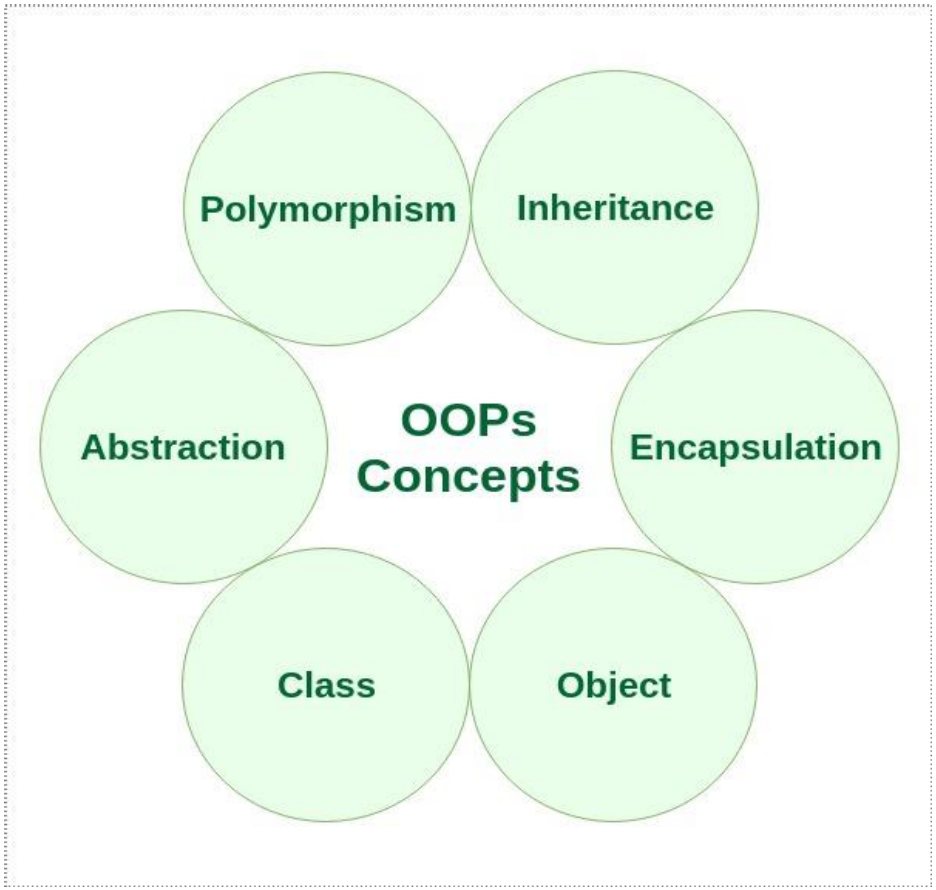
CSE 204
Object Oriented Programing:
Inheritance

Contents

- Overview
  - Inheritance
  - Base & Derived Classes
  - Direct & Indirect Base and Derived Classes
  - Derived Class as public
  - Data Members & Member Functions accessibility in derived class when declared as public, protected and private.

OOP Concepts



OOP Concepts

Class is the building block of C++ that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

An **Object** is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

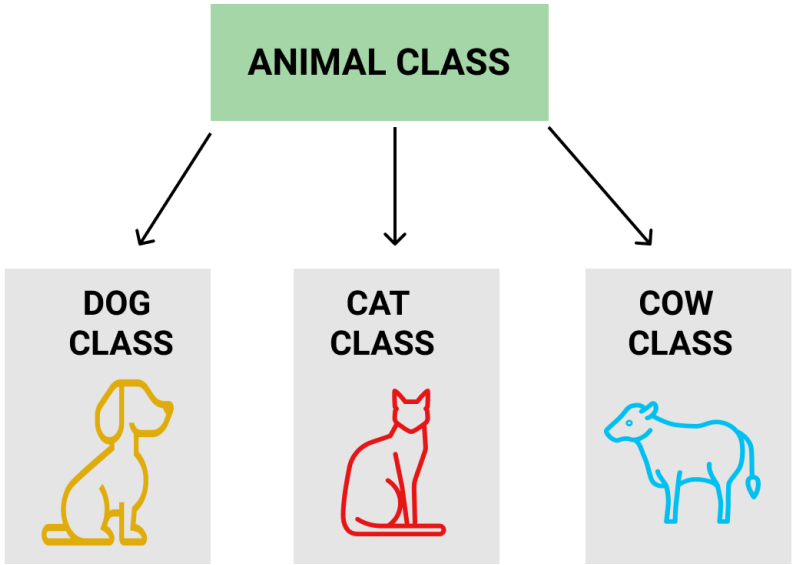
OOP Concepts

- Encapsulation: In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit.
- In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them

OOP Concepts

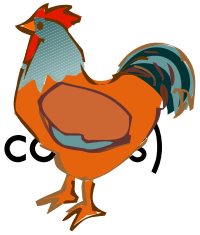
- Polymorphism: The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

- Inherite ve properties and cho ed Inheritance.



Intro Example: A Trip to the Aviary

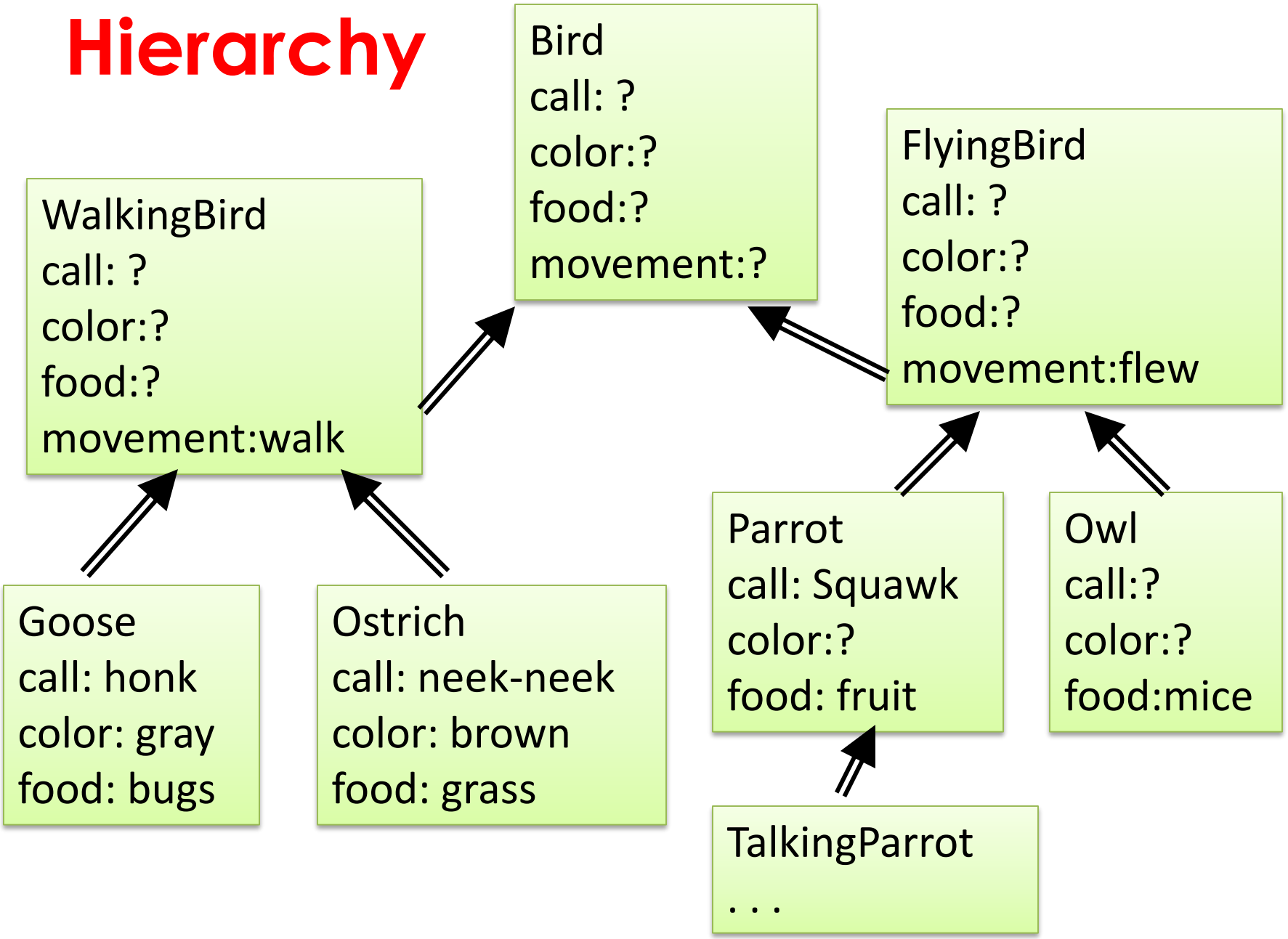
- Consider a collection of birds which have different properties
  - name
  - color (some of the same name are of different colors)
  - they eat different things
  - they make different noises
  - some make multiple kinds of sounds



Design Sketch

- Key is to design a `Bird` class hierarchy.
- Strategy
  - design classes for objects
  - identify characteristics classes have in common
  - design superclasses to store common characteristics

Hierarchy


Bird
call: ?
color:?
food:?
movement:?
WalkingBird
call: ?
color:?
food:?
movement:walk
FlyingBird
call: ?
color:?
food:?
movement:flew
Goose
call: honk
color: gray
food: bugs
Ostrich
call: neek-neek
color: brown
food: grass
Parrot
call: Squawk
color:?
food: fruit
Owl
call:?
color:?
food:mice
TalkingParrot
. . .

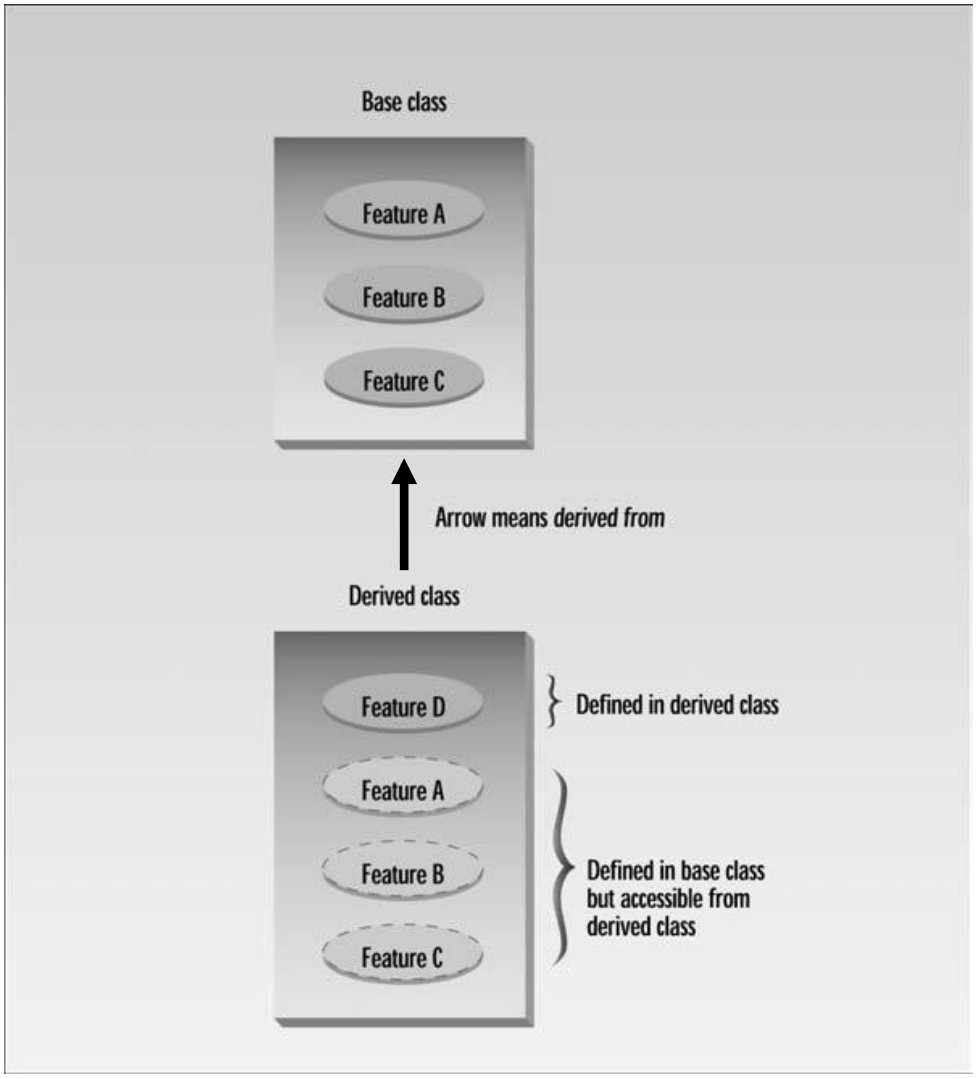
Inheritance

- Inheritance is probably the most powerful feature of object-oriented programming, after classes themselves.

- 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 embellishments and refinements of its own.

- The base class is unchanged by this process.



Inheritance: the arrow is interpreted as “derived from” arrow.

Inheritance

- Inheritance is an essential part of OOP. Its big payoff is that it permits code reusability.
- Once a base class is written and debugged, it need not be touched again.
- Reusing existing code saves time and money and increases a program's reliability.
- An important result of reusability is the ease of distributing class libraries.
- 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.

Specifying the Base Class

```
// counten.cpp; // inheritance with Counter class
#include <iostream>
using namespace std;
class Counter //base class
{
protected: //NOTE: not private
unsigned int count; //count
public:
Counter() : count(0) //no-arg constructor
{ }
Counter(int c) : count(c) //1-arg constructor
{ }
```

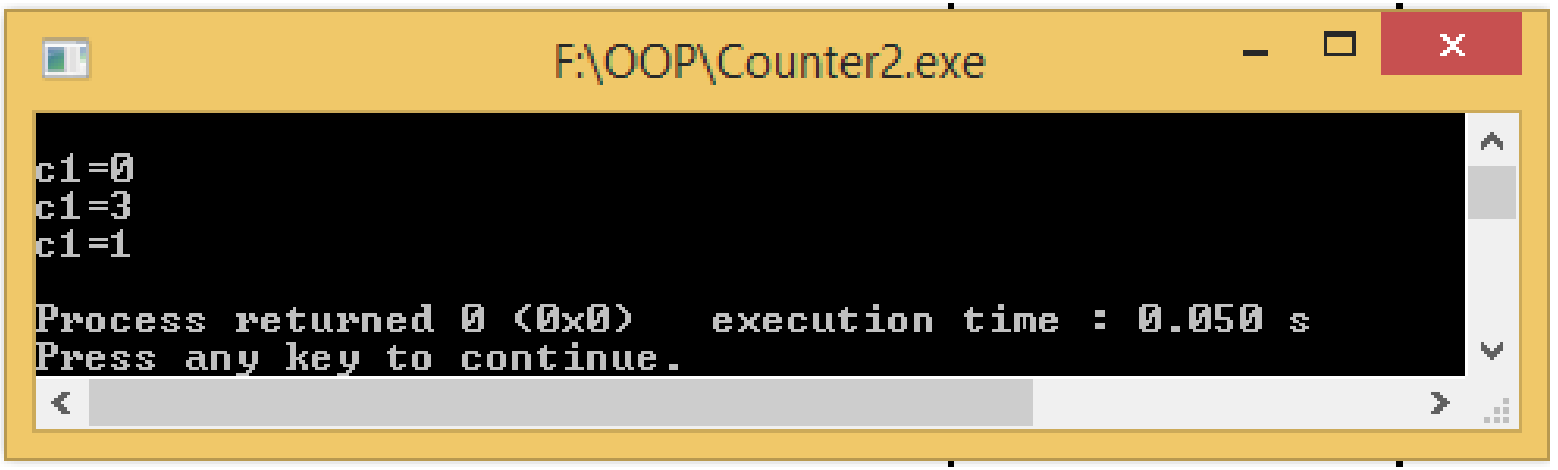
Specifying the Base Class (cont.)

```
unsigned int get_count() const //return count
{ return count; }
int inc_counter() //incr count (prefix)
{ return ++count; }
};
////////////////////////////////////////////////
class CountDn : public Counter //derived class
{
public:
Int dec_counter() //decr count (prefix)
{ return --count; }
};
```

Specifying the Base Class (cont.)

```
int main()
{
CountDn c1; //c1 of class CountDn
cout << "\nc1=" << c1.get_count(); //display c1
c1.inc_counter();  //increment c1, 3 times
c1.inc_counter();
c1.inc_counter();
cout << "\nc1=" << c1.get_count(); //display it
c1.dec_counter(); //decrement c1
cout << "\nc1=" << c1.get_count(); //display it
cout << endl;
return 0;
}
```

Specifying the Derived Class



Specifying the Derived Class

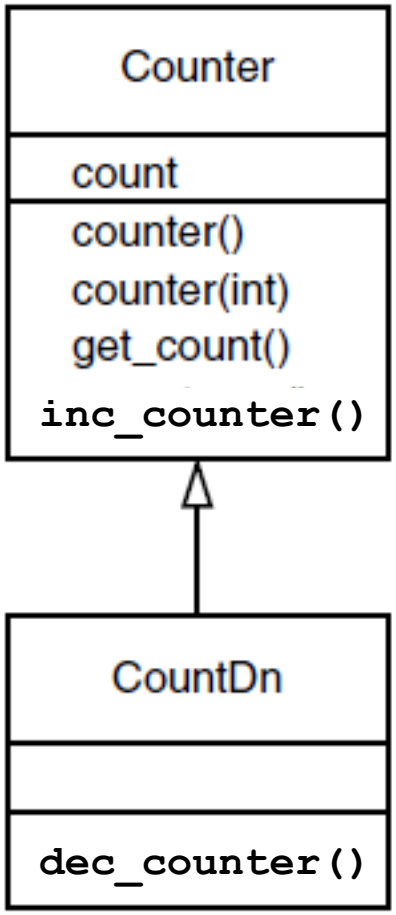
The listing starts off with the Counter class.

After Counter class declaration;

```
class CountDn : public Counter
{
};
```

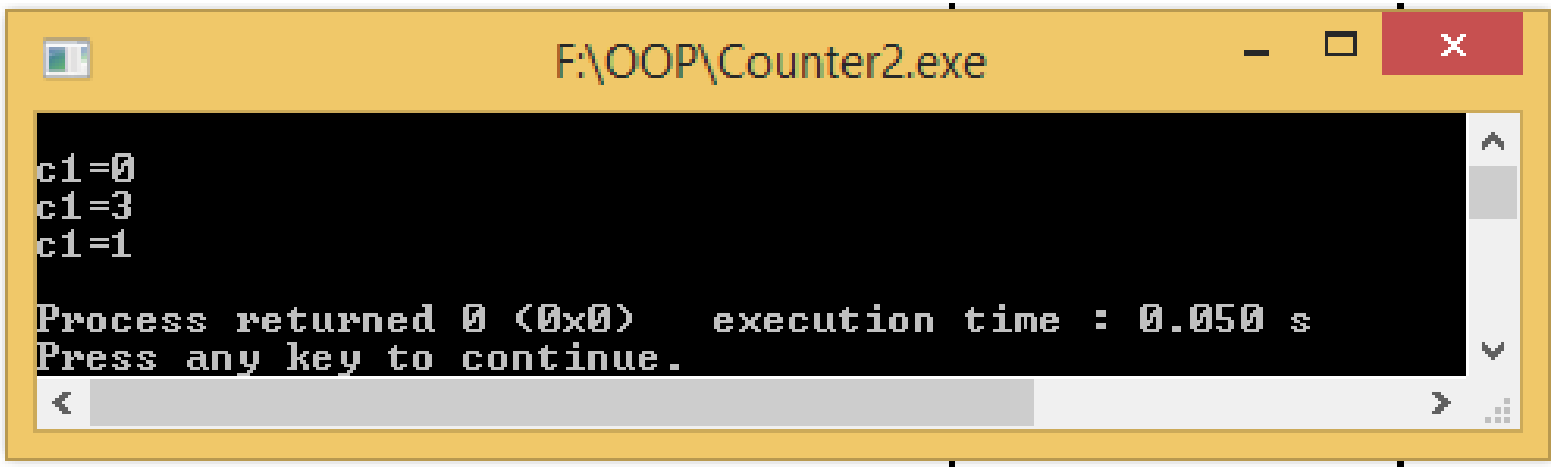
Generalization in UML Class Diagrams

Remember that the arrow means *inherited from* or *derived from* or *is a more specific version of*.



UML class diagram for COUNTER

Output of COUNTER



- c1=0 ←after initialization
- c1=3 ←after inc_counter(), 3 times
- c1=2 ←after dec_counter()
- 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.

Accessing Base Class Members

- An important topic in inheritance is knowing when a member function in the base class can be used by objects of the derived class. This is called accessibility.

Substituting Base Class Constructors

In the main() part of COUNTER we create an object of class

CountDn:

CountDn c1;

- This causes c1 to be created as an object of class CountDn and initialized to 0. why?

- It turns out that—at least under certain circumstances—if we don't specify a constructor, the derived class will use an appropriate constructor from the base class.

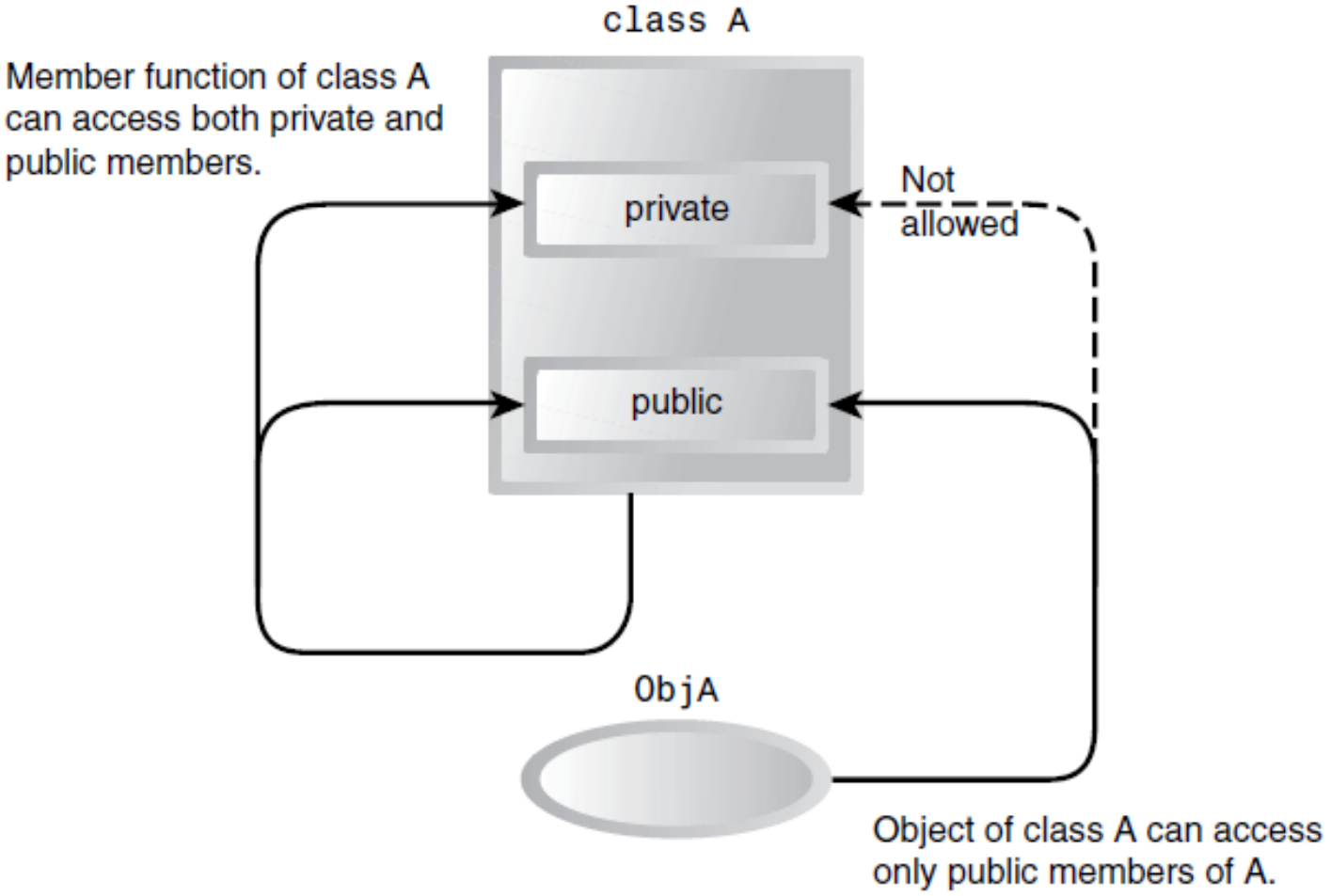
Substituting Base Class Member Functions

- The object c1 of the CountDn class also uses the inc_counter() and get_count() functions from the Counter class. How?
- Again the compiler, not finding these functions in the class of which c1 is a member, uses member functions from the base class.

The protected Access Specifier

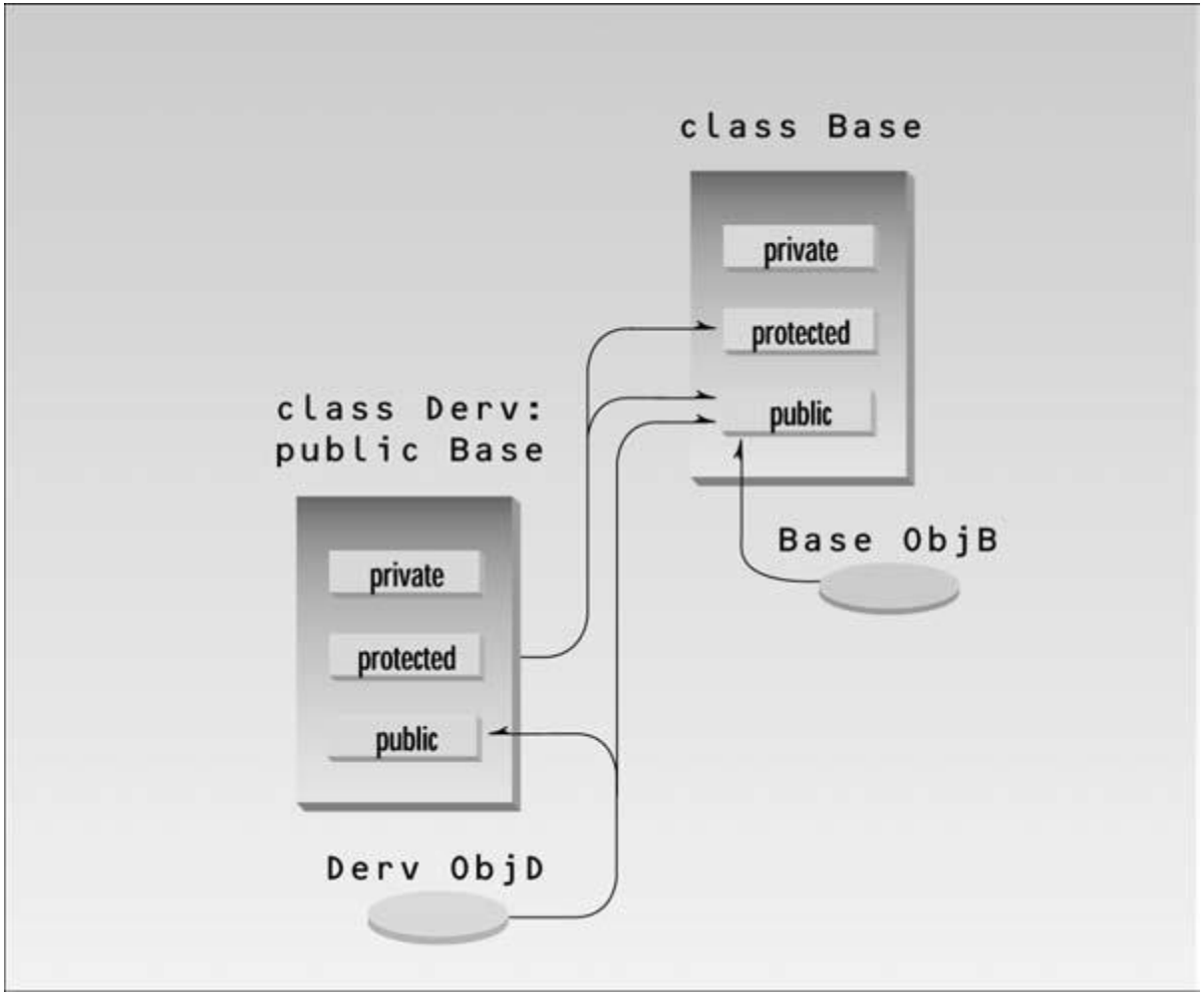
- Let's first review what we know about the access specifiers private and public.
- With inheritance, however, there is a whole raft of additional possibilities.
- The question that concerns us at the moment is, can member functions of the derived class access members of the base class?
- In other words, can dec_counter() in CountDn access count in Counter?
- The answer is that 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.

The public and private Access Specifier



Access specifiers without inheritance

The protected Access Specifier



Access specifiers with inheritance

The protected Access Specifier

- **The moral** is that if you are writing a class that you suspect might be used, at any point in the future, as a base class for other classes, then any member data that the derived classes might need to access should be made **protected** rather than **private** means -> "inheritance ready."

Dangers of protected

- Disadvantage to make class members protected is that the library's protected members can be accessible by simply deriving other classes from them.

- To avoid corrupted data, it's often safer to force derived classes to access data in the base class **using only public functions** in the base class, just as ordinary main() programs must do.

The protected Access Specifier

TABLE : 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

Base Class Unchanged

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

Other Terms

In some languages the base class is called the superclass 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.

```
c1.inc_count();

++c1;

void operator ++  ()
```

This declarator syntax tells the compiler to call this member function whenever the ++ operator is encountered, provided the operand

Next

- What happens if we want to initialize a CountDn object to a value?
- Can the one-argument constructor in Counter be used?

Derived Class Constructors

```
// counter2_2.cpp
// constructors in derived class
#include <iostream>
using namespace std;
/////////
class Counter
{
protected: //NOTE: not private
unsigned int count; //count
public:
Counter() : count() //constructor, no args
{ }
Counter(int c) : count(c) //constructor, one arg
{ }
```

Derived Class Constructors (cont.)

```
unsigned int get_count() const //return count
{ return count; }
Counter operator ++ () //incr count (prefix)
{ return Counter(++count); }
};
////////
```

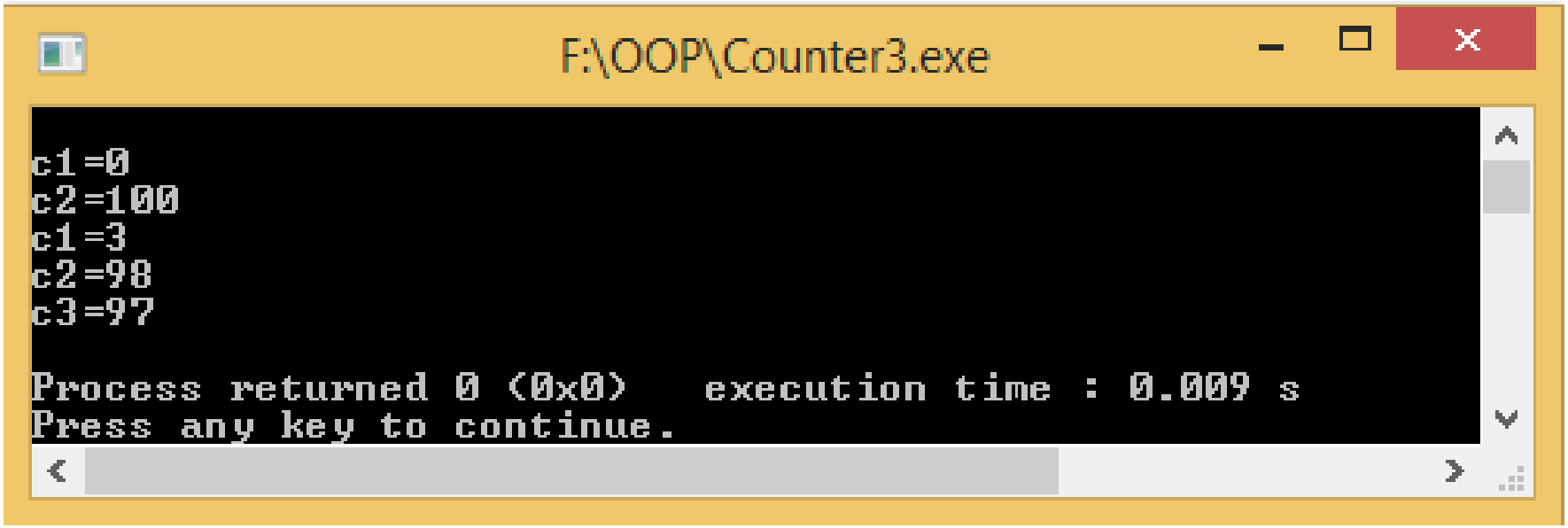
Derived Class Constructors (cont.)

```
class CountDn : public Counter
{
public:
CountDn() : Counter() //constructor, no args
{ }
CountDn(int c) : Counter(c) //constructor, 1 arg
{ }
CountDn operator -- () //decr count (prefix)
{ return CountDn(--count); }
};
//////////////
```

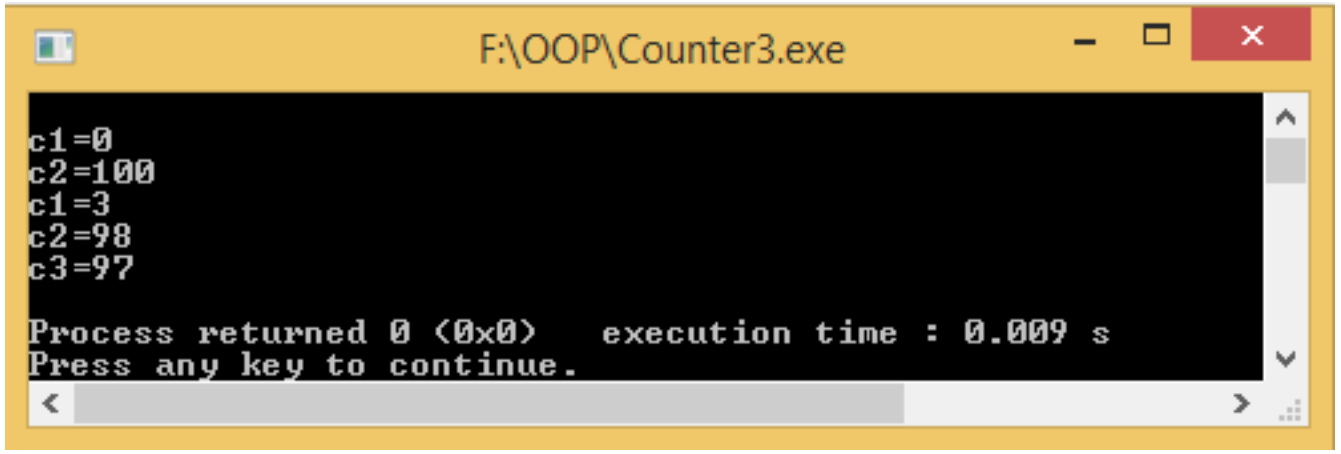
Derived Class Constructors (cont.)

```
int main()
{
CountDn c1; //class CountDn
CountDn c2(100);
cout << "\nc1=" << c1.get_count(); //display
cout << "\nc2=" << c2.get_count(); //display
++c1; ++c1; ++c1; //increment c1
cout << "\nc1=" << c1.get_count(); //display it
--c2; --c2; //decrement c2
cout << "\nc2=" << c2.get_count(); //display it
CountDn c3 = --c2; //create c3 from c2
cout << "\nc3=" << c3.get_count(); //display c3
cout << endl;
return 0;
}
```

Derived Class Constructors (cont.)



Derived Class Constructors (cont.)



This program uses two new constructors in the CountDn class. Here is the no-argument constructor:

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

the function name following the colon. This construction causes the CountDn() constructor to call the Counter() constructor in the base class.

Derived Class Constructors (cont.)

CountDn c2(100);

This constructor also calls the corresponding one-argument constructor in the base class:

CountDn(int c) : Counter(c) ← argument c is passed to Counter
{ }

The one-argument constructor is also used in an assignment statement.

CountDn c3 = --c2;

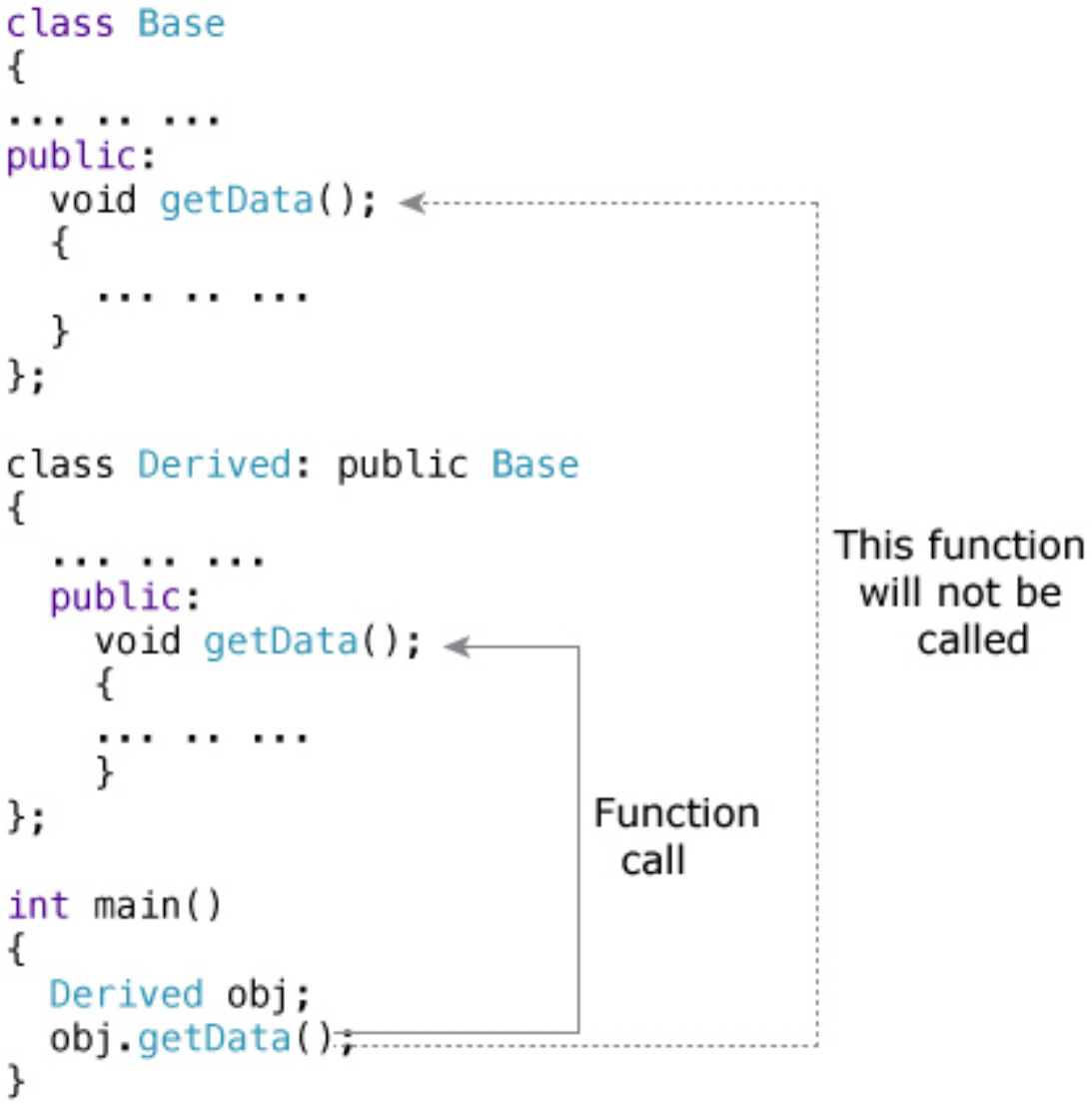
Overriding Member Functions

Suppose, both base class and derived class have a member function with same name and arguments (number and type of arguments).

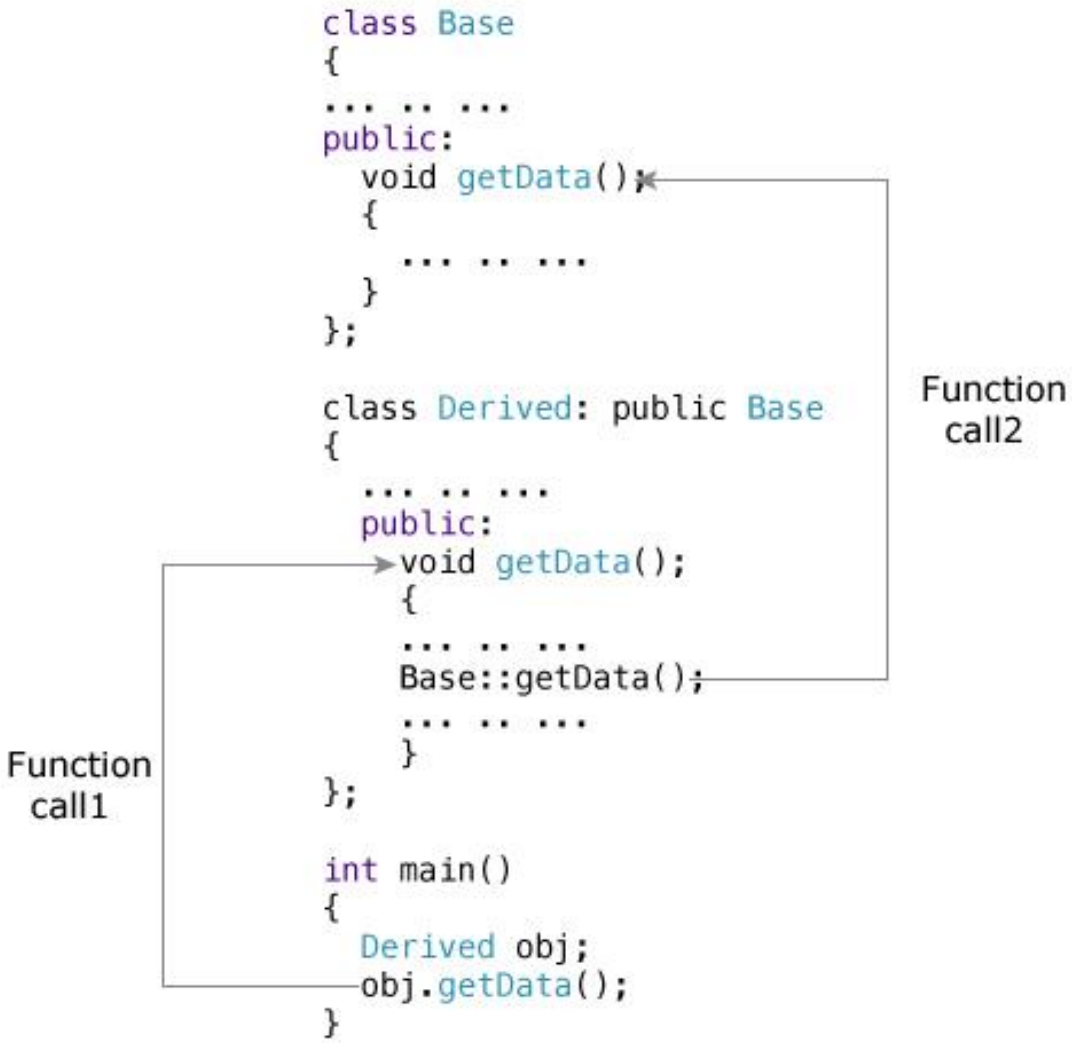
If we create an object of the derived class and call the member function which exists in both classes (base and derived), the member function of the derived class is invoked and the function of the base class is ignored.

This feature in C++ is known as **function overriding.**

Overriding Member Functions



How to access the overridden function in the base class from the derived class?

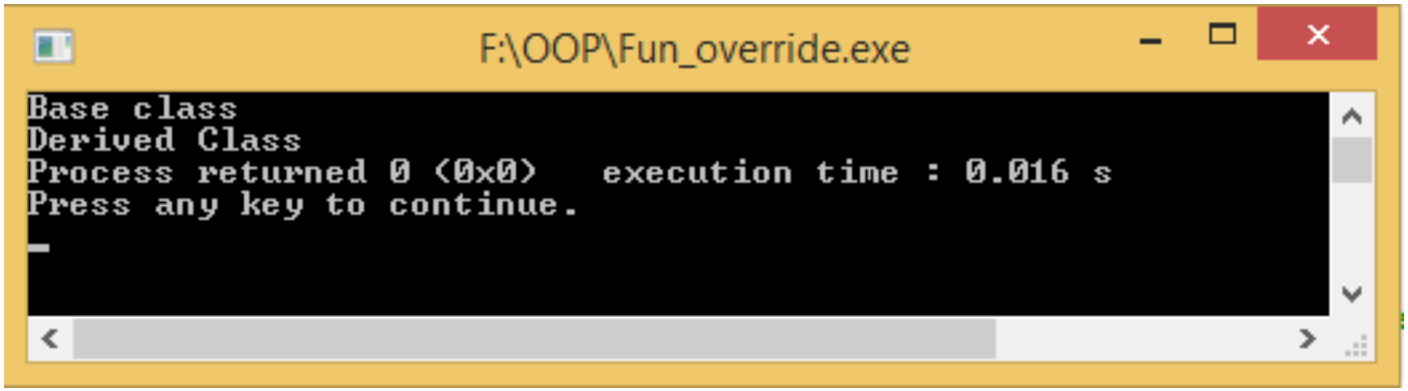


Overridden function

```
#include<iostream>
using namespace std;
class Base
{
 public:
 void show()
 {  cout << "Base class\t"; }
};

class Derived:public Base
{
 public:
 void show()
 { cout << "Derived Class"; }
};
```

Overridden function

```
int main()
{
   Base b;       //Base class object
   Derived d;     //Derived class object
   b.show();     //Early Binding Occurs
   d.show();
}
```



Overridden function (cont.)

Now change the main function:

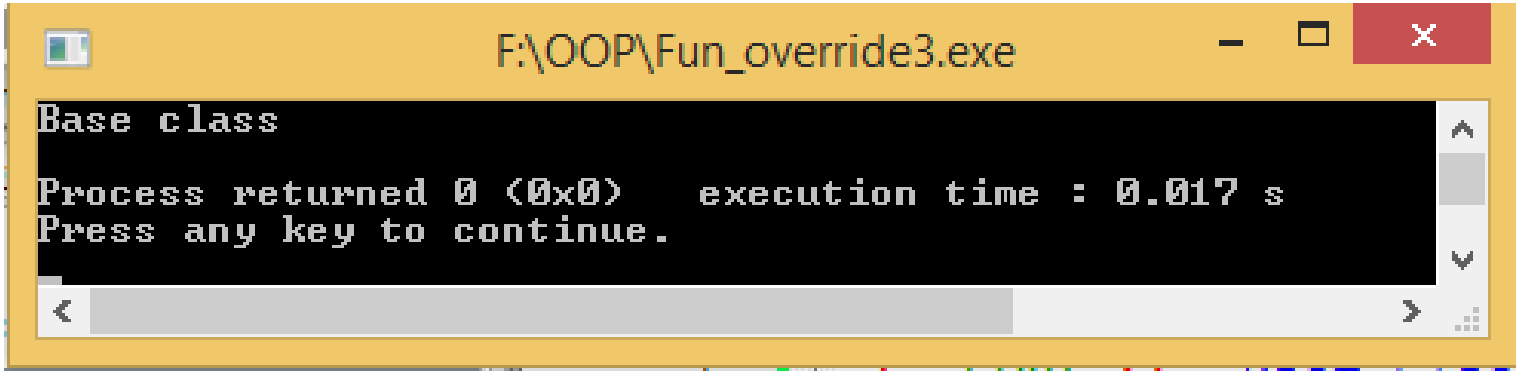
```
int main()
{
 Base* b;        //Base class pointer
 Derived d;      //Derived class object
 b = &d;         //passing derived class address into
base class pointer
 b->show();      //Early Binding Occurs
}
```

Output:?

Overridden function (cont.)

Now change the main function:

```
int main()
{
 Base* b;       //Base class pointer
 Derived d;     //Derived class object
 b = &d;        //passing derived class address into
base class pointer
 b->show();     //Early Binding Occurs
}
```



Overridden function (cont.)

In the previous example, although, the object is of Derived class, still Base class's method is called. This happens due to Early Binding.

Compiler on seeing Base class's pointer, set call to Base class's show() function, without knowing the actual object type.

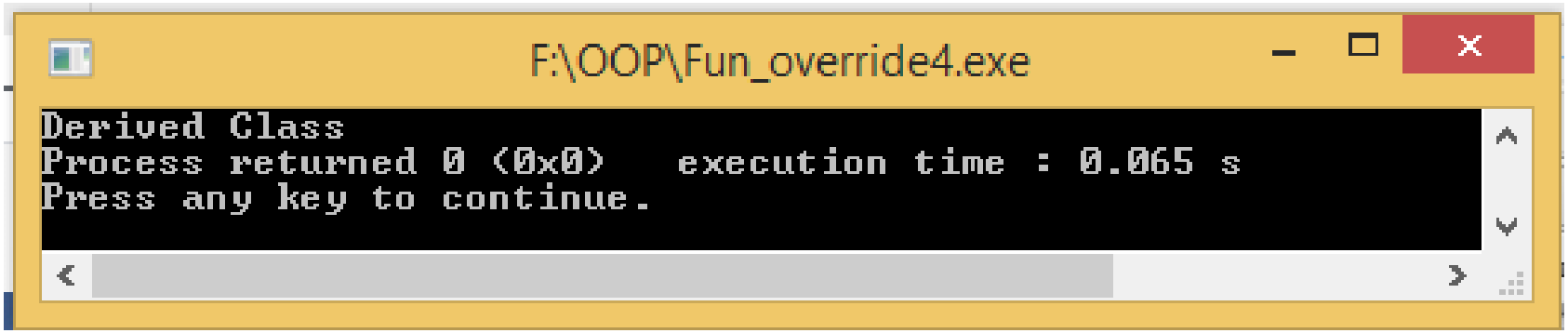
When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function

Using Virtual Keyword in C++

Again change the class Base and main function:

```
class Base //Fun_override4.cpp
{
    public:
    virtual void show()
    {  cout << "Base class\n"; }
}; // class B declaration same as previuos.
                int main()
                {
                 Base* b;        //Base class pointer
                 Derived d;      //Derived class object
                 b = &d;         //passing derived class address into
                base class pointer
                 b->show();      // ? Binding Occours
                }   //Output: ?
```

Using Virtual Keyword in C++



Using Virtual Keyword in C++

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform Late Binding on this function.

Virtual Keyword is used to make a member function of the base class Virtual.

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called **Dynamic Binding** or **Runtime Binding.**

Using Virtual Keyword and Accessing Private Method of Derived class

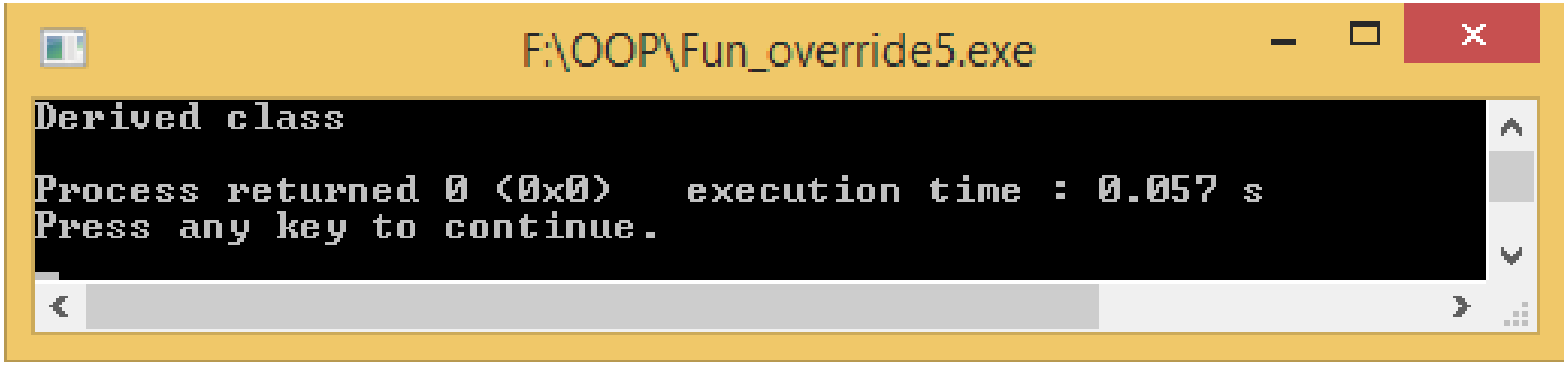
```
#include <iostream> //Fun_override.cpp
using namespace std;
class A
{
    public:
    virtual void show()
    {  cout << "Base class\n";  }
};

class B: public A
{
    private:
    virtual void show()
    { cout << "Derived class\n"; }
};
```

Using Virtual Keyword and Accessing Private Method of Derived class

```
int main()
{
    A *a;
    B b;
    a = &b;
    a->show();
}
```

Output: ?

Using Virtual Keyword and Accessing Private Method of Derived class



We can call private function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

"Abstract" Base Class

Classes used only for deriving other classes, as employee is in EMPLOY, are sometimes loosely called abstract classes, meaning that no actual instances (objects) of this class are created.

“Abstract” Base Class

```
//Abstract base class
class Base
{
    public:
    virtual void show() = 0;    // Pure Virtual Function
};

class Derived:public Base
{
    public:
    void show()
    {
        cout << "Implementation of Virtual Function in Derived class\n";
    }
};
```

"Abstract" Base Class

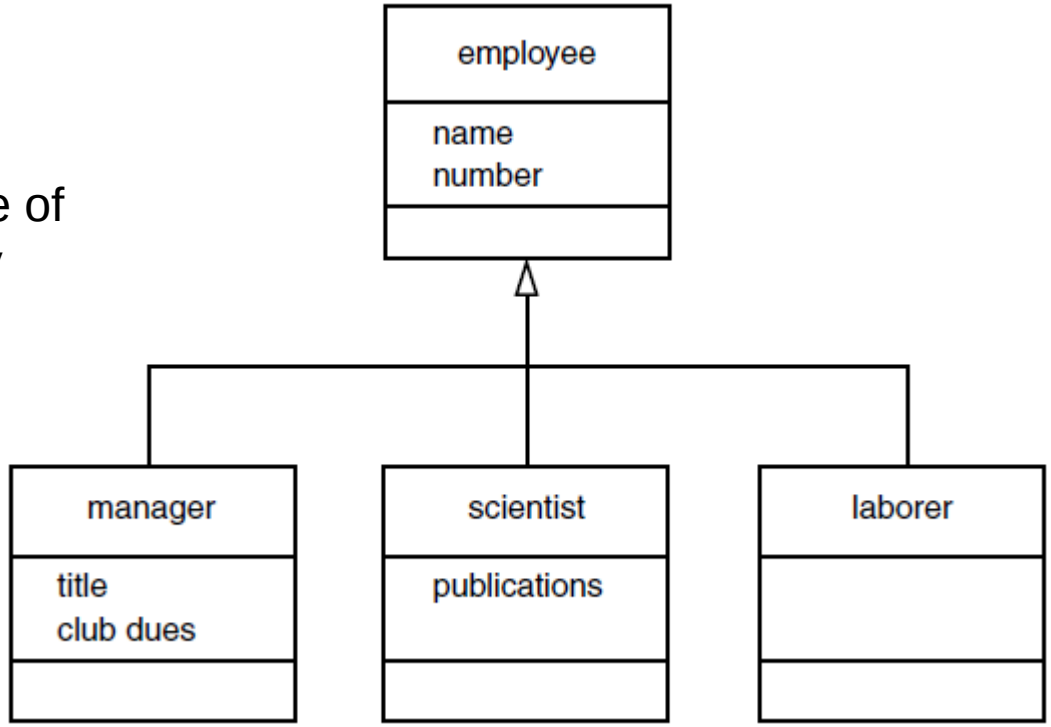
```
int main()
{
    Base obj;   //Compile Time Error
    Base *b;
    Derived d;
    b = &d;
    b->show();
}
```

Pure virtual Functions are virtual functions with no definition.
They start with virtual keyword and ends with = 0.

Class Hierarchies

So far inheritance has been used to add functionality to an existing class. Now let's look at an example where inheritance is used for a different purpose: as part of the original design of a program.

Our example models a database of employees of a widget company



Public and Private Inheritance

C++ provides a wealth of ways to fine-tune access to class members. One such access-control mechanism is the way derived classes are declared.

Public and Private Inheritance

Ex. class manager : public employee

Implications:

- keyword public: specifies that objects of the derived class are able to **access** public member functions of the base class.

- keyword private : When this 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.

Access Combinations

class manager : public employee

- keyword public: specifies that objects of the derived class are able to **access** public member functions of the base class.
- keyword private : When this 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.

Access Combinations

```
// pubpriv.cpp
// tests publicly- and privately-derived classes
#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;
};
//////////
```

Access Combinations

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

Access Combinations

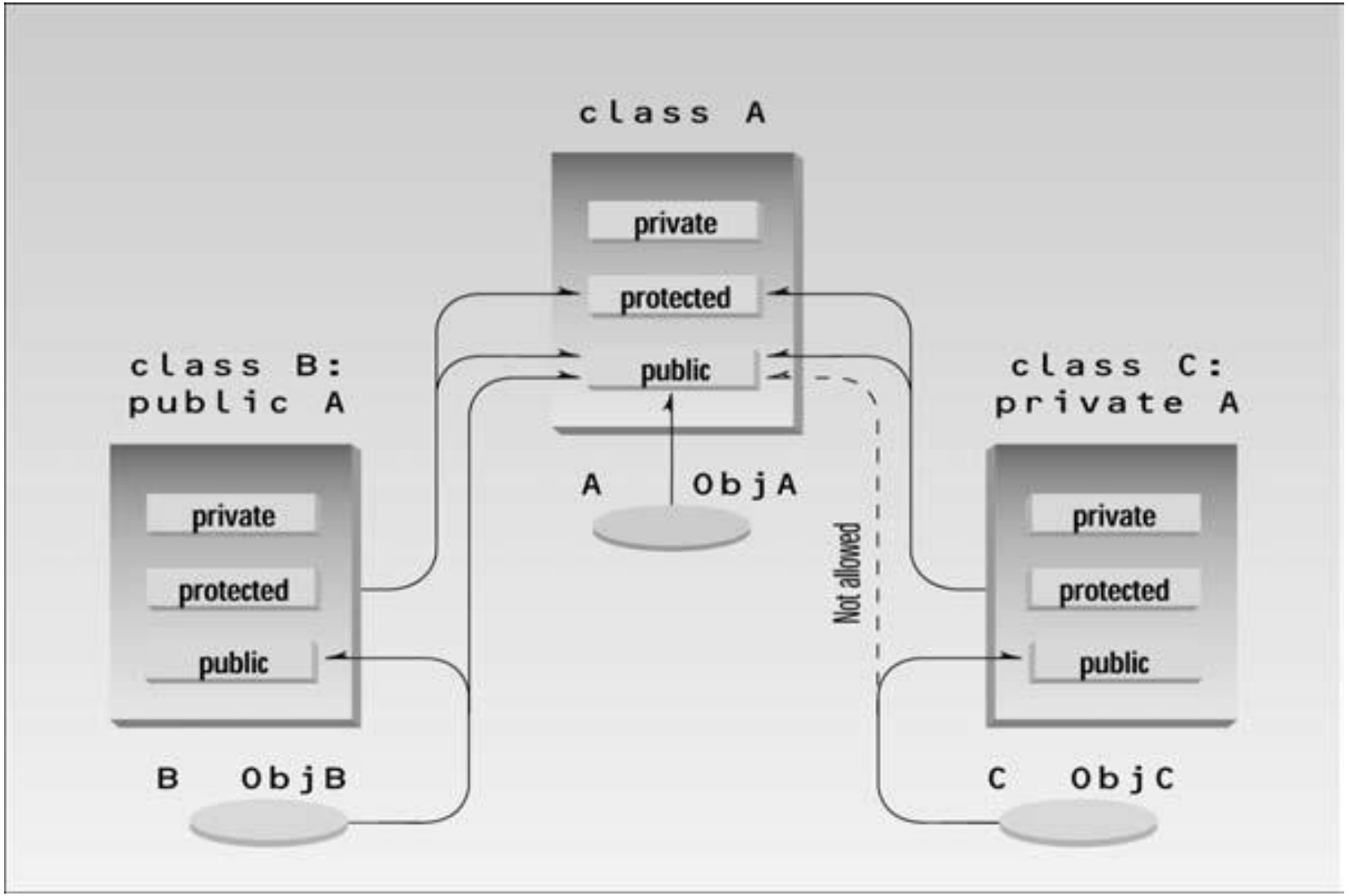
```
class C : private A //privately-derived class
{
public:
void funct()
{
int a;
a = privdataA; //error: not accessible
a = protdataA; //OK
a = pubdataA; //OK
}
};
```

Access Combinations

```
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;
}
```

Access Combinations

- As we've seen before, 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.
- What's new is the difference between **publicly derived** and **privately derived classes**.
- 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 and private derivation

```
class Base
{
public:
    int a;
protected:
    int b;
private:
    int c;
}
```

```
class Derived_Public : public Base
{
//a is public
//b is protected
//c is inaccessible */
};
```

```
class Derived_Protected :
protected Base
  {
   // a is protected
   //b is protected
  //c is inaccessible */
   };
```

```
class Derived_private : private
  Base
{
// a is private
//b is private
//c is inaccessible */
}
```

When to use what Inheritance

1. In most cases a derived class exists to offer an improved—or a more specialized—version of the base class.

2. In some situations, however, the derived class is created as a way of completely modifying the operation of the base class, hiding or disguising its original interface.