

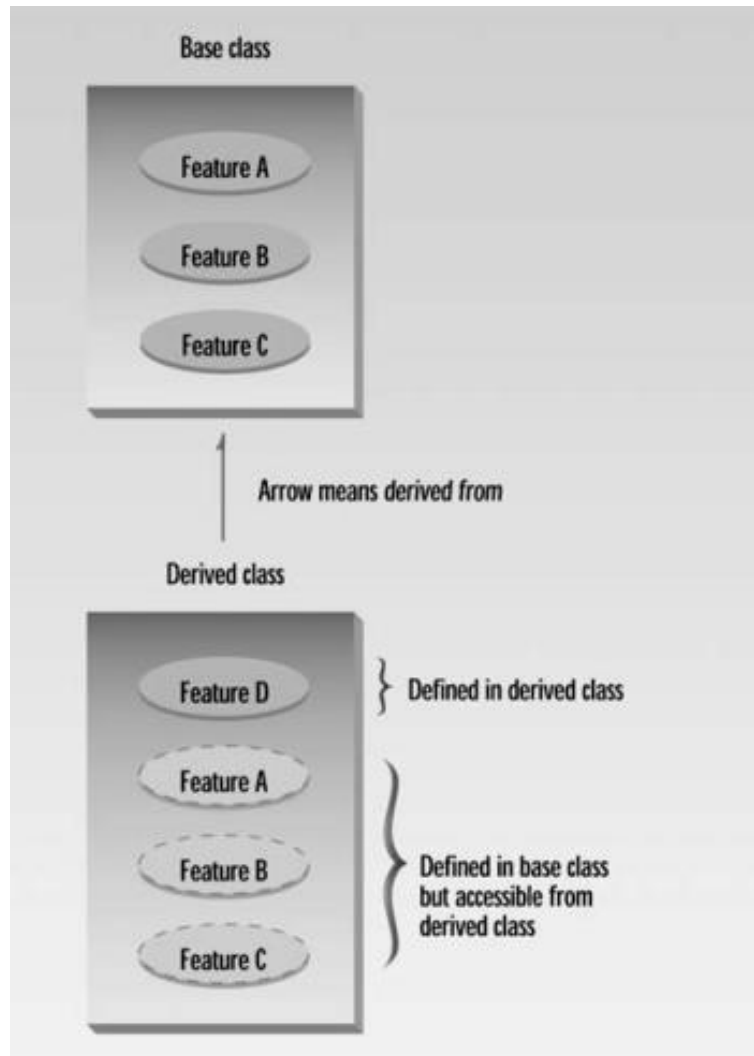
# Inheritance

Prof. Zankhana Dabhi,  
Department of Information Technology,  
D. D. University, Nadiad

# Inheritance

- Inheritance is the process of creating new classes, called derived classes, from existing or base classes.
- The derived class inherits all the capabilities of the base class but can add its own.
- The base class remain unchanged in this process.
- The most powerful feature of object-oriented programming, after classes themselves.

# Inheritance



# Inheritance

- Inheritance permits code reusability.
- Reusing existing code saves time and money and increases a program's reliability.
  - A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

# Counter Class

```
class Counter    {  
    protected:  
        int count;  
    public:  
        Counter() : count(0)    { }  
        Counter(int c) : count(c)    { }  
        int get_count() const    {  
            return count;    }  
        Counter operator ++ ()    {  
            return Counter(++count);    }  
};
```

# Example

- Suppose we want to use Counter Class for counting people entering a bank
  - We want to increment the count when they come in and decrement it when they go out, so that the count represents the number of people in the bank at any moment

# Solution 1

- Insert a decrement routine directly into the source code of the Counter class.
  - We might not want to do this.
    - The Counter class works very well and has undergone many hours of testing and debugging.
    - Insertion of decrement routine force the testing process need to be carried out again, and the routine may have bug which may require number of hours of debugging code that worked fine before we modified it.
  - We might not have access to its source code, especially if it was distributed as part of a class library.

# Solution 2

- To avoid the problems, mentioned in Solution 1, we can use inheritance.
  - Create a new class based on Counter, without modifying Counter itself.

# Derived Class : CountDn

```
class CountDn : public Counter    {  
    public:  
        Counter operator -- ()    {  
            return Counter(--count); }  
};
```

# Using Derived Class in main

```
int main()    {  
    CountDn c1;  
    cout << "\\nc1=" << c1.get_count();  
    ++c1; ++c1; ++c1;  
    cout << "\\nc1=" << c1.get_count();  
    --c1; --c1;  
    cout << "\\nc1=" << c1.get_count();  
    cout << endl;  
    return 0;  
}
```

# Accessing Base Class Members

- **Accessibility**
  - To know when a member function in the base class can be used by objects of the derived class.
- In the main method, we have written
  - **CountDn c1;**
    - Which will create c1 as **an object of class CountDn and initialized to 0.**
    - There is no constructor in the CountDn class, so what entity carries out the initialization?
    - if you don't specify a constructor, the derived class will use an appropriate constructor from the base class.
    - **There's no constructor in CountDn, so the compiler uses the no-argument constructor from Counter**

# Substituting Base Class Member Functions

- The object `c1` of the `CountDn` class also uses the `operator++()` and `get_count()` functions from the `Counter` class.
  - `++ c1;`
  - `cout << "\nc1=" << c1.get_count();`
- The compiler, not finding these functions in the class of which `c1` is a member, uses member functions from the base class.
- The `++` operator, the constructors, the `get_count()` function in the `Counter` class, and the `--` operator in the `CountDn` class all work with objects of type `CountDn`.

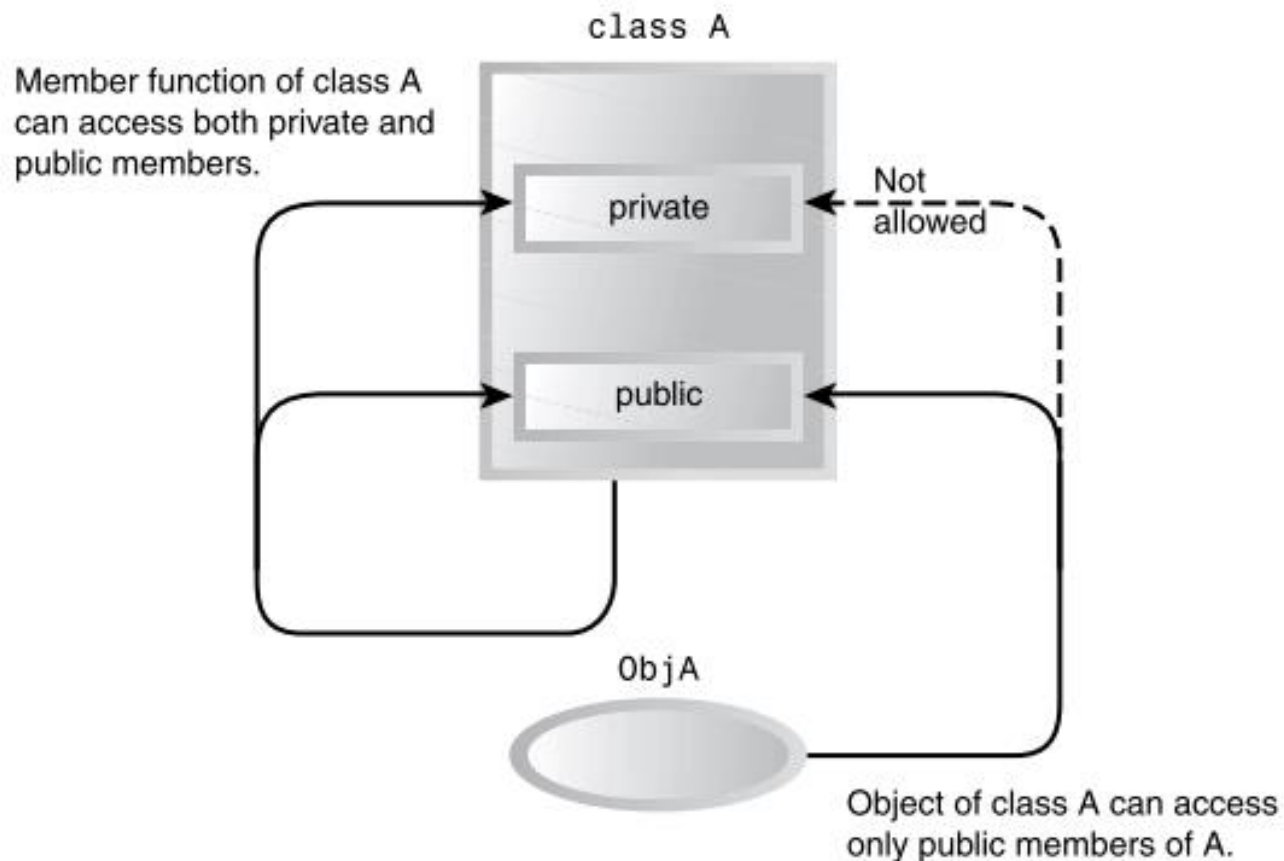
# Protected Access Specifier

- There is single change to the Counter class.
- The data member “count” is given a new specifier: protected, instead of private.

# Private & Public Access Specifier

- A member function of a class can always access class (data) members, whether they are public or private.
- But an object declared externally can only invoke (using the dot operator, for example) public members of the class. It's not allowed to use private members.

# Private & Public Access Specifier



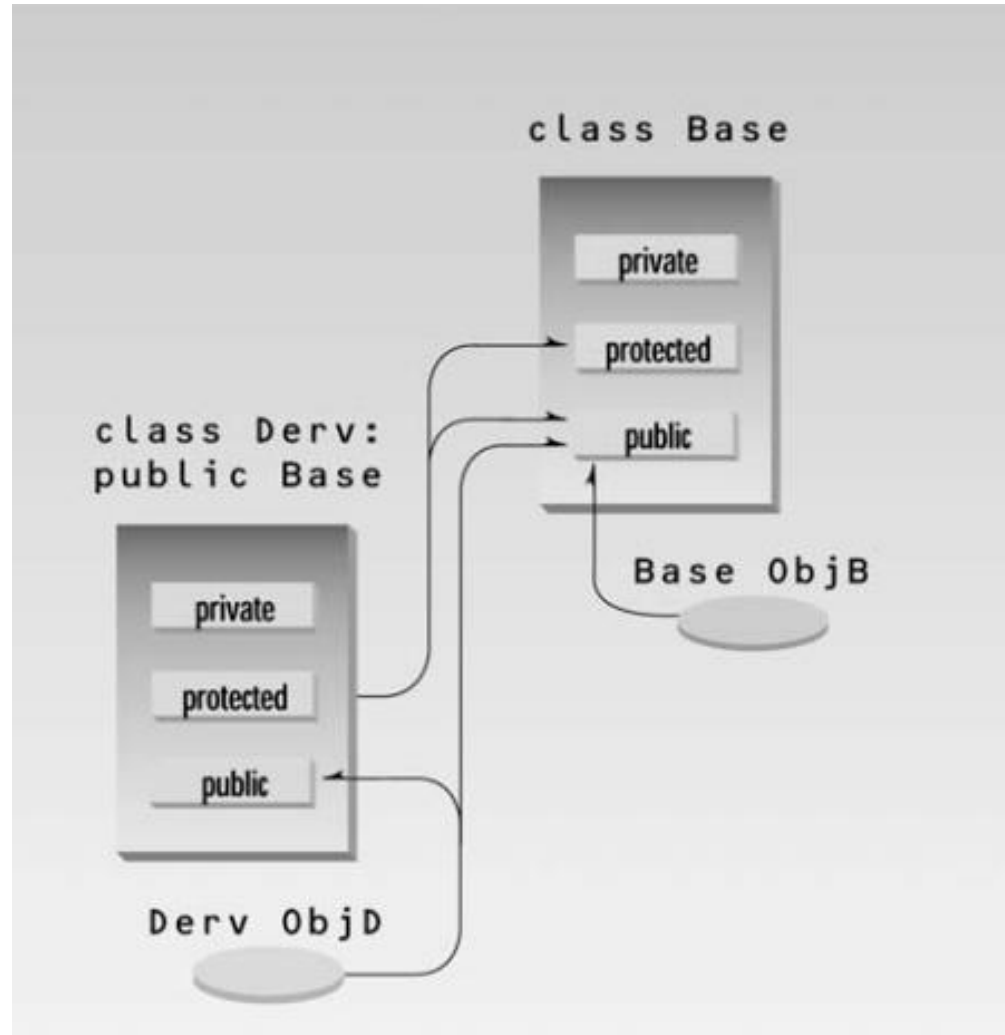
# Private & Public Access Specifiers

- Que. Can member functions of the derived class access members of the base class?
  - Can operator--() in CountDn access count in Counter?
- Ans. The member functions can access members of the base class if the members are public, or if they are protected. They can't access private members.

# Private, Public & Protected Access Specifier

- We don't want to make count public, since that would allow it to be accessed by any function anywhere in the program and eliminate the advantages of data hiding.
- A protected member can be accessed by member functions in its own class or in any class derived from its own class.
- It can't be accessed from functions outside these classes, such as main().

# Private, Public & Protected Access Specifier



# Private, Public & Protected Access Specifier

<i>Access Specifier</i>	<i>Accessible from Own Class</i>	<i>Accessible from Derived Class</i>	<i>Accessible from Objects Outside Class</i>
public	yes	yes	yes
protected	yes	yes	no
private	yes	no	no

# Remember !

- While creating a class that might be used, at any point in the future, as a base class for other classes, then any member data that the derived classes might need to access should be made protected rather than private.
- This ensures that the class is “inheritance ready.”

# Dangers of using Protected Access Specifier

- There is a disadvantage to making data members of class as protected.
  - For example, suppose you have written a class library, which you are distributing to the public. Any programmer who buys this library can access protected members of your classes simply by deriving other classes from them. This makes protected members considerably less secure than private members.

# Inheritance

- Inheritance doesn't work in reverse.
  - The base class and its objects don't know anything about any classes derived from the base class.
  - That means that objects of class Counter, such as c2, can't use the operator--() function in CountDn.
- The base class is also called the **superclass** and the derived class is called the **subclass**.
- Sometime the base class is also referred as the **parent** and the derived class as the **child**.

# Derived Class Constructors

- What happens if we want to initialize a CountDn object to a value? Can the one-argument constructor in Counter be used?
  - The answer is no.

# No Constructors in the base class and derived class

```
class B
{
};
class D:public B
{
    public:
        void fun()
        {
            cout<<"no constructor";
        }
};
int main()
{
    D objd;
    objd.fun();
}
```

**OP: no constructor**

# Constructor only in the base class

```
class B
{
    public:
        B()
        { cout<<"default constructor in base class is executed"; }
};
class D:public B
{
    public:
};
int main()
{
    D objd;
}
```

**OP: default constructor in base class is executed**

## Constructor only in the derived class

```
class B
{
    public:
};
class D:public B
{
    public:
        D()
        { cout<<"default constructor in derived class is executed"; }
};
int main()
{
    D objd;
}
```

**OP: default constructor in derived class is executed**

# Constructor in both base and derived class

```
class B
{
    public:
        B()
        { cout<<"default constructor in base class is executed"; }
};
class D:public B
{
    public:
        D()
        { cout<<"default constructor in derived class is executed"; }
};
int main()
{
    D objd;
}
```

**OP: default constructor in base class is executed**  
**default constructor in derived class is executed**

# Multiple constructors in base class and single constructor in derived class

```
class B
{
    public:
        B()
        { cout<<"default constructor in base class is executed"; }
        B(int a)
        { cout<<"one argument constructor in base class is executed"; }

};

class D:public B
{
    public:
        D(int a)
        { cout<<"one argument constructor in derived class is executed"; }

};

int main()
{
    D objd(5);
}
```

**OP: default constructor in base class is executed**  
**one argument constructor in derived class is executed**

## constructor in base and derived classes without default constructor

```
class B
{
    public:
        B(int a)
        {   cout<<"one argument constructor in base class is executed"; }
};
class D:public B
{
    public:
        D(int a)
        {   cout<<"one argument constructor in derived class is executed"; }
};
int main()
{
    D objd(5);
}
```

**OP: Error: "can not find default constructor to initialize base class B"**

# Explicit invocation in the absence of default constructor

```
class B
{
    public:
        B(int a)
        {   cout<<"one argument constructor in base class is executed"; }
};
class D:public B
{
    public:
        D(int a):B(a)
        {   cout<<"one argument constructor in derived class is executed"; }
};
int main()
{
    D objd(5);
}
```

**OP: one argument constructor in base class is executed**  
**one argument constructor in derived class is executed**

# Counter Class

```
class Counter    {  
    protected:  
        unsigned int count;  
    public:  
        Counter() : count(0)          { }  
        Counter(int c) : count(c)      { }  
        unsigned int get_count() const {  
            return count;              }  
        Counter operator ++ ()         {  
            return Counter(++count);   }  
};
```

# CountDn Class

```
class CountDn : public Counter {  
public:  
    CountDn() : Counter()      { }  
    CountDn(int c) : Counter(c) { }  
    CountDn operator -- ()     {  
        return CountDn(--count); }  
};
```

# Using Derived Class in main

```
int main()    {
    CountDn c1;
    CountDn c2(100);
    cout << "\\nc1=" << c1.get_count();
    cout << "\\nc2=" << c2.get_count();
    ++c1; ++c1; ++c1;
    cout << "\\nc1=" << c1.get_count();
    --c2; --c2;
    cout << "\\nc2=" << c2.get_count();
    CountDn c3 = --c2;
    cout << "\\nc3=" << c3.get_count();
    cout << endl;
    return 0;
}
```

# Derived Class Constructors

- The program uses two new constructors in the CountDn class
  - (1) `CountDn() : Counter() { }`
    - The constructor uses the function name following the colon. This construction causes the `CountDn()` constructor to call the `Counter()` constructor in the base class.
    - In `main()` when the line `CountDn c1;` gets executed,
      - The compiler will create an object of type `Count` and then call the `CountDn` constructor to initialize it. This constructor will in turn call the `Counter` constructor, which carries out the work. The `CountDn()` constructor could add additional statements of its own, but in this case it doesn't need to do.

# Derived Class Constructors

- We want to initialize any variables, whether they're in the derived class or the base class, before any statements in either the derived or base-class constructors are executed.
- By calling the base class constructor before the derived-class constructor starts to execute, we accomplish this.

# Derived Class Constructors

- The program uses two new constructors in the CountDn class
  - (2) CountDn c2(100);
    - uses the one-argument constructor in CountDn. This constructor also calls the corresponding one-argument constructor in the base class.
    - `CountDn(int c) : Counter(c) { }`
    - This construction causes the argument c to be passed from CountDn() to Counter(), where it is used to initialize the object
    - The one-argument constructor is also used in an assignment statement : `CountDn c3 = --c2;`

# Overriding Member Functions

- We can use member functions in a derived class which have the same name as those in the base class – referred as overriding of member functions

```
Class Array
{
    public:
        void insert() {}
        void f2() {}
};
Class Mod_array: public Array
{
    public:
        void insert() { Array::insert(); } //method overriding
        void f2(int x) {} //method hiding
        void f2() {}
};
Int main()
{
    Mod_Array b1;
    b1.insert(); // Array
    b1.Mod_insert(); // Mod_Array
    b1.f2(); // error
    b1.f2(5); //B
}
```

# Example : Stack

- Stack Class
  - If you tried to push too many items onto the stack, then data would be placed in memory beyond the end of the array.
  - If you tried to pop too many items, the results would be meaningless, since you would be reading data from memory locations outside the array.
- To overcome these limitations, a new class Stack2 is derived from Stack class.

# Stack Class

```
class Stack    {  
protected:    //NOTE: can't be private  
    enum { MAX = 3 };  
    int st[MAX];  
    int top;  
public:  
    Stack()    { top = -1; }  
    void push(int var) { st[++top] = var; }  
    int pop()   { return st[top--]; }
```

# Stack2 Class

```
class Stack2 : public Stack {
public:
    void push(int var) {           //error if stack is full
        if(top >= MAX-1) {
            cout << "\nError: stack is full"; exit(1);
        }
        Stack::push(var);
    }
    int pop() {
        if(top < 0) {           //error if stack empty
            cout << "\nError: stack is empty\n"; exit(1);
        }
        return Stack::pop();
    }
};
```

# Main()

```
int main()
{
    Stack2 s1;
    s1.push(11);
    s1.push(22);
    s1.push(33);
    cout << endl << s1.pop();
    cout << endl << s1.pop();
    cout << endl << s1.pop();
    cout << endl << s1.pop();
    cout << endl;
    return 0;
}
```

# How compiler know which function to call ?

- The Stack2 class contains two functions, push() and pop().
  - These functions have the same names, and the same argument and return types, as the functions in Stack.
- How does the compiler know which of the two push() functions to use?
  - Rule: When the same function exists in both the base class and the derived class, the function in the derived class will be executed. (This is true of objects of the derived class. Objects of the base class don't know anything about the derived class and will always use the base class functions.)

# How compiler know which function to call ?

- `S1.push(11)`
  - `s1` is an object of class `Stack2`, the `push()` function in `Stack2` will be executed, not the one in `Stack`.
  - The `push()` function in `Stack2` checks to see whether the stack is full. If it is, it displays an error message and causes the program to exit. If it isn't, it calls the `push()` function in `Stack`.

# Scope Resolution Operator

- How do push() and pop() in Stack2 access push() and pop() in Stack?
  - They use the scope resolution operator, ::, in the statements `Stack::push(var);` and `Stack::pop();`
  - Without the scope resolution operator, the compiler would think the push() and pop() functions in Stack2 were calling themselves, which would lead to program failure.

# Inheritance Example : Distance Class

- Derive a new class from Distance class.
  - This class will add a single data item to our feet-and-inches measurements: a sign, which can be positive or negative.

# Class Distance

```
enum posneg { pos, neg };    //for sign in DistSign
```

```
class Distance    {  
    protected:    //NOTE: can't be private  
        int feet;  
        float inches;  
    public:  
        Distance() : feet(0), inches(0.0)    { }  
        Distance(int ft, float in) : feet(ft), inches(in) { }  
        void getdist()    {  
            cout << "\nEnter feet: "; cin >> feet;  
            cout << "Enter inches: "; cin >> inches;  
        }  
        void showdist() const    {  
            cout << feet << "\'-" << inches << '\n';  
        }  
};
```

```

class DistSign : public Distance    {
private:
    posneg sign;           //sign is pos or neg
public:
    DistSign() : Distance()    {
        sign = pos;    }
    DistSign(int ft, float in, posneg sg=pos) : Distance(ft, in)    {
        sign = sg;
    }
    void getdist()    {
        Distance::getdist();
        char ch;
        cout << "Enter sign (+ or -): "; cin >> ch;
        sign = (ch=='+') ? pos : neg;
    }
    void showdist() const    {
        cout << ( (sign==pos) ? "(+)" : "(-)" );
        Distance::showdist();
    }
};

```

# Class Distance

```
int main()
{
    DistSign alpha;           //no-arg constructor
    alpha.getdist();

    DistSign beta(11, 6.25);
    DistSign gamma(100, 5.5, neg); //3-arg constructor

    cout << "\nalpha = "; alpha.showdist();
    cout << "\nbeta = "; beta.showdist();
    cout << "\ngamma = "; gamma.showdist();
    cout << endl;
    return 0;
}
```

# Constructors in DistSign

- Both constructors in DistSign call the corresponding constructors in Distance to set the feet-and-inches values.
  - `DistSign() : Distance()`
  - `DistSign(int ft, float in, posneg sg=pos) : Distance(ft, in)`
- They then set the sign variable. The no-argument constructor always sets it to pos.

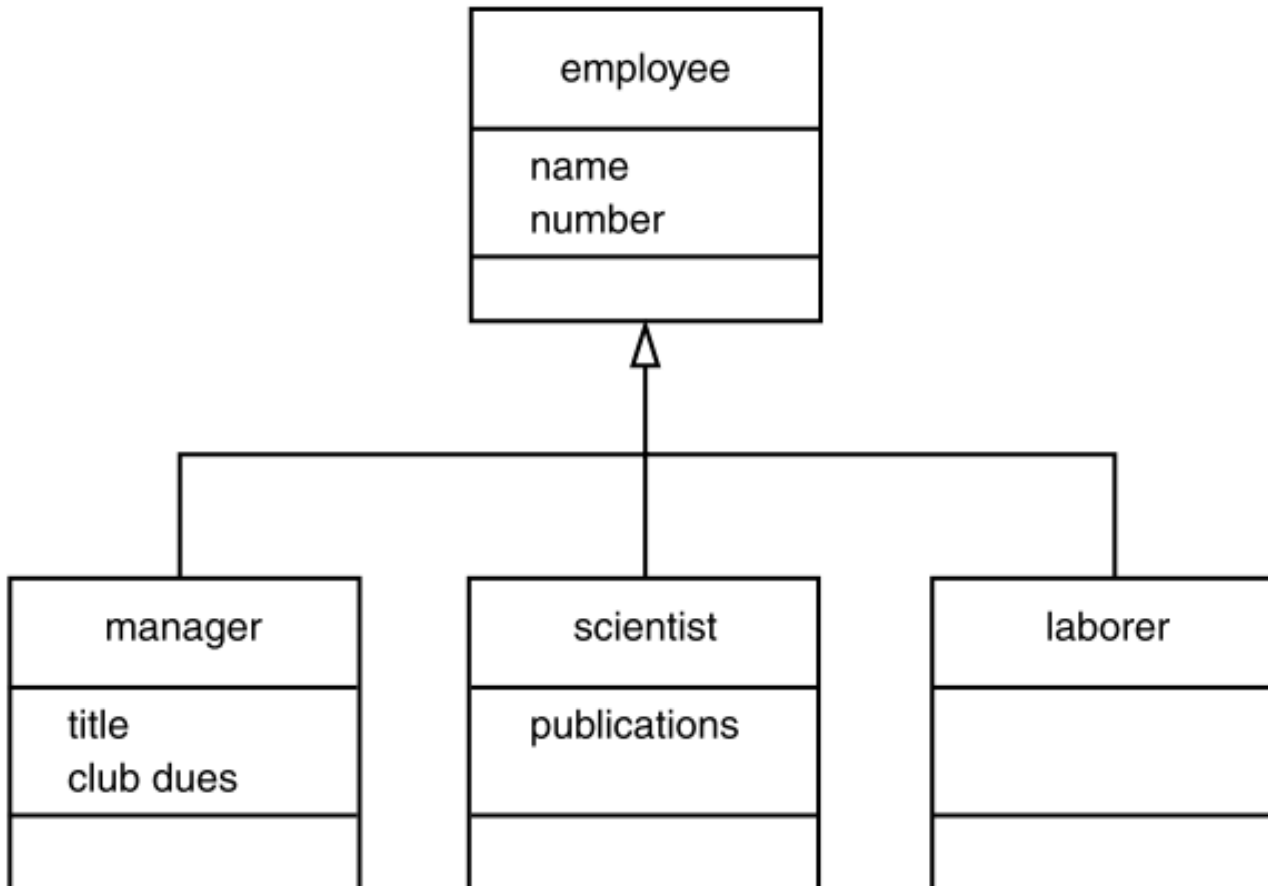
# Member Functions in DistSign

- The `getdist()` function in the `DistSign` class must ask the user for the sign as well as for feet-and-inches values, and the `showdist()` function must display the sign along with the feet and inches.
- These functions call the corresponding functions in `Distance`
  - `Distance::getdist();`
  - `Distance::showdist();`

# Class Hierarchies

- Consider an example to model a database of employees of a company.
- Three kinds of employees are represented.
  - **Managers** manage,
  - **Scientists** perform research to develop better devices,
  - **Laborers** operate the dangerous device stamping presses.
- The database stores a name and an employee identification number for all employees . In addition, it stores
  - Managers : store their titles and golf club dues.
  - Scientists : store the number of scholarly articles they have published.
  - Laborers : need no additional data.

# Class Hierarchies



```
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;
    }
};
```

```
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
{
}
```

```
int main() {  
    manager m1, m2;  
    scientist s1;  
    laborer l1;  
    cout << "\nEnter data for manager 1";  
    m1.getdata();  
    cout << "\nEnter data for manager 2";  
    m2.getdata();  
    cout << "\nEnter data for scientist 1";  
    s1.getdata();  
    cout << "\nEnter data for laborer 1";  
    l1.getdata();  
    cout << "\nData on manager 1";  
    m1.putdata();  
    cout << "\nData on manager 2";  
    m2.putdata();  
    cout << "\nData on scientist 1";  
    s1.putdata();  
    cout << "\nData on laborer 1";  
    l1.putdata();  
    return 0;  
}
```

# “Abstract” Base Class

- Notice that we **don't define any objects of the base class employee.**
- We use this as a general class whose sole purpose is to act as a base from which other classes are derived.
- The laborer class operates identically to the employee class, since it contains no additional data or functions.
- If in the future we decided to modify the laborer class, we would not need to change the declaration for employee.
- **Classes used only for deriving other classes, as employee, are sometimes loosely called abstract classes, meaning that no actual instances (objects) of this class are created.**

# Public & Private Inheritance

- **class manager : public employee**
  - Here, the keyword public specifies that objects of the derived class are able to access public member functions of the base class.
- **class manager : private employee**
  - When the private keyword is used, objects of the derived class cannot access public member functions of the base class.
  - Since objects can never access private or protected members of a class, the result is that no member of the base class is accessible to objects of the derived class.

```
class A {  
    private:  
        int privdataA;  
    protected:  
        int protdataA;  
    public:  
        int pubdataA;  
};
```

```
class B : public A {  
    public:  
        void funct() {  
            int a;  
            a = privdataA; //error: not accessible  
            a = protdataA; //OK  
            a = pubdataA; //OK  
        }  
};
```

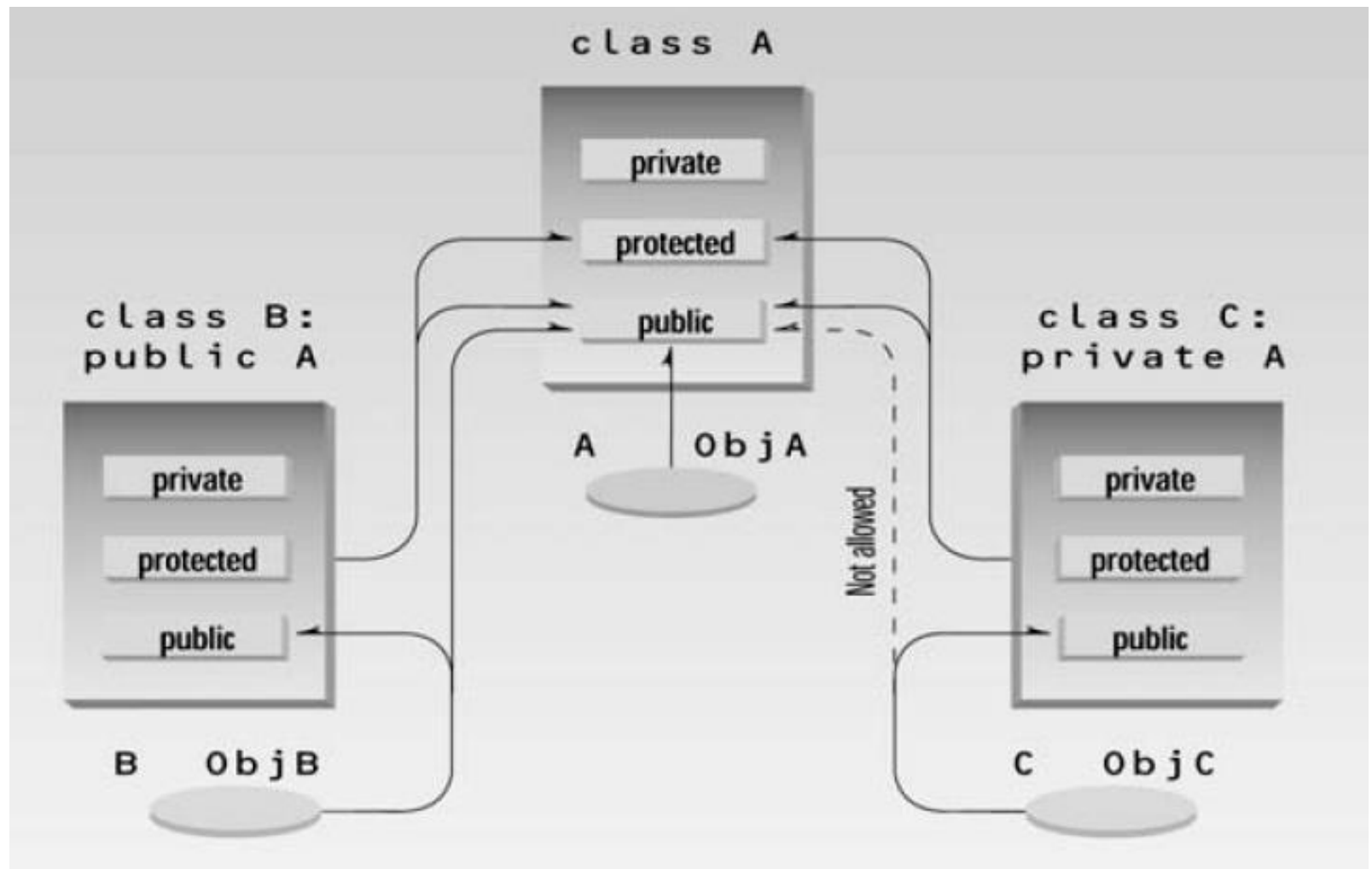
```
class C : private A {  
    public:  
        void funct() {  
            int a;  
            a = privdataA; //error  
            a = protdataA; //OK  
            a = pubdataA; //OK  
        }  
};
```

```
int main () {  
    int a;  
    B objB;  
    a = objB.privdataA; //error: not accessible  
    a = objB.protdataA; //error: not accessible  
    a = objB.pubdataA; //OK (A public to B)  
    C objC;  
    a = objC.privdataA; //error: not accessible  
    a = objC.protdataA; //error: not accessible  
    a = objC.pubdataA; //error: not accessible (A private to C)  
    return 0;  
}
```

# Public & Private Inheritance

- Functions in the derived classes can access protected and public data in the base class.
- Objects of the derived classes cannot access private or protected members of the base class.
- Objects of the publicly derived class B can access public members of the base class A, while objects of the privately derived class C cannot; they can only access the public members of their own derived class.

# Public & Private Inheritance



# Access Specifiers: When to Use What

- How do you decide when to use private as opposed to public inheritance?
- **Public**
  - In cases where a derived class exists to offer an improved (or specialized) version of the base class.
- **Private**
  - In cases where the derived class is created as a way of completely modifying the operation of the base class, hiding its original interface.

# Levels of Inheritance

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

```
class A
```

```
{ };
```

```
class B : public A
```

```
{ };
```

```
class C : public B
```

```
{ };
```

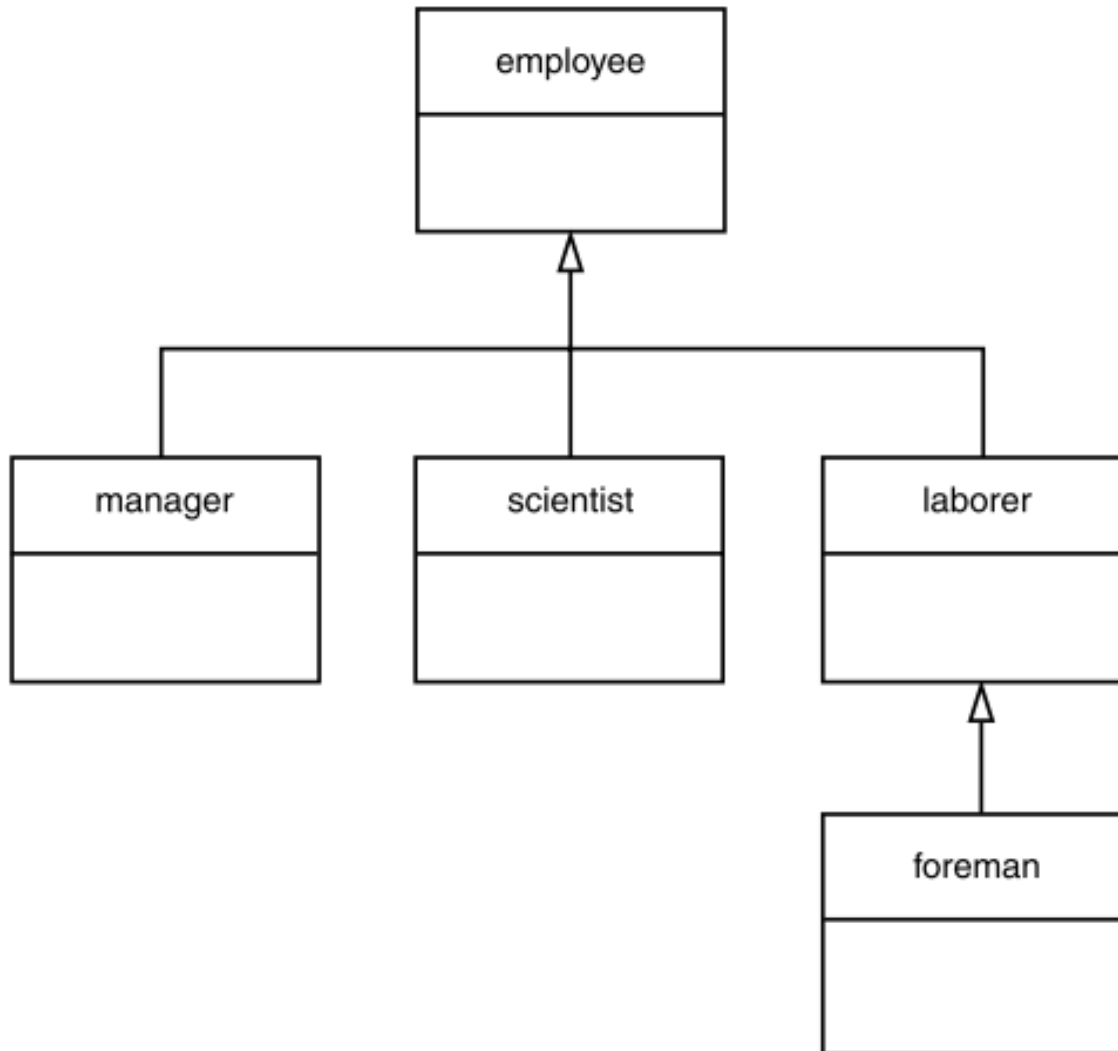
Here B is derived from A, and C is derived from B. The process can be extended to an arbitrary number of levels

- class base //single base class
- {
- public:
- int x;
- void getdata()
- {
- cout << "Enter value of x= "; cin >> x;
- }
- };
- class derive1 : public base // derived class from base class
- {
- public:
- int y;
- void readdata()
- {
- cout << "\nEnter value of y= "; cin >> y;
- }
- };
- class derive2 : public derive1 // derived from class derive1
- {
- private:
- int z;
- public:
- void indata()
- {
- cout << "\nEnter value of z= "; cin >> z;
- }
- void product()
- {
- cout << "\nProduct= " << x \* y \* z;
- }
- };

```
int main()
{
    derive2 a;    //object of derived
    a.getdata();
    a.readdata();
    a.indata();
    a.product();
    return 0;
}
```

## Multilevel Inheritance

# Multilevel Inheritance Example



```
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;
    }
};
```

```
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;
public:
    void getdata()    {
        laborer::getdata();
        cout << "    Enter quotas: "; cin >> quotas;
    }
    void putdata() const    {
        laborer::putdata();
        cout << "\n    Quotas: " << quotas;
    }
};
```

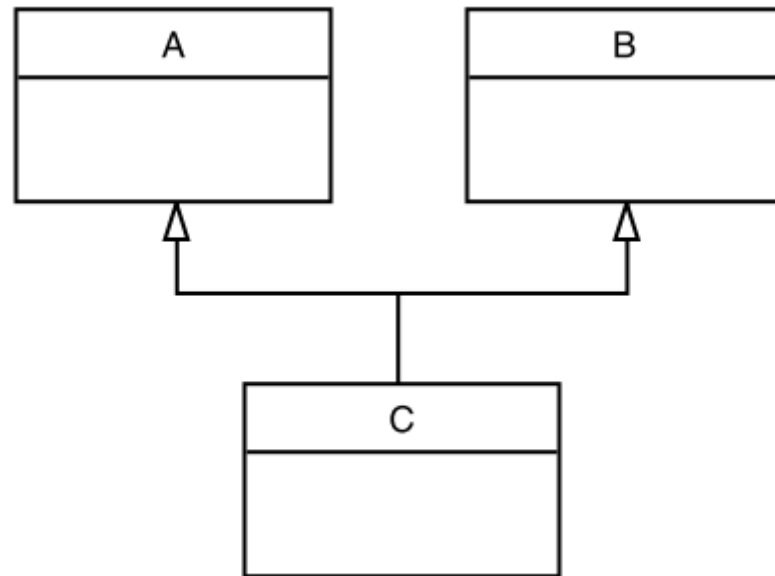
```
int main()    {
    laborer l1;
    foreman f1;

    cout << "\nEnter data for laborer 1";
    l1.getdata();
    cout << "\nEnter data for foreman 1";
    f1.getdata();

    cout << "\nData on laborer 1";
    l1.putdata();
    cout << "\nData on foreman 1";
    f1.putdata();
    cout << endl;
    return 0;
}
```

# Multiple Inheritance

- A class can be derived from more than one base class. This is called multiple inheritance.
- class C is derived from base classes A and B



# Multiple Inheritance

```
class A                // base class A
{
};

class B                // base class B
{
};

class C : public A, public B    // C is derived from A and B
{
};
```

# Constructor in Multiple inheritance

```
class B1
{   public:
    B1()
    {   cout<<"default constructor in class B1"; }
};
class B2
{   public:
    B2()
    {   cout<<"default constructor in class B2"; }
};
class D: public B1, public B2
{   public:
    D()
    {   cout<<"default constructor in class D"; }
};
int main()
{
    D objd;
}
```

**OP: default constructor in class B1**  
**default constructor in class B2**  
**default constructor in class D**

# Constructor in Multiple inheritance

```
class B1
{   public:
    B1()
    {   cout<<"default constructor in class B1"; }
};
class B2
{   public:
    B2()
    {   cout<<"default constructor in class B2"; }
};
class D: public B1, public B2
{   public:
    D() : B2() , B1()
    {   cout<<"default constructor in class D"; }
};
int main()
{
    D objd;
}
```

**OP: default constructor in class B1**  
**default constructor in class B2**  
**default constructor in class D**

# Constructor in Multilevel inheritance

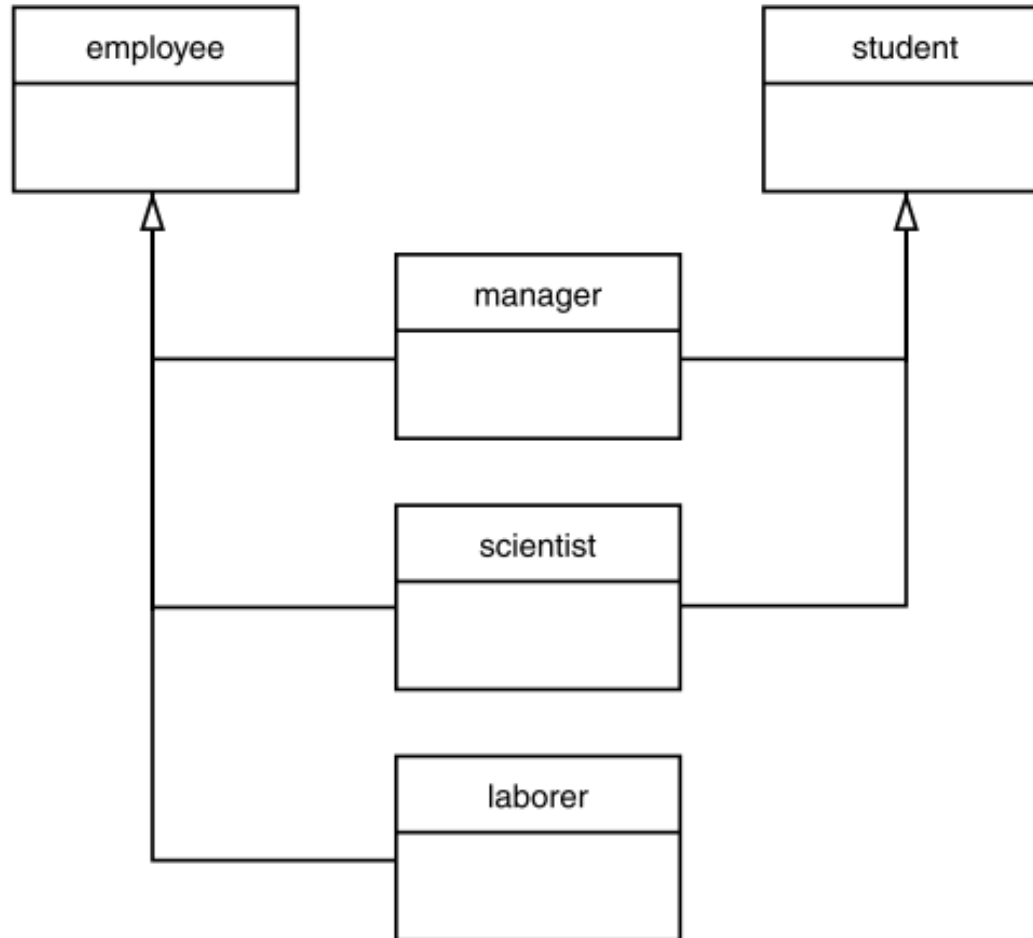
```
class B1
{   public:
    B1()
    {   cout<<"default constructor in class B1"; }
};
class B2 : public B1
{   public:
    B2()
    {   cout<<"default constructor in class B2"; }
};
class D: public B2
{   public:
    D()
    {   cout<<"default constructor in class D"; }
};
int main()
{
    D objd;
}
```

**OP: default constructor in class B1**  
**default constructor in class B2**  
**default constructor in class D**

# Multiple Inheritance Example

- We need to record the educational experience of the employees.
- We have already developed a class called student that models students with different educational backgrounds.
- We decide that instead of modifying the employee class to incorporate educational data, we will add this data by multiple inheritance from the student class.

# Multiple Inheritance Example



# Multiple Inheritance Example

```
class student
{ };
class employee
{ };
class manager : private employee, private student
{ };
class scientist : private employee, private student
{ };
class laborer : public employee
{ };
```

```
class student          { //educational background
private:
    char school[LEN];
    char degree[LEN];
public:
    void getedu()      {
        cout << "  Enter name of school or university: ";
        cin >> school;
        cout << "  Enter highest degree earned \n";
        cout << "  (Highschool, Bachelor's, Master's, PhD): ";
        cin >> degree;
    }
    void putedu() const {
        cout << "\n  School or university: " << school;
        cout << "\n  Highest degree earned: " << degree;
    }
};
```

```
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 : private employee, private student    {
private:
    char title[LEN];
    double dues;
public:
    void getdata() {
        employee::getdata();
        cout << "  Enter title: ";      cin >> title;
        cout << "  Enter golf club dues: "; cin >> dues;
        student::getedu();
    }
    void putdata() const    {
        employee::putdata();
        cout << "\n  Title: " << title;
        cout << "\n  Golf club dues: " << dues;
        student::putedu();
    }
};
```

```
class scientist : private employee, private student {
private:
    int pubs;
public:
    void getdata() {
        employee::getdata();
        cout << "  Enter number of pubs: "; cin >> pubs;
        student::getedu();
    }
    void putdata() const {
        employee::putdata();
        cout << "\n  Number of publications: " << pubs;
        student::putedu();
    }
};
```

```
class laborer : public employee  
{  
};
```

```
int main()    {
    manager m1;
    scientist s1;
    laborer l1;
    cout << "\nEnter data for manager 1";
    m1.getdata();
    cout << "\nEnter data for scientist 1";
    s1.getdata();
    cout << "\nEnter data for laborer 1";
    l1.getdata();
    cout << "\nData on manager 1";
    m1.putdata();
    cout << "\nData on scientist 1";
    s1.putdata();
    cout << "\nData on laborer 1";
    l1.putdata();
    return 0;
}
```

# Multiple Inheritance Example

- The **manager and scientist classes are privately derived from the employee and student classes.**
  - There is no need to use public derivation because objects of manager and scientist never call routines in the employee and student base classes.
- However, **the laborer class must be publicly derived from employer, since it has no member functions of its own and relies on those in employee.**

# Constructors in Multiple inheritance

```
class Type
{
private:
string dimensions;
string grade;
public:
Type() : dimensions("N/A"), grade("N/A")
{ }
Type(string di, string gr) : dimensions(di), grade(gr)
{ }
void gettype()
{
cout << " Enter nominal dimensions (2x4 etc.): ";
cin >> dimensions;
cout << " Enter grade (rough, const, etc.): ";
cin >> grade;
}
void showtype() const
{
cout << "\n Dimensions: " << dimensions;
cout << "\n Grade: " << grade;
}
};
```

```
class Distance
{
private:
int feet;
float inches;
public:
Distance() : feet(0), inches(0.0)
{ }
Distance(int ft, float in) : feet(ft), inches(in)
{ }
void getdist()
{
cout << " Enter feet: "; cin >> feet;
cout << " Enter inches: "; cin >> inches;
}
void showdist() const
{ cout << feet << "\'-" << inches << '\''; }
};
```

```

class Lumber : public Type, public Distance
{
private:
int quantity;
double price;
public:
Lumber() : Type(), Distance(), quantity(0), price(0.0)
{ }
Lumber( string di, string gr, int ft, float in,int qu, float prc ) : Type(di, gr),Distance(ft, in),
quantity(qu), price(prc)
{ }
void getlumber()
{
Type::gettype();
Distance::getdist();
cout << " Enter quantity: "; cin >> quantity;
cout << " Enter price per piece: "; cin >> price;
}
void showlumber() const
{
Type::showtype();
cout << "\n Length: ";
Distance::showdist();
cout << "\n Price for " << quantity
<< " pieces: $" << price * quantity;
}
};

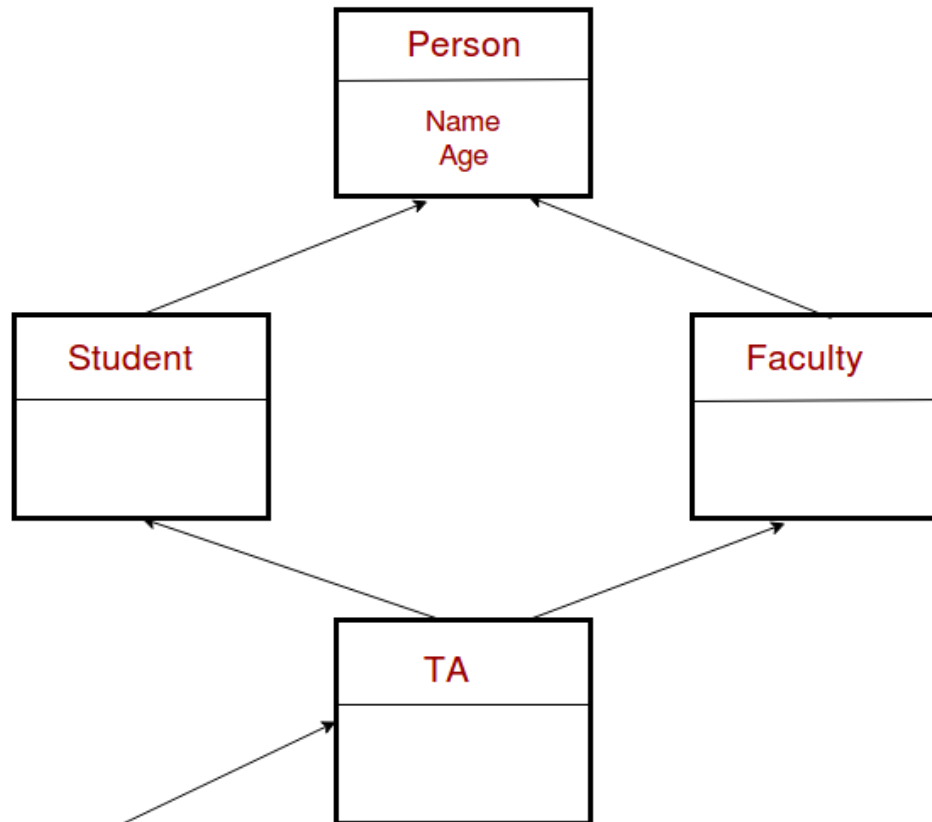
```

```
int main()
{
    Lumber siding;
    cout << "\nSiding data:\n";
    siding.getlumber();
    Lumber studs( "2x4", "const", 8, 0.0, 200, 4.45F );
    cout << "\nSiding"; siding.showlumber();
    cout << "\nStuds"; studs.showlumber();
    cout << endl;
    return 0;
}
```

# Ambiguity in Multiple inheritance

```
class A
{
public:
void show() { cout << "Class A\n"; }
};
class B
{
public:
void show() { cout << "Class B\n"; }
};
class C : public A, public B
{
};
int main()
{
C objC;
// objC.show();           //ambiguous--will not compile
objC.A::show();           //OK
objC.B::show();           //OK
return 0;
}
```

# Diamond Problem



Name and Age needed only once

# Contd..

- `#include<iostream>`
- `using namespace std;`
- `class Person {`
- `public:`
- `Person(int x) { cout << "Person::Person(int ) called" << endl; }`
- `};`
- `class Faculty : public Person {`
- `public:`
- `Faculty(int x):Person(x) {`
- `cout<<"Faculty::Faculty(int ) called"<< endl;`
- `}`
- `};`
- `class Student : public Person {`
- `public:`
- `Student(int x):Person(x) {`
- `cout<<"Student::Student(int ) called"<< endl;`
- `}`
- `};`
- `class TA : public Faculty, public Student {`
- `public:`
- `TA(int x):Student(x), Faculty(x) {`
- `cout<<"TA::TA(int ) called"<< endl;`
- `}`
- `};`
- `int main() {`
- `TA ta1(30);`
- `}`

- Output:

Person::Person(int ) called

Faculty::Faculty(int ) called

Person::Person(int ) called

Student::Student(int ) called

TA::TA(int ) called

# Contd..

- class Person {
- public:
- Person(int x) { cout << "Person::Person(int ) called" << endl; }
- Person()     { cout << "Person::Person() called" << endl; }
- };
- class Faculty : virtual public Person {
- public:
- Faculty(int x):Person(x) {
- cout<<"Faculty::Faculty(int ) called"<< endl;
- }
- };
- class Student : virtual public Person {
- public:
- Student(int x):Person(x) {
- cout<<"Student::Student(int ) called"<< endl;
- }
- };
- class TA : public Faculty, public Student {
- public:
- TA(int x):Student(x), Faculty(x) {
- cout<<"TA::TA(int ) called"<< endl;
- }
- };
- int main() {
- TA ta1(30);
- }

- Output:

Person::Person() called

Faculty::Faculty(int ) called

Student::Student(int ) called

TA::TA(int ) called

# How to call the parameterized constructor of the 'Person' class?

```
class Person {
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl; }
    Person() { cout << "Person::Person() called" << endl; }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int ) called"<< endl;
    }
};

class Student : virtual public Person {
public:
    Student(int x):Person(x) {
        cout<<"Student::Student(int ) called"<< endl;
    }
};

class TA : public Faculty, public Student {
public:
    TA(int x):Student(x), Faculty(x), Person(x) {
        cout<<"TA::TA(int ) called"<< endl;
    }
};

int main() {
    TA ta1(30);
}
```

- Output:

Person::Person(int ) called

Faculty::Faculty(int ) called

Student::Student(int ) called

TA::TA(int ) called

# Object Composition or Containership

- If a class B is derived by inheritance from a class A, we can say that “B is a kind of A.” This is because B has all the characteristics of A, and in addition some of its own. So inheritance is often called a “kind of” relationship.
- Containership is called a “has a” relationship. We say a library has a book or an invoice has an item line. It is also called a “part-whole” relationship: the book is part of the library

- Aggregation may occur when one object is an attribute of another. Here's a case where an object of class A is an attribute of class B:

```
class A { };
```

```
class B
```

```
{
```

```
    A objA;    //define objA as an object of class A
```

```
};
```

# Example

```
class student
{};
class employee
{};
class manager
{
student stu;           // stu is an object of class student
employee emp;          // emp is an object of class employee
};
class scientist
{
student stu;           // stu is an object of class student
employee emp;          // emp is an object of class employee
};
class laborer
{
employee emp;          // emp is an object of class employee
};
```

```
class student          { //educational background
private:
    char school[LEN];
    char degree[LEN];
public:
    void getedu()      {
        cout << "  Enter name of school or university: ";
        cin >> school;
        cout << "  Enter highest degree earned \n";
        cout << "  (Highschool, Bachelor's, Master's, PhD): ";
        cin >> degree;
    }
    void putedu() const {
        cout << "\n  School or university: " << school;
        cout << "\n  Highest degree earned: " << degree;
    }
};
```

```
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 {  
private:  
    char title[LEN];  
    double dues;  
    employee e1;  
    student s1;  
public:  
    void getdata() {  
        e1.getdata();  
        cout << "  Enter title: ";    cin >> title;  
        cout << "  Enter golf club dues: "; cin >> dues;  
        s1.getedu();  
    }  
    void putdata() const    {  
        e1.putdata();  
        cout << "\n  Title: " << title;  
        cout << "\n  Golf club dues: " << dues;  
        s1.putedu();  
    }  
};
```

```
class scientist {
private:
    int pubs;
    employee e1;
    student s1;
public:
    void getdata() {
        e1.getdata();
        cout << " Enter number of pubs: "; cin >> pubs;
        s1.getedu();
    }
    void putdata() const {
        e1.putdata();
        cout << "\n Number of publications: " << pubs;
        s1.putedu();
    }
};
```

```
class laborer
{
    private:
        employee emp;           //object of class employee
    public:
        void getdata()
        {
            emp.getdata();
        }
        void putdata() const
        {
            emp.putdata();
        }
};
```

```
int main()      {
    manager m1;
    scientist s1;
    laborer l1;
    cout << "\nEnter data for manager 1";
    m1.getdata();
    cout << "\nEnter data for scientist 1";
    s1.getdata();
    cout << "\nEnter data for laborer 1";
    l1.getdata();
    cout << "\nData on manager 1";
    m1.putdata();
    cout << "\nData on scientist 1";
    s1.putdata();
    cout << "\nData on laborer 1";
    l1.putdata();
    return 0;
}
```

