. But, if you want to change the order of evaluation, parenthesis should be used.
4. Not all operators in C++ language can be overloaded. The operators that cannot be overloaded in C++ are ::(scope resolution), .(member selection), .*(member selection through pointer to function) and ?:(ternary operator).

Almost any operator can be overloaded in C++. However there are few operators which cannot be overloaded. **Operators that are not overloaded** are follows

- scope operator - **::**
- **sizeof**
- member selector - **.**
- member pointer selector - *****
- Ternary operator - **?:**

b. Overloading unary operators:

Whenever an unary operator is used, it works with one operand. Operators ++ and -- are the two types of unary operators which can be overloaded.

Example:

```
#include<iostream>
using namespace std;
class complex
{
    int a,b,c;
   public:
      complex(){}
      void getvalue()
      {
            cout<<"Enter the Two Numbers:";
            cin>>a>>b;
      }
     void operator++()
     {
            a=++a;
            b=++b;
      }
      void operator--()
      {
            a=--a;
```

```
            b=--b;
      }
      void display()
      {
            cout<<a<<"+\t"<<b<<"i"<<endl;
       }
};

int main()
{
    complex obj;
    obj.getvalue();
    obj++;
    cout<<"Increment Complex Number\n";
    obj.display();
    obj--;
    cout<<"Decrement Complex Number\n";
    obj.display();
}
```

Output:

Enter the two numbers: 3 6

Increment Complex Number

4 +7i

Decrement Complex Number

3 + 6i

c. Overloading binary operators:

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very frequently like addition (+) operator, subtraction (-) operator and division (/) operator.

Following example explains how addition (+) and subtraction (-) operators can be overloaded. Similar way, you can overload division (/) operators.

Example:

```
#include<iostream>
class complex
{
          int a,b;
```

```
  public:
        void getvalue()
        {
          cout<<"Enter the value of Complex Numbers a,b:";
          cin>>a>>b;
        }
        complex operator+(complex ob)
        {
                complex t;
                t.a=a+ob.a;
                t.b=b+ob.b;
                return(t);
        }
        complex operator-(complex ob)
        {
                complex t;
                t.a=a-ob.a;
                t.b=b-ob.b;
                return(t);
        }
        void display()
        {
                cout<<a<<"+"<<b<<"i"<<"\n";
        }
};
int main()
{
  complex obj1,obj2,result,result1;
 obj1.getvalue();
  obj2.getvalue();
 result = obj1+obj2;
  result1=obj1-obj2;
 cout<<"Input Values:\n";
  obj1.display();
  obj2.display();
  cout<<"Result:";
  result.display();
 cout<<"Result1:";
 result1.display();
 return 0;
```

```
}
```

Output:

```
Enter the value of Complex Numbers a, b
4          5
Enter the value of Complex Numbers a, b
2          2
Input Values
4 + 5i
2 + 2i
Result
6 +  7i
2 +  3i
```

3.1.2 OPERATOR OVERLOADING USING FRIEND FUNCTION

The friend functions are more useful in operator overloading. They offer better flexibility, which is not provided by the member function of the class. The difference between member function and friend function is that the member function takes argument explicitly. On the contrary, the friend function needs the parameters to be explicitly passes. The syntax of operator overloading with friend function is as follows:

```
friend return-type operator operator-symbol (variable1, variable2)
{
        Statement1;
        Statement2;
}
```

The keyword **friend** precedes function prototype declaration. It must be written inside the class. The function can be defined inside or outside the class. The argument used in friend function is similar to normal function; the only difference being that friend function can access private member of the class through the objects. Friend function has no permission to access private members of a class directly. However, it can access the private members through objects of the same class.

a. Overloading unary operator (-) using friend function:

```
#include <iostream>
using namespace std;
class complex
{
float real, imag;
```

```
public:
complex()
{
        real=imag=0;
}
Complex (float r, float i)
{
        real=r;
        imag=i;
}
friend complex operator –(complex c)
{
        c.real= -c.real;
        c.imag= -c.imag;
        return c;
}
void display()
{
        cout<<"\n"<<real;
        cout<<"\n"<<imag;
}
};
void main()
{
        complex c1(1.5, 2.5);
        c1.display();
        c2=-c1;
        cout<<"After negation\n";
        c2.display();
}
```

OUTPUT:

Real: 1.5
Imag: 2.5
After negation
Real: -1.5
Imag: -2.5

Explanation:

In the above program, operator – is overloaded using friend function. The operator function is defined as friend. The statement c2=-c1 invokes the operator function. The statement also

returns the negated values of c1 without affecting actual value of c1 and assign it to object c2.

The negation operation can also be used with an object to alter its own data member variables. In such case, the object itself acts as a source and destination object.

b. Overloading binary operator (+) using friend function:

```
#include <iostream>
using namespace std;
class point {
  int x, y;
public:
  point() {} // needed to construct temporaries
  point(int px, int py) {
    x = px;
    y = py;
  }
  void show() {
    cout << x << " ";
    cout << y << "\n";
  }
  friend point operator+(point op1, Point op2); // now a friend
};

// Now, + is overloaded using friend function.
point operator+(Point op1, Point op2)
{
  point temp;
  temp.x = op1.x + op2.x;
  temp.y = op1.y + op2.y;
  return temp;
}
int main()
{
  point ob1(10, 20), ob2( 5, 30);
  ob1 = ob1 + ob2;
  ob1.show();
  return 0;
}
```

OUTPUT:

15 50

3.1.3 OVERLOADING SPECIAL OPERATORS

a. Overloading [ ] operator:

The subscript operator [ ] is normally used to access array elements. This operator can be overloaded to enhance the existing functionality of C++ arrays.

In C++, the [ ] is considered a binary operator when you are overloading it. Therefore, the general form of a member operator [ ]( ) function is as shown here:

```
type class-name::operator[](int i)
{
// . . .
}
```

Technically, the parameter does not have to be of type int, but an operator[ ]( ) function is typically used to provide array subscripting, and as such, an integer value is generally used. Following are some useful facts about overloading of []:

1) Overloading of [] may be useful when we want to check for index out of bound.
2) We must return by reference in function because an expression like “arr[i]” can be used an lvalue.

Syntax:

```
Datatype &operator[] (datatype);
```

Given an object called O, the expression

```
O[3]
```

translates into this call to the operator[ ]( ) function:

```
O.operator[](3)
```

That is, the value of the expression within the subscripting operators is passed to the operator[ ]( ) function in its explicit parameter. The this pointer will point to O, the object that generated the call.

Example:

```
#include <iostream>
using namespace std;
const int SIZE = 10;

class array
{
```

```
  private:
    int arr[SIZE];
  public:
    array()
    {
      register int i;
      for(i = 0; i < SIZE; i++)
      {
        arr[i] = i;
      }
    }
    int &operator[](int i)
    {
       if( i > SIZE )
       {
          cout << "Index out of bounds" <<endl;
          // return first element.
          return arr[0];
       }
       return arr[i];
    }
};
int main()
{
  safearay A;
  cout << "Value of A[2] : " << A[2] <<endl;
  cout << "Value of A[5] : " << A[5]<<endl;
  cout << "Value of A[12] : " << A[12]<<endl;

  return 0;
}
```

OUTPUT:

```
Value of A[2] : 2
Value of A[5] : 5
Index out of bounds
Value of A[12] : 0
```

b. Overloading () operator:

The function call operator () can be overloaded for objects of class type. When you overload ( ), you are not creating a new way to call a function. Rather, you are creating an operator function that can be passed an arbitrary number of parameters.

Example:

```
#include <iostream>
using namespace std;
class function{
int mark;
public:
   function(int m){
      cout << "constructor is called"<<endl;
      mark = m;
   }
 void display(){
   cout << "hey i got "<< mark <<" marks" << endl;
   }
  function operator()(int mk){
      mark = mk;
       cout << "operator function is called"<<endl;
      return *this;
   }
};
int main()
{
   function obj(85);
   obj.display();
   obj(44);
   obj.display();
   return 0;
}
```

OUTPUT:

```
Constructor is called
Hey I got 85 marks
Operator function is called
Hey I got 44 marks
```

b. Overloading -> operator:

The class member access operator (->) can be overloaded but it is bit trickier. It is defined to give a class type a "pointer-like" behavior. The operator -> must be a member function. If used, its return type must be a pointer or an object of a class to which you can apply.The operator-> is used often in conjunction with the pointer-dereference operator * to implement "smart pointers." These pointers are objects that behave like normal pointers except they perform other tasks when you access an object through them, such as automatic object deletion either when the pointer is destroyed, or the pointer is used to point to another object.

The dereferencing operator-> can be defined as a unary postfix operator. That is, given a class:

```
class Ptr{

  //...

  X * operator->();

};
```

Objects of class Ptr can be used to access members of class X in a very similar manner to the way pointers are used. For example:

```
void f(Ptr p )

{

  p->m = 10 ; // (p.operator->())->m = 10

}
```

The statement p->m is interpreted as (p.operator->())->m. Using the same concept, following example explains how a class access operator -> can be overloaded.

Example:

```
#include <iostream>
using namespace std;
class arrow{
int mark;
public:
   arrow(int m){
   mark = m;
   }
void display(){
   cout << "hey i got "<< mark <<" marks" << endl;
   }
```

```
  arrow *operator->(){
   return this;
   }
};
int main()
{
arrow arrowobj(65);
arrowobj.display();
arrowobj->display();
return 0;
}
```

OUTPUT:

```
hey i got 65
hey i got 65
```

c. Overloading new and delete operator

The new and delete operators can also be overloaded like other operators in C++. There are interesting possibilities of doing the same.

Note:

What is the need for overloading new and delete?

You may sometimes need to specialize the storage mechanism for a class for example;

a. You want to speed up the object creation mechanism
b. You may want to use storage object in already allocated memory for some other objects
c. C++ doesn't provide garbage collection as in java and c#, by overloading new and delete provide garbage collection for our objects

However, some care should be taken regarding the parameters to accept, value to return and place to declare. This article covers all possible forms of overloading these operators and their uses.

Given below is a simple sample demonstrating overloaded new and delete:

```
void* operator new(size_t num)
{
        …
        return pointer_to_memory;
}

void operator delete(void *ptr)
{
```

```
        free(ptr);
}
```

Observe the following:

1. The overloaded new operator receives a parameter num of type size_t. This is the number of bytes of memory to be allocated. The compiler calculates and sends this to us!
2. The return type of the overloaded new must be void*. It is expected to return a pointer to the beginning of the block of memory allocated. Note that after our overloaded new returns, the compiler then automatically calls the constructor also as applicable.
3. The overloaded delete operator receives a parameter ptr of type void*. This is the pointer the user is trying to delete.
4. The overloaded delete operator should not return anything.
5. In this sample implementation, since the focus is only on showing how to overload, we have done the memory allocation and deallocation using malloc() and free()functions. In real life situations, we would prefer to do something more than this!
6. So far we have been considering overloading new and delete in the global scope. It is also possible for us to overload new and delete within classes. This would restrict their scope and operations to that class. Perhaps a simple illustration will throw more light:

```
class A
{
public:
   void* operator new(size_t num)
   {
      //...
   }
};

class B
{
public:
   void* operator new(size_t num)
   {
```

```
    //...
  }
};

class C
{
};

int main()
{
  A *a = new A(); // Will call A's new operator
  B *b = new B(); // Will call B's new operator
  C *c = new C(); // Will call the global new operator
  char *ptr = new char;

  //...
}
```

As can be seen from the program, when the new operator is used to create an instance of A dynamically, the overloaded new operator of class A is used. The same rule holds good for class B. However, in the case of instantiating class C, the global new is used since there is no overloaded new operator in that class. The same rules hold good for the delete operator too. Thus, new and delete overloads can be present on a per-class basis.

Within these “local” new and delete overloads, if you wish to call the global new to allocate memory, you can continue to use the scope resolution operator (::) like this: ::new.

Example:

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

class student
{
string name;
int age;
public:
student()
{}
student (string name, int age)
{
```

```
this->name=name;                    //this pointer
this->age=age;
}
void display()
{
 cout<<"name and age"<<name<<"\t"<<age<<endl;
}
void* operator new(size_t size)     //size_t is going to return the bytes of memory
allocated
{
void* pointer;
cout<<"overloaded new with size"<<endl<<size;
pointer = malloc(size);                    //allocate memory using malloc function &
returns address
return pointer;
}
void operator delete (void* pointer)  //overload delete operator, need address to delete
memory
{
 cout<<"overloaded delete"<<endl;
free(pointer);                      //free memory pointed to by pointer
}
};
int main()
{
student *s1,*s2;
s1 = new student("reva",20);
s2 = new student("university",40);
s1->display();
s2->display();
delete s1;
delete s2;
return 0;
}
```

OUTPUT:

```
Overloaded new with size 8
Overloaded new with size 8
Name and age reva 20
Name and age university 40
Overloaded delete
```

Overloaded delete

Explanation:
In the above program new and delete operators are overloaded. 'new' operator creates objects as well as allocates memory for the object. As mentioned in the comments, pointers (s1 & s2) to class student are created and memory is allocated by calling constructors and passing name and age as arguments. Statement s1 = new student ("reva",20) is going to call overloaded operator new i.e. statement void* operator new (size_t size) and allocate specialized memory. Statement delete s1 is going to call overloaded operator delete operator releases memory allocated by new operator by calling void operator delete (void* pointer). Same is followed for s2 as well.

3.1.4 Copy constructors

One of the most important forms of an overloaded constructor is the copy constructor. Defining a copy constructor can help you prevent problems that might occur when one object is used to initialize another. Let's begin by restating the problem that the copy constructor is designed to solve. By default, when one object is used to initialize another, C++ performs a bitwise copy (identical copy). That is, an identical copy of the initializing object is created in the target object. Although this is perfectly adequate for many cases—and generally exactly what you want to happen—there are situations in which a bitwise copy should not be used. One of the most common way is when an object allocates memory when it is created.

For example, assume a class called MyClass that allocates memory for each object when it is created, and an object A of that class. This means that A has already allocated its memory. Further, assume that A is used to initialize B, as shown here:

MyClass B = A;

If a bitwise copy is performed, then B will be an exact copy of A. This means that B will be using the same piece of allocated memory that A is using, instead of allocating its own. Clearly, this is not the desired outcome. For example, if MyClass includes a destructor that frees the memory, then the same piece of memory will be freed twice when A and B are destroyed! The same type of problem can occur in two additional ways:

- First, when a copy of an object is made when it is passed as an argument to a function
- Second, when a temporary object is created as a return value from a function.

Remember, temporary objects are automatically created to hold the return value of a function and they may also be created in certain other circumstances. To solve the type of problem just described, C++ allows you to create a copy constructor, which the compiler

uses when one object initializes another. Thus, your copy constructor bypasses the default bitwise copy. The most common general form of a copy constructor is,

```
classname (const classname &o) {
// body of constructor
}
```

Here, o is a reference to the object on the right side of the initialization. It is permissible for a copy constructor to have additional parameters as long as they have default arguments defined for them. However, in all cases the first parameter must be a reference to the object doing the initializing.

It is important to understand that C++ defines two distinct types of situations in which the value of one object is given to another. The first is assignment. The second is initialization, which can occur any of three ways:

- When one object explicitly initializes another, such as in a declaration
- When a copy of an object is made to be passed to a function
- When a temporary object is generated (most commonly, as a return value)

The copy constructor applies only to initializations. For example, assuming a class called myclass, and that y is an object of type myclass, each of the following statements involves initialization.

```
myclass x = y; // y explicitly initializing
x func(y); // y passed as a parameter
y = func(); // y receiving a temporary, return object
```

Example:

```
#include<iostream>
using namespace std;
class Example       {
// Variable Declaration
int a,b;
public:
//Constructor with Argument
Example(int x,int y) {
// Assign Values In Constructor
   a=x;
   b=y;
   cout<<"\nIm Constructor";
}
void Display()    {
cout<<"\nValues :"<<a<<"\t"<<b;
}
};
```

```
int main()          {
Example Object(10,20);
//Copy Constructor
Example Object2=Object;
// Constructor invoked.
Object.Display();
Object2.Display();
return 0;
}
```

OUTPUT:

```
Im Constructor
Values :10      20
Values :10      20
```

Explanation:

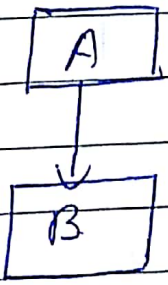
In this case, object2= object performs the assignment operation. If = is not overloaded (as it is not here), a bitwise copy will be made. Therefore, in some cases, you may need to overload the = operator as well as create a copy constructor to avoid certain types of problems.

Camlin | Page
Date / /

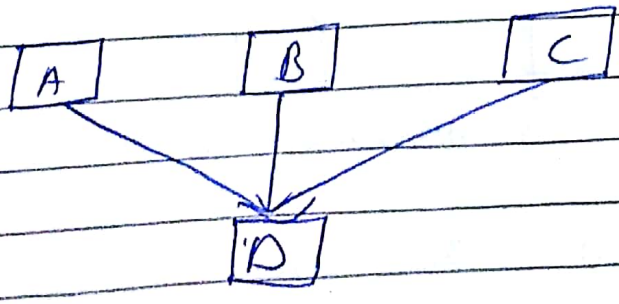
Chapter 2 : Inheritance

Q1 Inheritance → The process in which new classes that are created are able to ~~inherit~~ derive properties from an existing class.
→ The existing class is called base class and the new class is called derived class.
→ Five different types :-

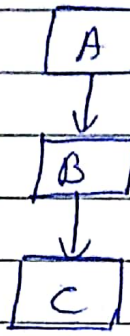
Q5 (i) Single Inheritance → Single derived class inherits from single base class.



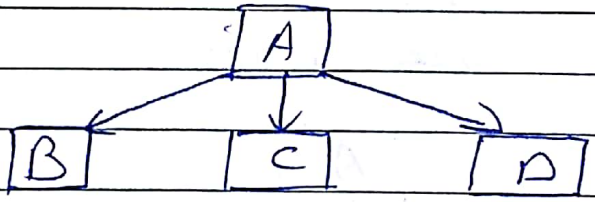
(ii) Multiple Inheritance → Single derived class inherits from multiple base classes



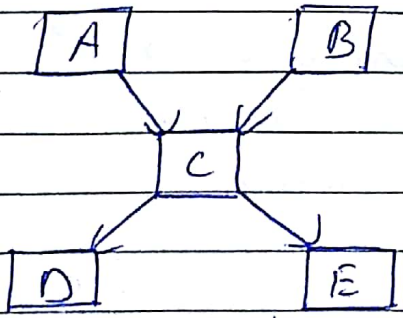
(iii) Multilevel Inheritance → One derived class acts as the base class for another derived class.



(iv) Hierarchical Inheritance → One ~~der~~ base class has multiple derived classes.



(v) Hybrid Inheritance → Combination of two or more types of inheritance.



→ Ex: Combination of multiple and hierarchical

→ Members of the base class can be accessed by objects of the derived class provided they are not private.

Short notes on each type of inheritance

① Single Inheritance → Simplest form of inheritance in which a derived class inherits from one base class only.

→ Syntax:

```
class A
{
};
class B : public A
{
};
```

→ Ex Program :-

```
#include <iostream>
using namespace std;
class animal
{
    public : int legs = 4;
};
class cat : public animal //Public inheritance
{
    public : int tail = 1;
};
int main ()
{
    cat obj;
    cout << obj.legs << obj.tail;
    return 0;
}
```

② Multiple Inheritance → A derived class inherits from more than one base class
→ Constructors are called in the same order as that of inheritance

→ Syntax :
```
class A
{
};
class B
{
};
class C : public A, public B;
{
}
```

→ Example :
```
#include <iostream>
using namespace std;
class A
{
    public:
        A()
        {
            cout << "A's constructor called \n";
        }
};
class B
{
    public:
        B()
        {
            cout << "B's constructor called \n";
        }
};
```

```
class C : public A, public B
{ public:
    C()
    {
        cout << " C's constructor called \n " ;
    }
};
int main ()
{
    C c;
    return (0);
}
```

③ Hierarchial Inheritance → ~~One~~ Multiple derived class inherit from a single base class.

→ Syntax :
```
class A { };
class B : public A
{ };
class C : public A
{ };
```

→ Ex :-
```
#include <iostream>
using namespace std;
class number
{
    int num;
public:
    void getnum (int x)
    { num = x;
      return num; } };
```

```
class square : public number
{
    public :
        int area ()
        {
            int num, ar ;
            num = getnum();
            ar = num * num;
            return ar;
        }
};
class cube : public number
{
    public :          ;cubearea()
        int cubearea
        {
            int num, area ;
            num = getnum();
            area = 6 * num * num;
            return area;
        }
};
int main ()
{
    square obj;
    cube obl;
    obj. getnum(5) ;
    obj. area();
    obl. getarea(3);
    obl. cubearea() ;
    return 0;
}
```

Camlin Page
Date / /

Q7 (4) Multilevel Inheritance -> One derived class is the base class for another derived class

-> Syntax :

```
class A { };
class B : public A
{ };
class C : public B
{ };
```

-> Ex :

```
#include<iostream>
using namespace std;
class A
{
    public :
        void display ()
            { cout << "A "; }
};
class B : public A
{
};
class C : public B
{
};
int main ()
{
    C obj ;
    obj. display ();
    return 0;
}
```

1/6/17 (5) Hybrid Inheritance → Combination of two or more types of inheritance.

→ Syntax for hybrid of single and multiple

```
class A
{    };
class B : public A
{    };
class C
{    };
class D : public B, public C
{    };
```

→ Ex:

```
#include <iostream>
using namespace std;
int a, b, c, d, e;
class A
{
    public:
    { void getAB()
      { cin >> a >> b;
      }
};
class B : public A
{
    public:
      void getC()
      { cin >> c;
      }
};
```

```
class C
{
    public :
        void getD()
        {
            cin >> d;
        }
};
class D : public B, public C
{
    public:
        int sum()
        { getAB(); getC(); getD();
          e = a+b+c+d;
        return e;
        }
};
int main()
{
    D obj;
    obj.sum();
    return 0;
}
```

// Q4 Base Class Access Control

-> Three access specifiers, namely public, private and protected

(1) Public Inheritance -> all public members become public members of the derived class.
-> Protected members become protected members of the ~~bas~~ derived class.
-> Private members are left as is, i.e. not accessible.

-> Ex :-

```
#include <iostream>
using namespace std;
class base
{
    int i, j;
public:
    void set(int x, int y)
        { i=x; j=y; }
    void show()
        { cout<<i<<j; }
};
class derived : public base
{
    int k;
public:
    void setk(int x)
        { k=x; }
    void showk()
```

```
    {cout << k; }
};
int main()
{
    derived obj;
    obj.set(1,2); // i=1, j=2
    obj.setk(3); // k=3
    obj.show(); // 1 2
    obj.showk(); // 3
    return 0;
}
```

-> The object obj of the derived class can freely access all ~~data~~ public members of the base class.

(ii) Private Inheritance -> All public and private members become private members of the derived class.

-> They can only be accessed inside the derived class.

-> Ex:-

```
#include <iostream>
using namespace std;
class base
{
    int i,j;
public:
    void setij(int x, int y)
        { i = x; j = y; }
    void showij()
        { cout << i << j; }
};
```

```
class derived :: private base
{
    public:
        void show()
        { cout << "Derived Class"; }
};
int main()
{
    derived obj;
    obj.setij(1, 2);
    obj.showij();
    obj.show;
    return(0);
}
```

→ The object of derived class cannot access setij() and showij() because they are private members of class derived and the main function is not a part of the derived or base class.

(iii) Protected Inheritance → Two cases, first is public and private inheritance of protected members and second is protected inheritance of public, private and protected members.

→ Case 1: (a) Public inheritance of the base class implies that protected members of the base class are private to the base class but accessible by the derived class

(b) If a derived class acts as the base class for another derived class, then the protected members of the initial base class are accessible by all the derived classes.

Ex:-

```
#include<iostream>
using namespace std;
class base
{   protected: int i=1, j=2;
    public:
        void show()
            { cout << "Base Class"; }
};
class derived1 : public base
{
    public:
        void show1()
            { cout << "Derived 1"; 
              cout << i << j; } };
class derived2 : public derived1
{
    public:
        void show2()
            { cout << "Derived 2"; 
              cout << i << j; } };
int main()
{
    derived2 obj;
    obj.show(); // Displays Base Class.
    obj.show1(); // Displays Derived 1, 1, 2
    obj.show2(); // Derived 2 1 2.
    return 0; }
```

(d) If the base class is inherited privately, then the object of the derived class would not be able to access the members of the base class as main() is not a part of the derived example.

Note: Refer example on previous page. Replace public (access specifier) with private and the program will not compile.

→ Case 2 : If the base class is inherited as protected, then all public & protected members become protected members of the derived class.

```
Ex: #include<iostream>
    using namespace std;
    class base
    {
      protected : int x;
      public :
        void show(int a)
          { x = a;
            cout << x;
          }
    };
    class derived : protected base
    {
      int y;
```

```
public:
    void showy(int b)
    {   y = b;
        cout << b;
    }
};
int main()
{
    derived obj;
    obj.showy(1); // Displays 1
    obj.show(20); // Invalid
    return(0);
}
```

Here, & show() is a protected member of derived. obj can't access it as main() is not a part of the derived class.

//Q#4 Inheritance Table

Access in Base	Base Inheritance	Access in Derived Class
Public Protected Private	Public	Public Protected No access
Public Protected Private	Protected	Protected Protected No Access
Public Protected Private	Private	Private Private No Access.

Inheriting Multiple Base Classes.

-> Syntax:

```
class derived-class : access specifier # base-class1
                      access specifier  base-class2
                      :
                      access specifier  base-classn
{ // Body
};
```

-> Ex: Pg 427-428 (A C++ Complete Reference)

Date / /

Constructors and Destructors in Inheritance

→ Called when objects are created
→ It is possible for both base and derived classes to have constructors and destructors.
→ Order of execution: Constructor of base class,
constructor of derived class,
destructor of derived class,
destructor of base class.

→ In case of multiple base class inheritance, the constructors are called in the order in which the base classes are inherited and the destructors follow the reverse order

→ Ex:

```
#include <iostream>
using namespace std;
class base1
{
    public:
    base1() { cout << " Constructing B1 "; }
    ~base1() { cout << " Destructing B1 "; }
};
class base2
{
    public:
    base2() { cout << " Constructing B2 "; }
    ~base2() { cout << " Destructing B2 "; }
};
```

```
class derived : public base1, public base2
{
  public:
    derived() { cout << "Constructing derived"; }
    ~derived() { cout << "Destructing derived"; }
};
int main()
{
    derived obj;
    return 0;
}
```

Output :-

```
Constructing B1
Constructing B2
Constructing Derived
Destructing Derived
Destructing B2
Destructing B1
```

Passing Parameters to Base Class Constructors

→ General Form:

```
derived constructor(arg-list) : base1(arg-list),
                                base2(arg-list),
                                :
                                basen(arg-list)
{ // Body of Constructor.
};
```

→ Derived class constructor must declare parameters required both by itself and the inherited base class's constructor.

→ Ex:

```
#include <iostream>
using namespace std;
class base1
{
    protected:
        int i;
    public:
        base1(int x) { i = x;
                       cout << "B1";
                     }
        ~base1() { cout << "Destructing B1"; }
};
class base2
{
    protected:
        int k;
    public:
        base2(int y) { k = y;
                       cout << "B2";
                     }
        ~base2() { cout << "Destructing B2"; }
};
class derived : public base1, public base2
{
    protected:
        int j;
    public:
        derived(int x, int y, int z): base1(y),
                                      base2(z)
        { j = x;
          cout << "Derived"; }
```

```
~derived () { cout << "Destructing Derived"; }
void show ()
{ cout << i << j << k; }
};
int main ()
{
    derived obj (1,2,3);
    obj.show (); // 2 , 3.
    return (0);
}
```

→ 2 gets passed to constructor of base1
3 gets passed to constructor of base2.

Output :-

```
B1
B2
Derived
2 , 3
Destructing Derived
Destructing B2
Destructing B1
```

Granting Access / Access Declaration

→ Used to change the access specification of inherited data members.
→ Two ways : (i) 'using' statement
(ii) Scope Resolution operator.
→ General form :-

```
base-class :: member ;
```

→ Ex:-

```
#include <iostream>
using namespace std;
class base
{
    int i;          // private to base
public:
    int j, k;
    void seti(int x) { i = x; }
    int geti() { return i; }
};
class derived : private base
{
public:
    base :: j;
    base :: seti;
    base :: geti;
    base :: i ;     // Invalid - private member of base

    int a;
};
int main()
{
    derived obj;
    obj.i = 5;      // Invalid
    obj.j = 10;
    obj.k = 15;     // Invalid - private member of derived

    obj.a = 20;
    obj.seti(5);    // i is set to 5
    obj.geti();     // 5
    return(0);
}
```

Camlin Page
Date / /

11Q10 Virtual Base Classes

-> Ambiguity can occur during multiple inheritance
-> Two ways of overcoming this issue.
-> One Way is using the scope resolution operator to mention which class the ~~fun~~ ~~fun~~ member belongs to.
-> Second is using virtual base classes
-> Virtual Base classes prevent multiple copies of the base class in an object.

-> Program: Page 441 -> Scope Resolution
Page 442 -> Virtual Base Class.