# WEEK 10
# MULTIPLE INHERITANCE

Dr. Yuzana Win

# TOPICS COVERED

- Levels of Inheritance
- Multiple Inheritance
- Friend Function and Classes

# LEVELS OF INHERITANCE

- Classes can be derived from classes that are themselves derived.

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

- Here B is derived from A, and C is derived from B. The process can be extended to an arbitrary number of levels – D could be derived from C, and so on.

# MULTIPLE INHERITANCE

# MULTIPLE INHERITANCE

- A class can be derived from more than one base class.
- This is called **multiple inheritance.**

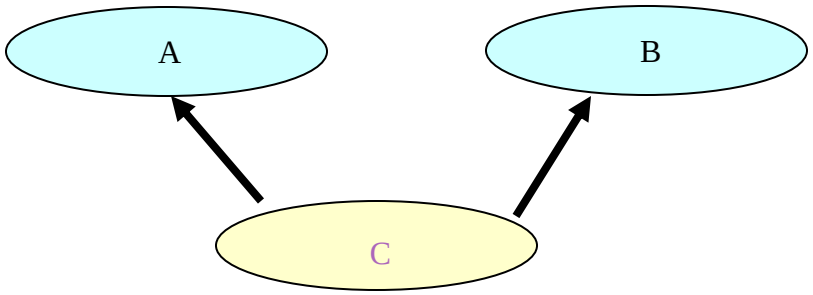
- As in use of multiple inheritance,
  - Constructor in Multiple inheritance
  - Member functions in Multiple inheritance
  - Ambiguity in Multiple Inheritance
  - Aggregation

# CONSTRUCTOR IN MULTIPLE INHERITANCE

```
class A
{   private:
          int a;
     public:
          A( )
          { a=0;}
           A(int p)
           { a=p;}
};
```

```
class B
{   private:
          int b;
     public:
          B( )
          { b=0;}
           B(int q)
           { b=q;}
};
```

```
class C: public A, public B
{   private:
          int c;
     public:
          C( ): A( ), B( )
          { c=0;}
           C(int p, int q, int r): A(p),
   B(q)
           { c=r;}
};
```




```
main( )
{   C objC1;
     C objC2(3, 4, 5);
}
```

# MEMBER FUNCTIONS IN MULTIPLE INHERITANCE

```
class A
{ private:
      int a;
   public:
      void set(int  p)
      { a=p;       }
       void display( )
       { cout<<a;   }
};
```

```
class C: public A,  public B
{    private:
           int c;
       public:
          void set(int  p, int q, int r)
          {  A:: set(p);
              B:: set(q);
              c=r;
          }
          void display( )
         {  A::display( );
             B:: display( );
             cout<<c;
         }
};
```

```
class B
{ private:
      int b;
   public:
      void set(int  q)
      { b=q;       }
       void display( )
       { cout<<b;   }
};
```

```
main( )
{    C objC;
      objC.set(3,4,5);
       objC.display();
}
```

Output:
3 4 5

# AMBIGUITY IN MULTIPLE INHERITANCE

```
class A
{ private:
     int a;
  public:
     void set(int  p)
     { a=p;       }
      void display( )
      { cout<<a;   }
};
```

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

```
class B
{ private:
     int b;
  public:
     void set(int  q)
     { b=q;       }
      void display( )
      { cout<<b;   }
};
```

```
main( )
{   C objC;
     objC.display(); // error- ambiguous-will not
    compile
     objC.A::display(); // OK
      objC.B::display(); // OK
}
```

# AGGREGATION (CLASSES WITHIN CLASSES)

```
class A
{
  private:
      int a;
   public:
      void set(int  p)
      { a=p;       }
       void display( )
       { cout<<a;   }
};
```

```
int main( )
{
      B objB;
       objB.set(5);
       objB.display();
}
```

```
class B
{
   private:
      A objA;
   public:
      void set(int  q)
      { objA.set(q);       }
       void display( )
       { objA.display();}
};
```

Output:
5

# CONSTRUCTORS, DESTRUCTORS, AND MULTIPLE INHERITANCE

Constructors are called in order of derivation, destructors in reverse order.

```
#include <iostream>
using namespace std;
class base {
public:
  base() { cout << "Constructing
base\n"; }
  ~base() { cout << "Destructing
base\n"; }
};
class derived1 : public base {
public:
   derived1() { cout << "Constructing
                       derived1\n"; }
  ~derived1() { cout << "Destructing
                       derived1\n"; }};
```

```
class derived2: public derived1 {
public:
   derived2() { cout << "Constructing
                        derived2\n"; }
   ~derived2() { cout << "Destructing
                         derived2\n"; }
};
int main()
{
  derived2 ob;
   // construct and destruct ob
  return 0;
}
```

**Output➔???**

# CONSTRUCTORS, DESTRUCTORS, AND MULTIPLE INHERITANCE

**Output**

```
Constructing base
Constructing derived1
Constructing derived2
Destructing derived2
Destructing derived1
Destructing base
```

# MULTIPLE INHERITANCE

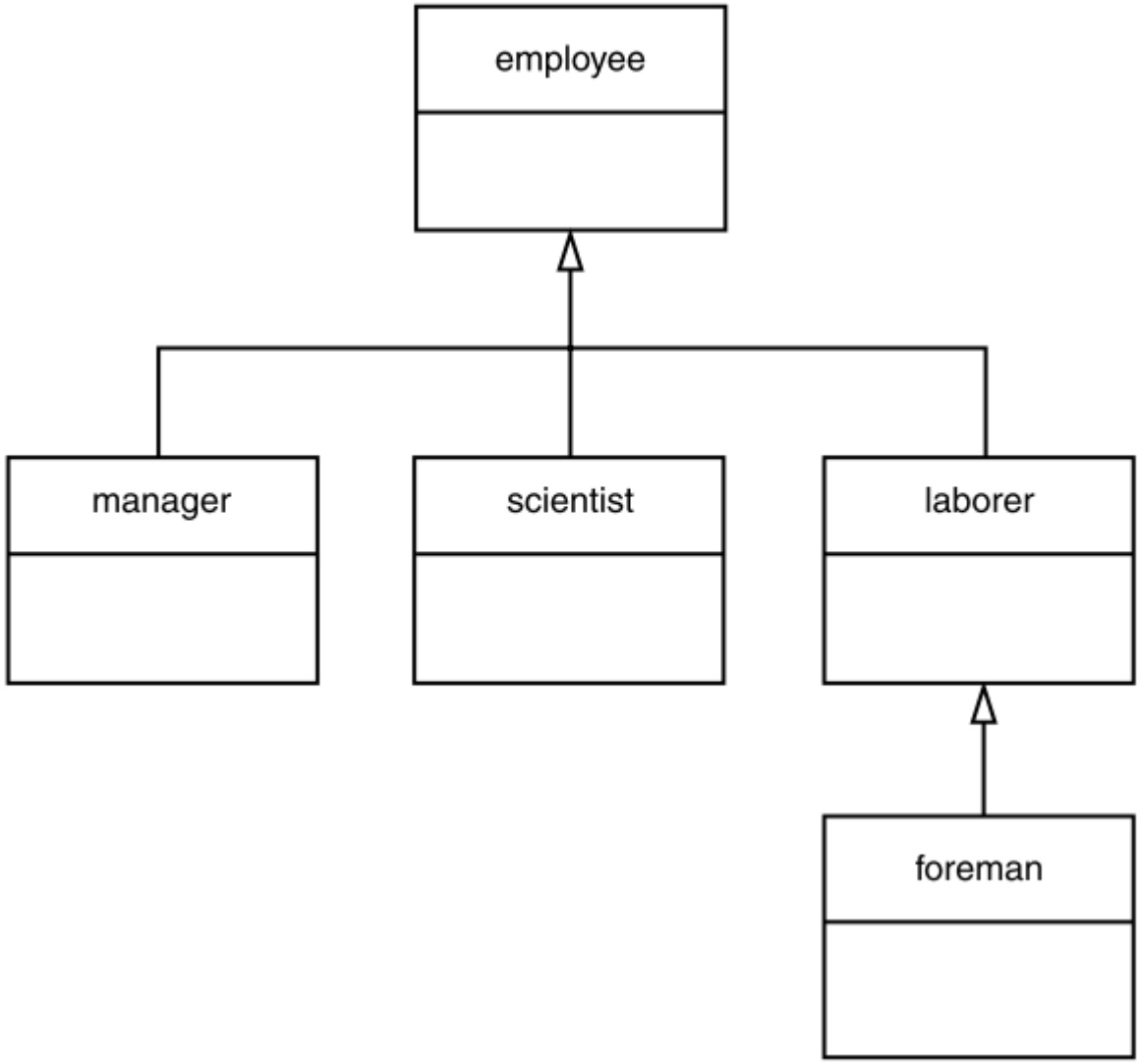



**Fig: UML Class Diagram for EMPLOYEE2**

# MULTIPLE INHERITANCE

```
// employ2.cpp
// multiple levels of inheritance
#include <iostream>
using namespace std;
const int LEN = 80;
class employee
{
private:
  char name[LEN];
  unsigned long number;
public:
   void getdata()
{ cout << "\n   Enter last name:";
  cin >> name;
  cout << "    Enter number:";
  cin  >> number;
}
void putdata() const
{cout << "\n   Name:" << name;
cout << "\n   Number:" << number;
} };
class manager : public employee
{
private:
     char title[LEN];
     double dues;
public:
     void getdata()
{ employee::getdata();
   cout << "   Enter title:";
   cin >> title;
   cout << "   Enter golf club dues:";
  cin >> dues; }
void putdata() const
{ employee::putdata();
   cout << "\n   Title:" << title;
   cout << "\n   Golf club dues:"<<
dues; } };
```

# MULTIPLE INHERITANCE

```
class scientist : public employee
{
private:
     int pubs;
public:
    void getdata()
    { employee::getdata();
      cout << “   Enter number of
      pubs:”;
     cin >> pubs;
     }
     void putdata() const
     { employee::putdata();
       cout << “\n   Number of
       publications:”<< pubs;
} };
class laborer : public employee
{ };
```

```
class foreman : public laborer
{
private:
    float quotas;
//percent of quotas met successfully
public:
   void getdata()
{
   laborer::getdata();
   cout << “   Enter quotas:”;
   cin >> quotas;
}
   void putdata() const
{
   laborer::putdata();
   cout << “\n   Quotas:”
   << quotas;
} };
```

# MULTIPLE INHERITANCE

```
int main()
{
laborer l1;
foreman f1;
cout << endl;
cout << "\nEnter data for laborer 1";
l1.getdata();
cout << "\nEnter data for foreman 1";
f1.getdata();
cout << endl;
cout << "\nData on laborer 1";
l1.putdata();
cout << "\nData on foreman 1";
f1.putdata();
cout << endl;
return 0;
}
```

**Output➔???**

# CONSTRUCTORS, DESTRUCTORS, AND MULTIPLE BASE INHERITANCE

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

```
class derived: public base1, public
base2 {
public:
   derived()
   { cout << "Constructing derived\n"; }
   ~derived()
   { cout << "Destructing derived\n"; }
};

int main()
{
   derived ob;
   // construct and destruct ob
   return 0;
}
```

**Output➔???**

# CONSTRUCTORS, DESTRUCTORS, AND MULTIPLE BASE INHERITANCE

**Output**

Constructing base1
Constructing base2
Constructing derived
Destructing derived
Destructing base2
Destructing base1

**if→ class derived: public base2, public base1**

**Output→???**

# VIRTUAL BASE CLASSES

## Ambiguity problem in multiple inheritance

```
#include <iostream>
using namespace std;
class base {
    public:  int i;
};
class derived1 : public base {
    public:  int j;
};
class derived2 : public base {
    public:  int k;
};
class derived3 : public derived1,
public derived2 {
    public:  int sum;
};
```

```
int main()
{
   derived3 ob;
  ob.i = 10; // this is ambiguous,
which i???
  ob.j = 20;
  ob.k = 30;
  // i ambiguous here, too
   ob.sum = ob.i + ob.j + ob.k;
  // also ambiguous, which i?
   cout << ob.i << " ";
   cout << ob.j << " " << ob.k << " ";
   cout << ob.sum;
   return 0;
}
```

# SOLUTION I: USING SCOPE RESOLUTION OPERATOR

```
#include <iostream>
using namespace std;
class base {
    public:  int i;
};
class derived1 : public base {
    public:  int j;
};
class derived2 : public base {
    public:  int k;
};
class derived3 : public derived1,
public derived2 {
    public:  int sum;
};
```

```
int main()
{
   derived3 ob;
   ob.derived1::i = 10;  // use
derived1's i
   ob.j = 20;
   ob.k = 30;
    // scope resolved
   ob.sum = ob.derived1::i + ob.j +
ob.k;
   // also resolved here
   cout << ob.derived1::i << " ";
   cout << ob.j << " " << ob.k << " ";
   cout << ob.sum;
   return 0;}
```

Output:

10 20 30 60

# SOLUTION II: USING VIRTUAL BASE CLASS

```
#include <iostream>
using namespace std;
class base {
    public:  int i;
};
class derived1 : virtual public base {
    public:  int j;
};
class derived2 : virtual public base {
    public:  int k;
};
class derived3 : public derived1,
public derived2 {
    public:  int sum;
};
```

```
int main()
{
    derived3 ob;
    ob.i = 10;  // now unambiguous
    ob.j = 20;
    ob.k = 30;
    // unambiguous
    ob.sum = ob.i + ob.j + ob.k;
    // unambiguous
    cout << ob.i << " ";
    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;
    return 0;
}
```

Output:

10 20 30 60

# ASSIGNMENT 1 : MULTI-LEVEN INHERITANCE

- Create a class, Point which has x, y as member data, and get and show as member functions.
- Create a class, Shape, which has aggregate Point object as member data, and get and show as member functions.
- Create a class, Polygon, which derives Shape class and has color as member data. Then, add "area" function that can calculate area of kind of polygon.
- Create a class, Triangle, which derives Polygon class and has base and height as member data. And , also override "area" function of base class.
- Create a class, Rectangle, which derives Polygon class and has length and width as member data. And, also override "area" function of base class.
- Create a class, Circle, which derives Polygon class and has radius as member data. And, also override "area" function of base class.
- The get and show functions are included in Polygon, Triangle, Rectangle and Circle, respectively.
- Write a main() function that test the hierarchy of the above classes.

# ASSIGNMENT 2 : MULTIPLE INHERITANCE

- Create a class, Employee which has empno, rank, department, salary as member data, and get and show as member functions.
- Create a class, Student, which has prno, name, course as member data, and get and show as member functions.
- Create a class, IMCEITS-Student, which derives the above two classes and adds "Search" function that can search student records by prno or course.
- Write a main() functon that gets 5 IMCEITS-Student records and search and display the student record according to the input value of prno or course.

# FRIENDS FUNCTIONS AND CLASSES

# IT'S GOOD TO HAVE FRIENDS

- A *friend* function of a class is defined outside the class's scope (i.e. **not** member functions), yet has the right to access the non-public members of the class.

- Single functions or entire classes may be declared as friends of a class.

- These are commonly used in operator overloading. Perhaps the most common use of friend functions is **overloading << and >>** for I/O.

- Declare as a friend, can access to private data members.

# FRIEND FUNCTION EXAMPLE

```
#include <iostream>
using namespace std;
class Rectangle {
    int width, height;
    public:
     void set_values (int, int);
     int area ()
       {return (width * height);}
  friend  Rectangle  duplicate
(Rectangle);
};
void  Rectangle::set_values (int a, int
b)
{
      width = a;
      height = b;
}
```

```
Rectangle duplicate (Rectangle  r)
{
    Rectangle rect;
    rect.width = r.width*2;
    rect.height = r.height*2;
    return (rect);
}

int main ()
{
    Rectangle rect, rectb;
    rect.set_values (2,3);
    rectb = duplicate (rect);
    cout << rectb.area();
    return 0;
}
```

Output:

24

# FRIEND FUNCTION EXAMPLE

```
class A;
class B{
      public:
            void display1(A);
            void display2(A);
};
class A{
      int a;
      friend void B::display1(A);
      friend void show(A);
      public:
            A(int i):a(i){}
};
```

```
void B::display1(A obj)    // friend function
{     //accessing private data of A from B's member functio
    cout<<endl<<"a="<<obj.a;
}
void B::display2(A obj)   // not a friend function
{     //accessing private data of A from B's member functio
     cout<<endl<<"a="<<obj.a;  //error
}
void show(A obj)//friend function
{    //accessing private data of A from global function
  cout<<endl<<"From global function, a="<<obj.a;
}
```

```
int main()
{      A obja(10);
        B objb;
        objb.display1(obja);
        show(obja);
        return 0;
}
```

# FRIEND FUNCTION: NOTES

- The friend functions can serve, for example, to conduct operations between two different classes.
- Generally, the use of friend functions is **out of an object-oriented programming** methodology.
- So whenever possible it is better to use members of the same class to perform operations with them.

# FRIEND CLASS

```
#include <iostream>
using namespace std;
class   Square;
class  Rectangle {
        int width, height;
       public:
          int area ()
         {return (width * height);}
         void convert (Square a);
};
class Square {
   private:
        int side;
   public:
       void set_side (int a)
      {side=a;}
      friend class CRectangle;
};

void  Rectangle::convert (Square a) {
        width = a.side;
        height = a.side;
 }

int main ()
{
       Square sqr;
       Rectangle rect;
       sqr.set_side(4);
       rect.convert(sqr);
       cout << rect.area();
       return 0;
}
```

# FRIEND CLASS

```
class A
{   private:
        int a;
        void show()
        {   cout<<a; }
    public:
        A(int i):a(i){}
        friend class B;
};
```

```
class B
{ public:
    void display1(A obj)
    {
        cout<< obj.a;//accessing class A's private data
        obj.show(); //accessing class A's private functions
    }
void display2(A obj)
    {
        cout<< obj.a;//accessing class A's private data
    }
};
```

```
main( )
{   A obja(10);
    B objb;
    objb.display1(obja);
    objb.display2(obja);
}
```

Output:

10 1010

# QUIZ

Q1. What will be the output?

```
#include<iostream>
using namespace std;
class abc {
   public:
      static int x;
      int i;

      abc() {
         i = ++x;
      }
   };
int abc::x;
main() {
   abc m, n, p;

   cout<<m.x<<" "<<m.i<<endl;
}
```

**A - 3 1**
**B - 3 3**
**C - 1 1**
**D - 1 3**

# QUIZ

Q2.
By default the members of the structure are
A - private
B - protected
C - public
D - Access specifiers not applicable for structures.

Q3.
Which operator is used to resolve the scope of the global variable?
A - –>
B - .
C - *
D - ::

Q4. A C++ program statements can be commented using
A - Single line comment
B - Multi line comment
C - Either (a) or (b)
D - Both (a) and (b).

Q5. What will be the output?

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

int main () {
   // local variable declaration:
   int x = 1;

   switch(x) {
   case 1 :
      cout << "Hi!" << endl;
      break;
   default :
      cout << "Hello!" << endl;
   }
}
```

**A - Hello**
**B - Hi!**
**C - HelloHi**
**D - Compile error**

Q6. What will be the output?

```
#include<iostream>

using namespace std;
main() {
   int a[] = {1, 2}, *p = a;

   cout<<p[1];
}
```

**A - 1**
**B - 2**
**C - Compile error**
**D - Runtime error**

Q7. A constructor that accepts _____ parameters is called the default constructor.

A. one
B. two
C. no
D. three

Q8. Destructor has the same name as the constructor and it is preceded by _____ .

A. !
B. ?
C. ~
D. $

Q9. Which constructor function is designed to copy objects of the same class type?

A. Create constructor
B. Object constructor
C. Dynamic constructor
D. Copy constructor

Q10. Which of the following statement is correct?

A. Constructor has the same name as that of the class.
B. Destructor has the same name as that of the class with a tilde symbol at the beginning.
C. Both A and B.
D. Destructor has the same name as the first member function of the class.

Q11. Copy constructor must receive its arguments by _____.

A. either pass-by-value or pass-by-reference
B. only pass-by-value
C. only pass-by-reference
D. only pass by address

Q12. A function with the same name as the class, but preceded with a tilde character (~) is called _____ of that class.

A. constructor
B. destructor
C. function
D. object

Q13. _____ used to make a copy of one class object from another class object of the same class type.

A. constructor
B. copy constructor
C. destructor
D. default constructor

Q14. Constructors _____ to allow different approaches of object construction.

A. cannot overloaded
B. can be overloaded
C. can be called
D. can be nested

Q15. Which of the following type of data member can be shared by all instances of its class?

A. Public
B. Inherited
C. Static
D. Friend