CS217 OBJECT ORIENTED PROGRAMMING

Spring 2020

TYPE CONVERSION

- The process of converting a value from one data type to another is called a **type conversion.** Type conversions can happen in many different cases:

```
double d{ 3 };// initialize double variable with integer value 3
      d = 6;// assign double variable the integer value 6
```

OR

```
void doSomething(long l){ }
doSomething(3); // pass integer value 3 to a function expecting a lo
ng parameter
```

OR

```
double division{4.0/3}; // division with a double and an integer
```

IMPLICIT TYPE CONVERSION (COERCION)

- **Implicit type conversion** (also called **automatic type conversion** or **coercion**) is performed whenever one data type is expected, but a different data type is supplied. If the compiler can figure out how to do the conversion between the two types, it will.
- All of the above examples are cases where implicit type conversion will be used.
- Whenever a value from one fundamental data type is converted into a value of a larger fundamental data type from the same family, this is called a **numeric promotion** (or **widening**, though this term is usually reserved for integers).

```
long l{64 }; // widen the integer 64 into a long
double d{0.12f }; // promote the float 0.12 into a double
```

- When we convert a value from a larger type to a similar smaller type, or between different types, this is called a **numeric conversion.** Unlike promotions, which are always safe, conversions may or may not result in a loss of data

```
double d{ 3 }; // convert integer 3 to a double (between different types)
short s{ 2 }; // convert integer 2 to a short (from larger to smaller type within same ty
pe family)
```

EXPLICIT TYPE CONVERSION (CASTING)

- When you want to promote a value from one data type to a larger similar data type, using implicit type conversion is fine.
- In C++, there are 5 different types of casts:

1. **C-style casts**
2. **static casts**
3. **const casts**
4. **dynamic casts**
5. and **reinterpret casts**.

- The latter four are sometimes referred to as **named casts**.

C-STYLE CASTS

```
int i1 = 10;
int i2 = 4;
float f = (float) i1 / i2; //Casting
```

- C++ will also let you use a C-style cast with a more function-call like syntax:

```
int i1 = 10;
int i2 = 4;
float f = float(i1) / i2;
```

STATIC_CAST

- C++ introduces a casting operator called **static_cast**, which can be used to convert a value of one type to a value of another type.

```
char ch = ‘a’ ;

std::cout << ch << ' ' << static_cast<int>(ch) << '\n'; // prints a 97
```

- The **static_cast** operator takes a single value as input, and outputs the same value converted to the type specified inside the angled brackets.
- Static_cast is best used to convert one fundamental type into another.

```
int i1 { 10 };
int i2 { 4 };

// convert an int to a float so we get floating point division rather than integer division
float f { static_cast<float>(i1) / i2 };
```

- Compilers will often complain when an *unsafe implicit type conversion* is performed. For example, consider the following program:

```
int i { 48 };
char ch = i; // implicit conversion
```

- Casting an int (4 bytes) to a char (1 byte) is potentially unsafe (as the compiler can't tell whether the integer will overflow the range of the char or not), and so the compiler will typically complain.
- To get around this, we can use a static cast to explicitly convert our integer to a char:

```
int i { 48 };
// explicit conversion from int to char, so that a char is assigned to variable
char ch = static_cast<char>(i);
```

OBJECT RELATIONSHIP

- is-a relationship
- has-a relationship:

HAS-A RELATIONSHIP

- In a *has-a* relationship, an object contains one or more objects of other classes as members

A car ***has a*** steering
An office ***has a*** department

DEPARTMENT HAS An Office

HAS-A RELATIONSHIP

```
class Office
{
  Department d;
  // any other  members
};
```

IS-A RELATIONSHIP

- Sometimes, one class ***is*** an extension of another class

A car ***is a*** vehicle
Cricket ***is a*** sport

IS-A RELATIONSHIP

- The extended (or child) class contains all the features of its base (or parent) class, and may additionally have some unique features of its own

- The key idea behind inheritance

INHERITANCE

- Introduction
- Derived class and base class
- Derived class constructors, OVERRIDING member functions
- Class Hierarchies, Public and Private inheritance
- Levels of inheritance, Multiple inheritance

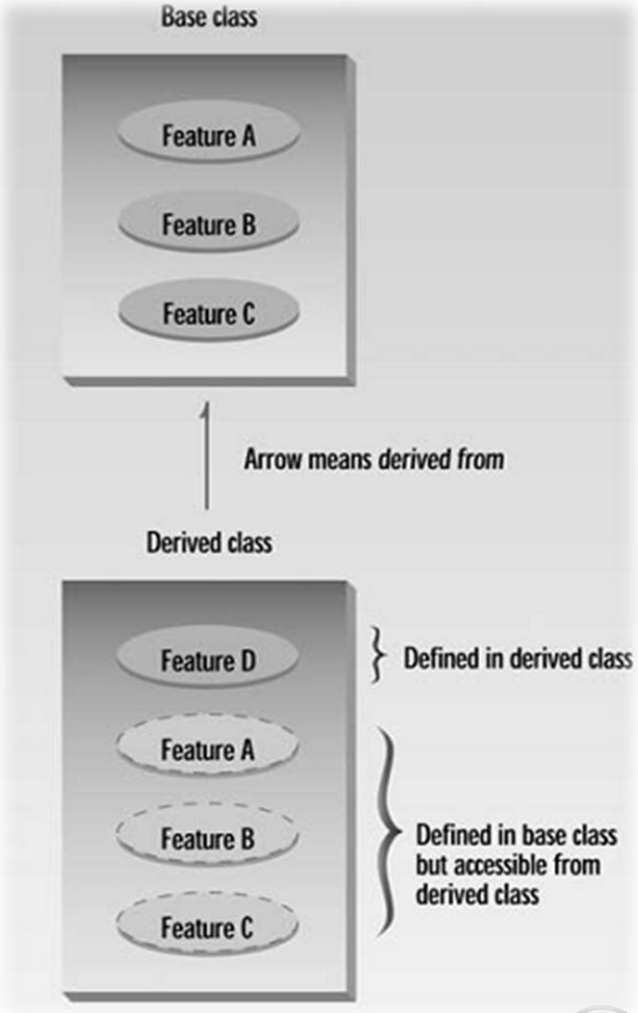
INTRODUCTION

•**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 features of its own.

- •The base class is unchanged by this process

•Inheritance permits *code reusability*: Once a base class is written and debugged, it need not be touched again.



Derived Class and the Base Class

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


/////////////// Base class//////////////
class Shape {
  public:
       void setWidth(int w)
       { width = w; }

       void setHeight(int h)
       { height = h; }

  protected:
       int width; int height; };
```

```
/////////// Derived class ///////////
class Rectangle: public Shape {
 public:
       int getArea()
       { return (width * height); }
protected:
       int a;
 };

int main() {
       Rectangle Rect;
       Rect.a;
       Rect.setWidth(5);
       Rect.setHeight(7);
       Shape s1;
       s1.getarea();
       cout<<"Total area: "<<Rect.getArea()
<<endl;

return 0;
```

- **Following the `Shape` class, there is the specification for a new class, `Rectangle`.**
  - **This class incorporates a new function, `getArea()`, which returns the area .**

- **The new `Rectangle` class inherits all the features of the `Shape` class.**
  - **`Rectangle` class does not need any `setHeight()` or `setWidth()` functions, because these already exist in `Shape` class.**

- **The first line of `Rectangle` specifies that it is derived from `Shape`:**

```
class Rectangle: public Shape
```

  - **This sets up the relationship between the classes. This line says that *`Rectangle` is derived from the base class `Shape`*.**

- When a member function in the base class can be used by objects of the derived class, it is called *accessibility*.
- In the `main()`, we create an object of class `Rectangle`:

```
Rectangle Rect;
```

- This causes Rect to be created as an object of class Rectangle.
- Later on we called two member function through Rect:

```
Rect.setWidth(5);
Rect.setHeight(7);
```

- But how is this possible? There is no `setWidth` or `setHeight` functions in the `Rectangle` class specifier.
  - If you don't specify a function, the derived class will use an appropriate function from the base class.
- In `Rectangle` there's no `setWidth` or `setHeight`, so the compiler uses these functions from `Shape`.

- A member function of a class can always access class members, whether they are `public` or `private`. But an object declared externally can only access the `public` members of the class.
  - It's not allowed to use `private` members.

- **Can member functions of the derived class access members of the base class?**

  Member functions can access members of the base class if the members are `public`. They can't access `private` members.

- `protected`: if the members are `protected`, they can be accessed by member function of derived classes, but not by other functions.

TABLE 9.1 Inheritance and Accessibility

Access Specifier	*Accessible from Own Class*	*Accessible from Derived Class*	*Accessible from Objects Outside Class*
public	yes	yes	yes
protected	yes	yes	no
private	yes	no	no

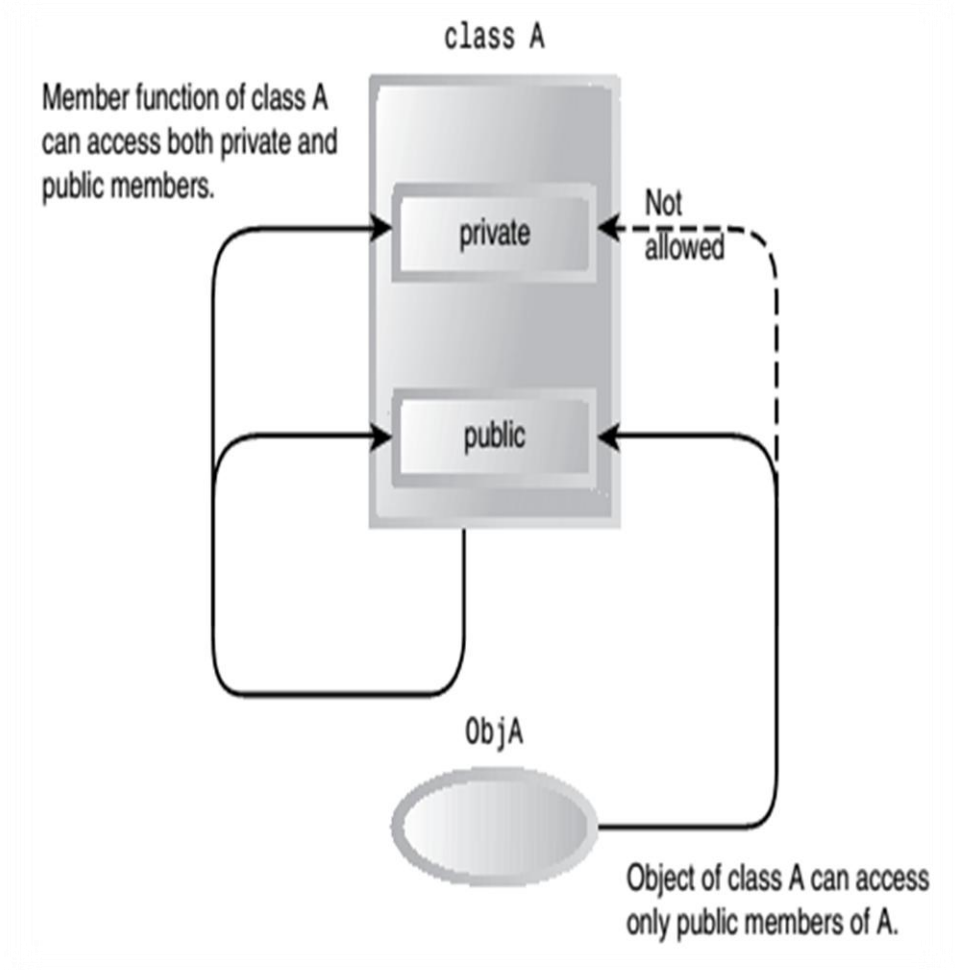
Remember, even if other classes have been derived from it, the base class remains unchanged.

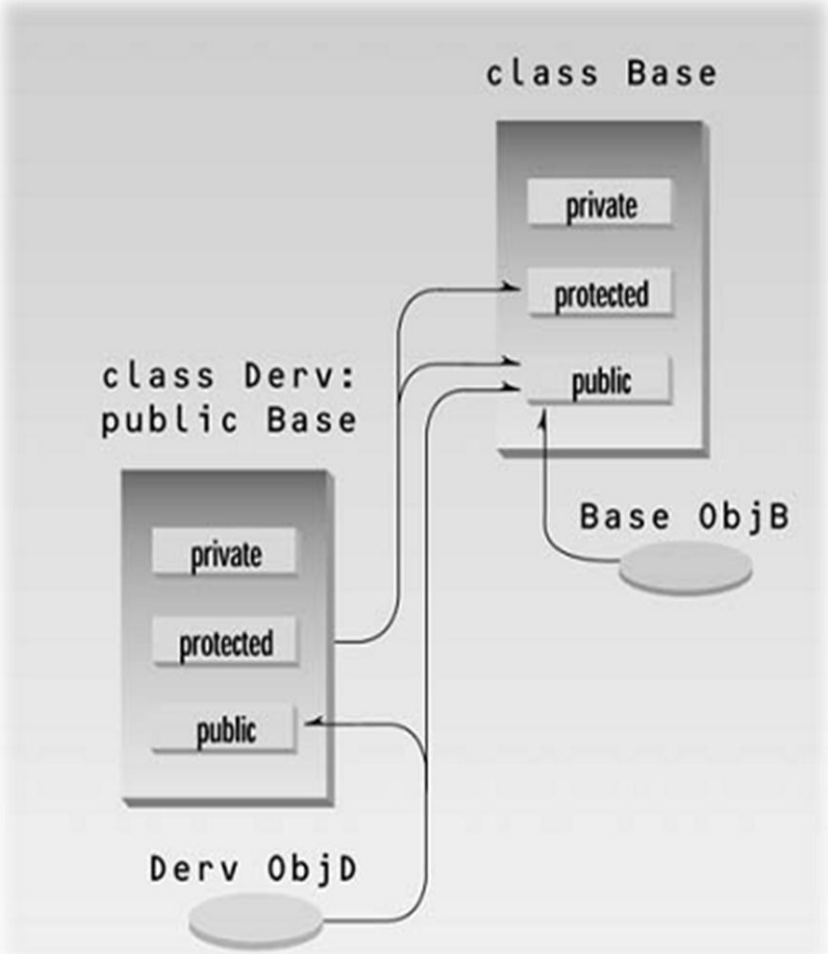
- In the `main()` part of the program, we could define objects of type `Shape`:

```
Shape shp;
```

  - Such objects would behave just as they would if `Rectangle` didn't exist.

- In some languages the base class is called the *super class* and the derived class is called the *subclass*. Some writers also refer to the base class as the *parent* and the derived class as the *child.*


class A
Member function of class A can access both private and public members.
private
Not allowed
public
ObjA
Object of class A can access only public members of A.


class Base
private
protected
public
class Derv: public Base
Base ObjB
private
protected
public
Derv ObjD

Derived Class Constructors

```
class Counter{
     protected:
          int count;

     public:
          Counter() : count(10)
          { }

          Counter(int c) : count(c)
          { }

          int get_count()
                    { return count; }
               }; // end of class Counter

class CountDn : public Counter{

     public:
                    CountDn() : Counter()
                    { }

                    CountDn(int c) : Counter(c)
                    { }
     }; // end of class CountDn

int main()
{
     CountDn c1;
     CountDn c2(100);

     cout << "\nc1=" << c1.get_count();
     cout << "\nc2=" << c2.get_count();
return 0; }
```

- The program uses two constructors in the `Counter` class (the base class):
  1. `Counter()`, that is without arguments
  2. `Counter(int c)` , that is with one argument

- The `CountDn` (the derived class) is also having two constructors:
  1. `CountDn()`, that is without arguments
  2. `CountDn(int c)` , that is with one argument.

- The following constructor has an unfamiliar feature: the function name following the colon

```
CountDn() : Counter()
{ }
```

  - This construction causes the `CountDn()` constructor to call the `Counter()` constructor in the base class.

- In `main()`, when we say

```
CountDn c1;
```

  - the compiler will create an object of type `CountDn` and then call the `CountDn` constructor.
  - The `CountDn` constructor in turn call the `Counter` constructor (of base class), which carries out the work, that is, initialize the variable `count`.

- Calling a constructor from the initialization list may seem odd, but it makes sense.
  - You 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.

- **The statement**

  `CountDn c2(100);`

  in `main()` **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)` ← **argument c is passed to** `Counter`

  - **This construction causes the argument** `c` **to be passed from** `CountDn()` **to** `Counter(int c)`**, where it is used to initialize the object.**

- You can use member functions in a derived class that **override**—that is, have the same name as—those in the base class.
  - You might want to do this so that calls in your program work the same way for objects of both base and derived classes.

- When the same function exists in both the base class and the derived class, the function in the derived class will be executed.

- We say that the ***derived class function overrides the base class function.***

- Classes used only for deriving other classes, as `counter` in our example, are sometimes called **abstract classes**, meaning that no actual instances (objects) of this class are created.

PUBLIC AND PRIVATE INHERITANCE

- Our examples so far have used publicly derived classes, with declarations like:

```
class Rectangle: public Shape
```

- The keyword `public` specifies that objects of the derived class are able to access public member functions of the base class.
- When the key word `private` is used, objects of the derived class cannot access public member functions of the base class.
- The result is that no member of the base class is accessible to objects of the derived class.
  - Since objects can never access `private` or `protected` members of a class.

```
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////////
class A                  //base class
   {
   private:
      int privdataA;     //(functions have the same access
   protected:            //rules as the data shown here)
      int protdataA;
   public:
      int pubdataA;
   };
////////////////////////////////////////////////////////////////
class B : public A       //publicly-derived class
   {
   public:
      void funct()
         {
         int a;
         a = privdataA;  //error: not accessible
         a = protdataA;  //OK
         a = pubdataA;   //OK
         }
   };
////////////////////////////////////////////////////////////////
class C : private A      //privately-derived class
   {
   public:
      void funct()
         {
         int a;
         a = privdataA;  //error: not accessible
         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;
}
```

- The program specifies a base class, **A** ,with `private`, `protected`, and `public` data items.

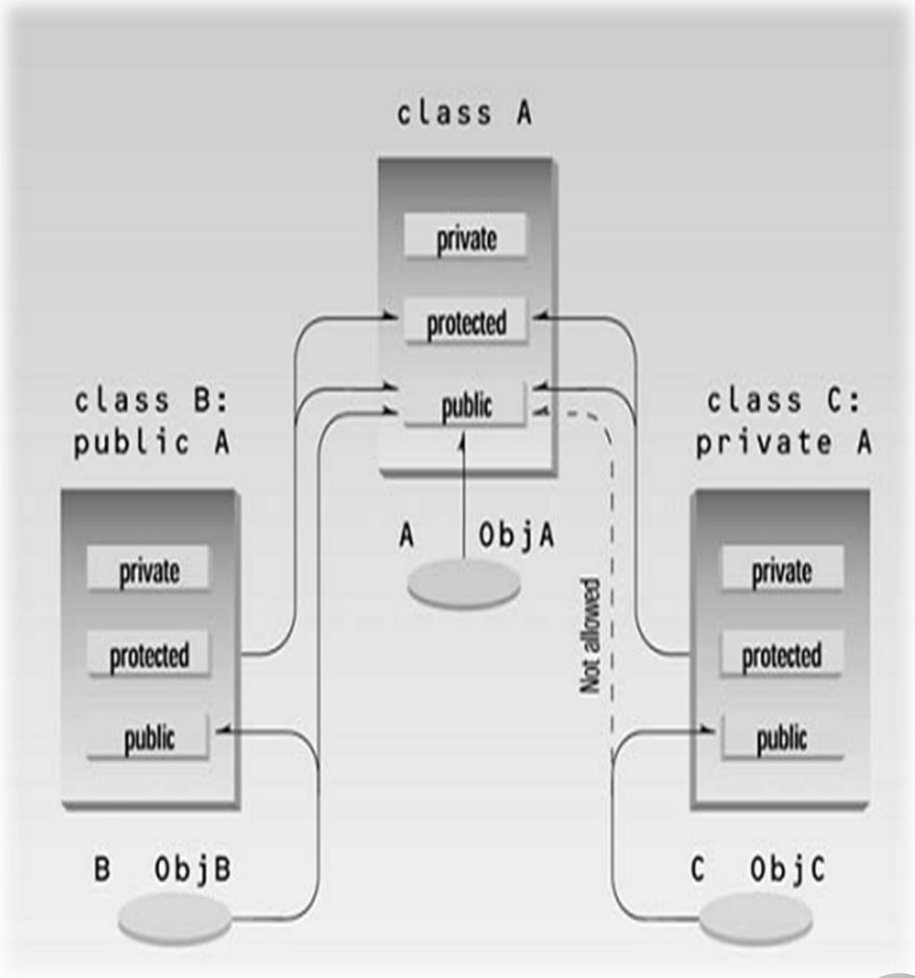
- Two classes, **B** and **C**, are derived from **A**.
  - **B** is publicly derived.
  - **C** is privately derived.

- Functions in the derived classes can access protected and public data in the base class, Objects of the classes (either of base class or derived class) cannot access `private` or `protected` members of the base class.

- If you don't supply any access specifier (`public` or `private`) when creating a class, `private` is assumed.

Access in Base Class	Base Class Inherited as	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

- 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.



LEVELS OF INHERITANCE

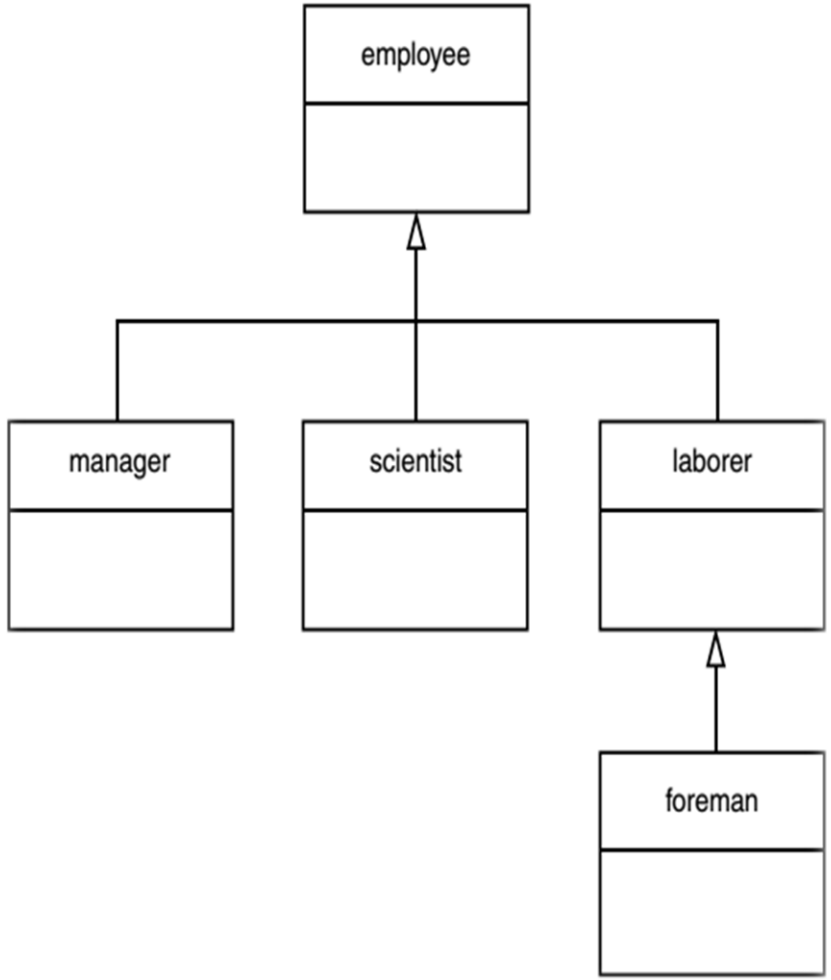
```
class employee
{ };

class manager: public employee
{ };

class scientist: public employee
{ };

class laborer : public employee
{ };

class foreman : public laborer
{ };
```



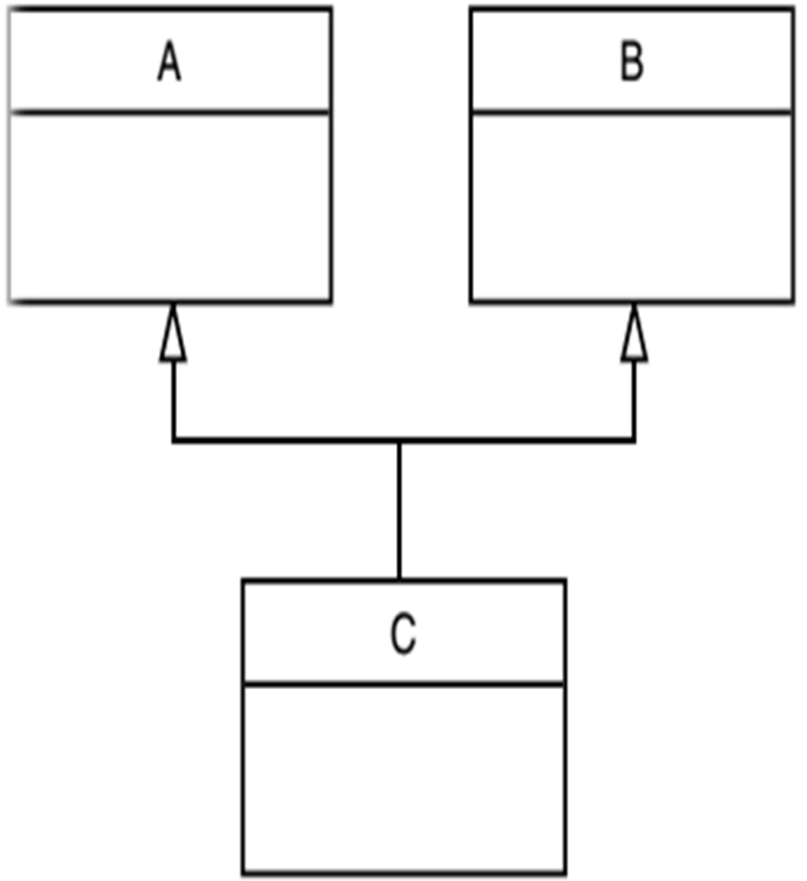
MULTIPLE INHERITANCE

- A class can be derived from more than one base class.
- This is called *multiple inheritance*.
- The syntax for multiple inheritance is similar to that for single inheritance

```
class A
{};

class B
{};

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



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

class Shape {
public:
void setWidth(int w)
{ width = w; }

void setHeight(int h)
{height = h; }

protected:
int width; int height;
};

class PaintCost {
public:
int getCost(int area)
{ return area * 70; }

};
```

```
class Rectangle:public Shape, public
PaintCost{
public:
int getArea()
{ return (width * height); }
};

int main() {
Rectangle Rect;
int area;

Rect.setWidth(5);
Rect.setHeight(7);

area = Rect.getArea()
cout<<"Total area:"<< Rect.getArea() <<
endl;
cout << "Total paint cost: $" <<
Rect.getCost(area) << endl;
return 0;
}
```

- Ambiguity in Multiple Inheritance
  - What if two base classes have functions with the same name, while a class derived from both base classes has no function with this name?
    - How do objects of the derived class access the correct base class function?
  - The name of the function alone is insufficient, since the compiler can't figure out which of the two functions is meant.

```
// ambigu.cpp
// demonstrates ambiguity in multiple inheritance
#include <iostream>
using namespace std;
////////////////////////////////////////////////////////////
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;             //object of class C
// objC.show();        //ambiguous--will not compile
   objC.A::show();     //OK
   objC.B::show();     //OK
   return 0;
   }
```

- The problem is resolved using the scope-resolution operator **::** to specify the class in which the function lies.
  - Thus `objC.A::show();` refers to the version of `show()` that's in the `A` class
  - While `objC.B::show();` refers to the function in the `B` class.

- Another kind of ambiguity arises if you derive a class from two classes that are each derived from the same class.
  - This creates a diamond-shaped inheritance tree.

```
#include <iostream>
using namespace std;
////////////////////////////////////////////
class A
   {
   public:
      void func();
   };
class B : public A
   {  };
class C : public A
   {  };
class D : public B, public C
   {  };
////////////////////////////////////////////
int main()
   {
   D objD;
   objD.func();  //ambiguous: won't compile
   return 0;
   }
```

- Classes **B** and **C** are both derived from class **A**, and class **D** is derived by multiple inheritance from both **B** and **C**.

- Trouble starts if you try to access a member function in class **A** from an object of class D.
  - In this example `objD` tries to access `func()`. However, both B and C contain a copy of `func()`, inherited from **A**. The compiler can't decide which copy to use, and signals an error.

- There are various advanced ways of coping with this problem, but you should certainly not use it in serious programs unless you have considerable experience

SUMMARY

- A class, called the derived class, can inherit the features of another class, called the base class.
- The derived class can add other features of its own, so it becomes a specialized version of the base class.
- Inheritance provides a powerful way to extend the capabilities of existing classes, and to design programs using hierarchical relationships.
- Accessibility of base class members from derived classes and from objects of derived classes is an important issue.

SUMMARY

- Data or functions in the base class that are prefaced by the keyword `protected` can be accessed from derived classes but not by any other objects, including objects of derived classes.

- Classes may be publicly or privately derived from base classes. Objects of a publicly derived class can access `public` members of the base class, while objects of a privately derived class cannot.

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

- Inheritance permits the reusability of software: Derived classes can extend the capabilities of base classes with no need to modify—or even access the source code of—the base class