

Chapter 7: Inheritance

1) Introduction.....	7-2
2) Inheritance - public base classes	7-4
3) The protected Access Level.....	7-9
4) Member Initialization Lists.....	7-10
5) What Isn't Inherited?	7-14
6) Assignments Between Base And Derived Objects.....	7-15
7) Compile-Time vs. Run-Time Binding	7-16
8) virtual Functions.....	7-19
9) Polymorphism	7-25
10) virtual Destructors.....	7-26
11) Pure virtual Functions.....	7-27
12) Abstract Base Classes	7-29
13) An Extended Inheritance Example	7-30
14) Exercises.....	7-34

Introduction

- Object-oriented languages generally share three characteristics:
 - ▶ encapsulation
 - ▶ inheritance
 - ▶ polymorphism
- **Encapsulation** couples functions and data as a single entity.

- **Polymorphism** (many forms) refers to the same message invoking a different method. There are different forms of polymorphism.

```
int a,b,c;  
double x,y,z;  
  
c = a + b;           // operator overloading  
x = y + z;  
  
print(a);           // function overloading  
print(x);
```

- **Inheritance** gives the capability of reusing software by specializing existing general solutions.

Introduction

- Languages that support inheritance provide a way of creating `type - subtype` relationships.
- An existing class can be thought of as a generalization for a more specialized class.

```
class Vehicles;           // general
class Cars;              // specific
class Trucks;           // specific
```

- More specialized classes can be derived from the general class.
- Methods and data from the general classes are inherited by the specialized classes. This leads directly to code reuse.
- The specialized class will normally add methods and/or data to support the specialization.
- Large class hierarchies can be built.
- In C++, the general class is called the **base** class and the specialized class is called the **derived** class. Neither **base** nor **derived** are keywords.

Inheritance - public base classes

- Modeling a derived class as a specialization of a base class is often called the *is-a* relationship. To specify that one class is derived from another, the following notation is used.

```
class Loan
{
    ...
    ...
};
class BusinessLoan : public Loan
{
    ...
    ...
};
```

- ▶ Loan is the **base** class; BusinessLoan is the **derived** class

```
class Airplane {
    ...
    ...
};
class DC10: public Airplane {
    ...
    ...
};
```

- ▶ Airplane is the **base** class; DC10 is the **derived** class

- `public` specifies that public inheritance is being modeled. This form of inheritance is used to model the *is-a* relationship. Although C++ also allows `private` and `protected` inheritance, these are rarely used.

Inheritance - public base classes

- Recall the `Loan` class. There are many different loan types on the market. There are business loans, car loans, and home mortgage loans. Each of these is-a special type of loan. Thus each can be derived from `Loan`. Here's the `Loan.h` file with header guards.

`Loan.h`

```
1. //
2. //   Loan.h
3. //
4. #ifndef LOAN
5. #define LOAN
6. #include <string>
7. using namespace std;
8. class Loan
9. {
10. private:
11.     string name;
12.     double amount, rate;
13.     int years;
14. public:
15.     Loan(string, int, double, double);
16.     Loan(string, int, double);
17.     Loan(string, int);
18.     //Loan() { } Explanation to follow
19.     string getName();
20.     double getAmount();
21.     int getYears();
22.     double getRate();
23.     void setName(string n);
24.     void setAmount(double a);
25.     void setYears(int y);
26.     void setRate(double r);
27. };
28. #endif LOAN
```

Inheritance - public base classes

- To derive Mortgage from Loan, we write:

Mortgage.h

```
1. // Mortgage.h
2. //
3. #ifndef MORTGAGE
4. #define MORTGAGE
5. #include <string>
6. using namespace std;
7. class Mortgage : public Loan
8. {
9. private:
10.     double payment;
11.     double balance;
12.     double principal;
13.     double interest;
14.     double computePayment();
15. public:
16.     Mortgage(string, int, double, double);
17.     double getPayment();
18.     void    makePayment();
19.     double getBalance();
20.     double getPrincipal();
21.     double getInterest();
22. };
23. #endif
```

- A Mortgage is a Loan and thus Mortgage objects have use of all (non-constructor) Loan methods in addition to the Mortgage methods. This is a good example of code reuse, one of the major benefits of inheritance.
- The file where Mortgage methods are implemented is shown on the following page.

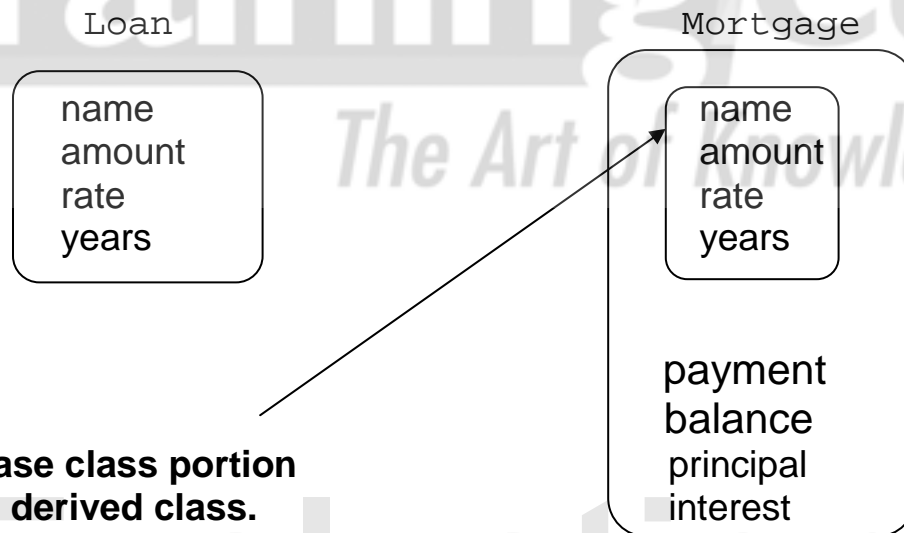
Inheritance - public base classes

MortgageFunctions.cpp

```
1. #include "Loan.h"
2. #include "Mortgage.h"
3. #include <iostream>
4. #include <math.h>
5. using namespace std;
6. void Mortgage::makePayment()
7. {
8.     // code for calculating payment
9. }
10. Mortgage::Mortgage(string name, int years,
11.    double amt, double rate)
12. {
13.     setName(name);
14.     setYears(years);
15.     setAmount(amt);
16.     setRate(rate);
17.     payment = computePayment();
18.     balance = amt;
19. }
20. double Mortgage::getPayment()
21. {
22.     return payment;
23. }
24. double Mortgage::getBalance()
25. { return balance;
26. }
27. double Mortgage::computePayment()
28. {
29.     See exercise 2
30. }
31. double Mortgage::getInterest()
32. { return interest;
33. }
34. double Mortgage::getPrincipal()
35. { return principal;
36. }
```

Inheritance - public base classes

- The code for `computePayment` and `makePayment()` is dependent on data from the base class.
- If base class data is `private`, then even derived class methods cannot access them. This is true even though a derived class inherits data from its base class.
- In this example, the data contained within each object is displayed below. Thus we have come upon the unusual situation whereby a derived class object cannot access its base class portions.



- Of course you can solve this problem by using the base class public interface. Or you can use the `protected` access level.

The protected Access Level

- If a member is `protected`, then it is as though it were declared as `public` with respect to derived classes but as though it were declared as `private` with respect to the rest of the program.
- This means that these items can be accessed in deeper derived classes as well.
- If you are implementing a set of classes, some of which will be base classes for derived classes, then it is a good idea that these classes declare their data `protected` rather than `private`. This makes it easier for derived class methods to access their base class data.

<code>rate</code>	rather than	<code>getRate()</code>
<code>amount</code>	rather than	<code>getAmount()</code>

Evaluation
Copy

Member Initialization Lists

- There is an important issue relating to inheritance in C++ that we have overlooked. Since a derived class contains base class data, any constructor in the derived class is responsible for initializing its base class data.
- In the code below, this is handled through the set member functions provided by the `Loan` class.

```
Mortgage::Mortgage( string name, int years,
                   double amt, double rate)
{
    payment = computePayment();
    balance = amt;
    setName(name);
    setYears(years);
    setAmount(amt);
    setRate(rate);
}
```

- Since the compiler is well aware of the inheritance relationship between `Loan` and `Mortgage`, it expects that any `Mortgage` constructor will call a `Loan` constructor to initialize the `Loan` data inside the `Mortgage` object.
- The initialization is handled by a member initialization list. When we last visited this topic, it was to ensure that contained objects had their constructors called from within containing constructors.

```
Line::Line(const Point & a1, const Point & a2)
: p1(a1), p2(a2) {}
```

Member Initialization Lists

- In this case there are no contained objects but rather there is an inheritance relationship. So the initialization list simply names the appropriate constructor.

```
Mortgage::Mortgage( string name, int years,
                  double amt, double rate)
: Loan(name, years, amt, rate)
{
    payment = computePayment();
    balance = amt;
}
```

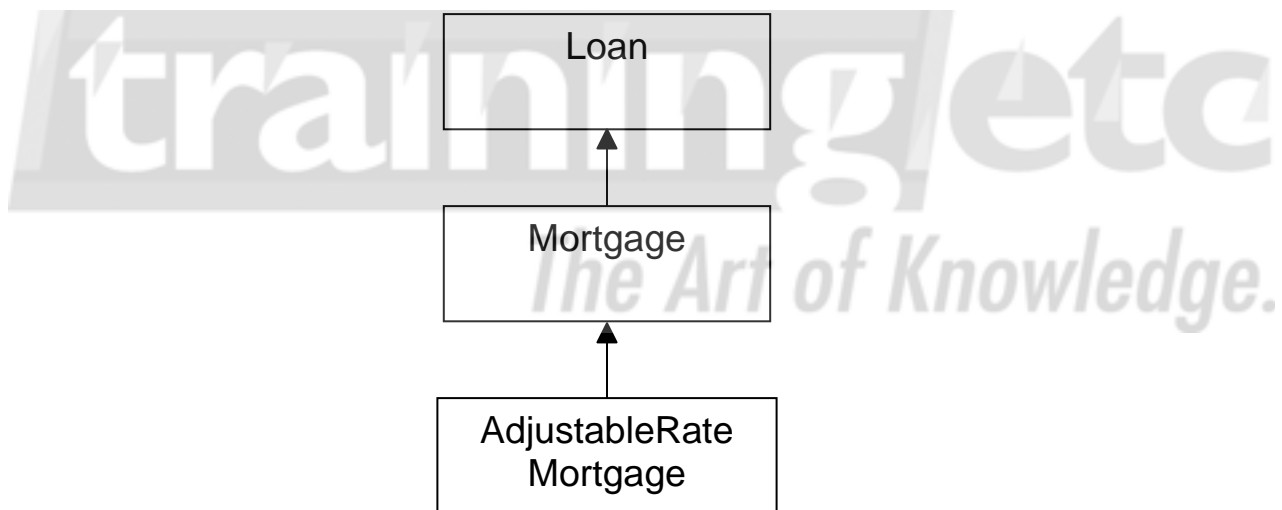
- If you leave out the initialization list, the compiler will try to execute the default `Loan` constructor. If one is not present, an error is issued.

- ▶ In fact, `MortgageFunctions.cpp` will not compile for that very reason.

Evaluation
Copy

Member Initialization Lists

- Working with initialization lists is simple. Regardless of the depth of an inheritance hierarchy, a constructor anywhere in the hierarchy need only be responsible for its immediate base class.
- Thus if an `AdjustableRateMortgage` is derived from `Mortgage`, `AdjustableRateMortgage` constructors need only call the `Mortgage` constructor, which in turn will call the `Loan` constructor.



- Following is the skeleton code for such a scenario.

Member Initialization Lists

7-13.cpp

```
1. //
2. // 7-13.cpp
3. //
4. #include <iostream>
5. using namespace std;
6. class Loan
7. {
8. public:
9.     Loan()
10.    {
11.        cout << "Loan Constructor\n";
12.    }
13.    int getID(){ return 1; }
14. };
15. class Mortgage : public Loan
16. {
17. public:
18.     Mortgage()
19.    {
20.        cout << "Mortgage Constructor\n";
21.    }
22.    int getID(){ return 2; }
23. };
24. class AdjustableRateMortgage : public Mortgage
25. {
26. public:
27.     AdjustableRateMortgage()
28.    {
29.        cout << "ARM Constructor\n";
30.    }
31.    int getID() { return 3; }
32. };
33. int main()
34. {
35.     AdjustableRateMortgage arm;
36.     return 0;
37. }
```

What Isn't Inherited?

- Not all `public` and `protected` function members are inherited under public derivation.
- Constructors are not inherited. This makes sense because constructing a base class object is very different than constructing a derived class object.
- Destructors are likewise not inherited. A derived class may have its own cleanup to handle and thus it cannot rely on a base class constructor.
- Assignment operators are not inherited either. A base class assignment could never know about assigning derived class data.

The Art of Knowledge.

Evaluation
Copy

Assignments Between Base And Derived Objects

- Any derived object can be assigned to a base class object because of the *is-a* relationship. That is, a Mortgage is a Loan. However the opposite is not true.

```
Loan myLoan;  
Mortgage myMortgage;  
AdjustableRateMortgage myArm;
```

```
myLoan = myArm;  
myLoan = myMortgage;
```

- After the last assignment, `myLoan` consists of the sliced Mortgage, that is the Loan portion of `myMortgage`.
- Likewise, a base class pointer can be assigned the address of any derived object! Of course the opposite is not true.

```
Loan myLoan, *lp;  
Mortgage myMortgage;  
AdjustableRateMortgage myArm;
```

```
lp = &myLoan;  
lp = &myArm;  
lp = &myMortgage;
```

- Things start to get interesting when a pointer is used to launch a method.

Compile-Time vs. Run-Time Binding

- For example, suppose each class below has a `getID()` method. Which `getID()` method would get executed in the code below?

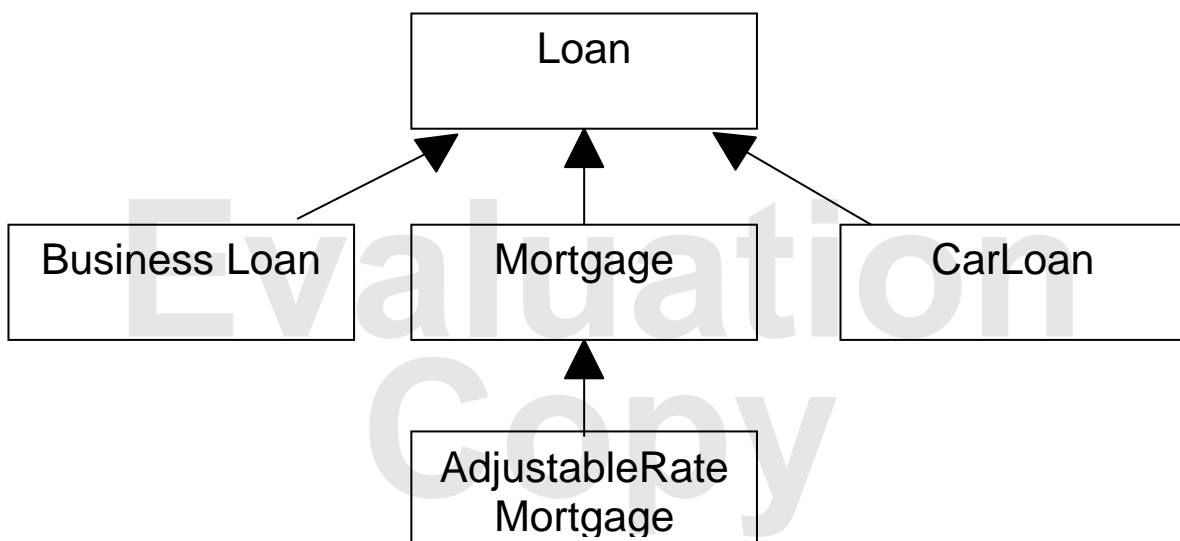
```
Loan myLoan, *lp;  
Mortgage myMortgage;  
AdjustableRateMortgage myArm;  
int id;
```

```
lp = &myLoan;  
id = lp -> getID();
```

```
lp = &myArm;  
id = lp -> getID();
```

```
lp = &myMortgage;  
id = lp -> getID();
```

- To answer this question, first imagine a large set of classes related through public inheritance.



Compile-Time vs. Run-Time Binding

- Each class would have its own methods to implement its respective behavior.
- Sometimes one method provides the ideal behavior for a specific action independent of the specialized class. For these methods, the developer will always want the base class method to act as the reusable default for that action.
- In C++ this is exactly what you get. In other words, unless you have taken measures to the contrary, the same base class `getId()` method will be executed in the code on page 16.
 - ▶ Method such as `getId()`, whose behavior is invariant over specialization, should only be coded in the base class and should be reused by all sub classes.
- On the other hand, some methods are such that their behavior varies over specialization.
 - ▶ For example, a `computePayment()` method would be implemented slightly differently depending upon what kind of loan we are considering.
- For methods such as this, we would like to execute the “correct” one on a class specific basis.

Compile-Time vs. Run-Time Binding

- You could get this behavior by naming the functions differently in each class. But then you would have to write code such as this:

```
if ( object isa Loan )
    LoanComputePayment();
else if ( object isa Mortgage )
    MortgageComputePayment();
...
```

- Code such as the above is both error prone and a maintenance hazard. Every time a new class is added, or deleted, each piece of code such as the segment above must be changed.
- A better solution is to let the compiler handle these details. In C++ this is accomplished by simply making the base class method a `virtual` method.
- You can think of pointers (or references) to class objects in a derivation hierarchy as having more than one type.
 - ▶ The type given by the compiler (early type, compile type).
 - ▶ The type given during execution (late type, run time type).
- A `virtual` function uses the run-time type of the pointer (or reference). This is called **late binding**.
- A **non-virtual** function uses the compile time type of the pointer (or reference). This is called **early binding**.

virtual Functions

- Simply by adding one word, `virtual`, to the following program, a different behavior for the program is effected.

7-19.cpp

```
1. #include <iostream>
2. using namespace std;
3. class Loan
4. {
5. public:
6.     Loan()
7.     {
8.         cout << "Loan Constructor\n";
9.     }
10.    virtual int getID() { return 1; }
11. };
12. class Mortgage : public Loan
13. {
14. public:
15.     Mortgage()
16.     {
17.         cout << "Mortgage Constructor\n";
18.     }
19.     int getID() { return 2; }
20.
21. };
22. class AdjustableRateMortgage : public Mortgage
23. {
24. public:
25.     AdjustableRateMortgage()
26.         : Mortgage()
27.     {
28.         cout << "AdjustableRateMortgage
29.     Constructor\n";
30.     }
31.     int getID() { return 3; }
32. };
33. int main()
34. {
35.     AdjustableRateMortgage arm;
36.     return 0;
37. }
```

virtual Functions

- Here are a few syntax caveats having to do with virtual functions:
 - ▶ You do not have to repeat the `virtual` keyword in derived classes although by doing so the readability of the program is enhanced.
 - ▶ A family of virtual functions must all have the same signature or the virtual chain is broken. This means they must have the same:
 - return type
 - argument list
 - const or non
- The word `virtual` can only appear in the class definition. When the virtual function is defined in the functions file, it is a syntax error to use the word "virtual".

Evaluation
Copy

virtual Functions

- Here's an example to summarize the differences between virtual and non-virtual functions. First we define a class named `Animal`.

7-21.cpp

```
1. #include <iostream>
2. #include <string>
3. using namespace std;
4. class Animal
5. {
6.     static int ids;
7.     protected:
8.     int id;
9.     string name;
10. public:
11.     Animal(string);
12.     int getID() const;
13.     virtual void speak() const;
14. };
15. int Animal::ids = 1;
```

- This class contains one `virtual` and one non-virtual function. Here are the `Animal` functions

```
Animal::Animal(string n)
{
    name = n;
    id = ids++;
}
int Animal::getID() const
{
    return id;
}
void Animal::speak() const
{
    cout << "What a zoo\n";
}
```

virtual Functions

- Each class derived from `Animal` will use the `getID` method as a default but will override the `speak` method. Two such classes, `Dog` and `Lion` are shown next.

7-21.cpp (cont.)

```
1. class Dog : public Animal
2. {
3. public:
4.     Dog(string);
5.     virtual void speak() const;
6. };
7. Dog::Dog(string n)
8. : Animal(n)
9. {}
10. void Dog::speak() const
11. {
12.     cout << "Name: " << name << ": " << "Bow Wow\n";
13. }
14.
15. class Lion : public Animal
16. {
17. public:
18.     Lion(string);
19.     virtual void speak() const;
20. };
21. Lion::Lion(string n)
22. : Animal(n) {}
23. void Lion::speak() const
24. {
25.     cout << "Name: " << name << ": "
26.     << "Roar\n";
27. }
```

- Finally we need a class which can collect many animals. We call this class `Collection`.

virtual Functions

7-21.cpp (cont.)

```
28. class Collection
29. {
30.     int capacity;
31.     int howmany;
32.     Animal **p;
33. public:
34.     Collection(int number = 10);
35.     void addAnimal(Animal & pt);
36.     Animal * operator[](int pos);
37.     int size();
38. };
39. Collection::Collection(int number = 10)
40. {
41.     capacity = number;
42.     p = new Animal *[capacity];
43.     howmany = 0;
44. }
45. void Collection::addAnimal(Animal & pt)
46. {
47.     if ( howmany < capacity )
48.         p[howmany++] = &pt;
49.     else
50.         cout << "reject " << pt.getID() << "\n";
51. }
52. Animal * Collection::operator[](int pos)
53. {
54.     if ( pos >= 0 && pos < howmany )
55.         return p[pos];
56.     else {
57.         cout << "Bad value: " << pos << "\n";
58.         return 0;
59.     }
60. }
61. int Collection::size()
62. {
63.     return howmany;
64. }
```

virtual Functions

- The `AddElement` method can take any `Animal` because the parameter is a reference to an `Animal`. It places the address of the `Animal` into an array of `Animal` pointers
 - ▶ `operator[]` returns an animal pointer.
- Here is the application file. It simply adds animals to the `zoo`, a `Collection` object, and then calls the `speak` method for each one.

7-21.cpp (cont.)

```
1. int main()
2. {
3.     Collection zoo(10);
4.     Animal Annie("mal");
5.     Dog zip("zippy");
6.     Dog chip("chippy");
7.     Lion leo("leo");
8.     zoo.addAnimal(zip);
9.     zoo.addAnimal(chip);
10.    zoo.addAnimal(leo);
11.    zoo.addAnimal(Annie);
12.    for (int i = 0; i < zoo.size(); i++)
13.        zoo[i] -> speak();
14.    return 0;
15. }
```

- The dramatic part of this code is the last loop. The compiler is smart enough to call the correct `speak` method based on the type of pointer returned from the `operator[]` method. This developer is freed from the burden of determining the type of pointer.

Polymorphism

- `virtual` functions together with pointers or references provide a powerful C++ feature.
- Functions whose tasks are "similar but different" should be implemented in this way. This concept is called `polymorphism`. It is the single most powerful idea in object-oriented programming.
- The previous example demonstrated where the power lies. The animal classes represented a typical scenario.
 - ▶ A group of objects of various subtypes are placed in a collection.
 - ▶ The collection defines a way of selecting each item in it.
 - ▶ The selection mechanism returns a pointer (or a reference) which then calls the correct method from a family of them without any programmer intervention.

Evaluation
Copy

virtual Destructors

- Recall that a constructor is automatically executed during object creation.

```
void fun1()
{
    Dog zip("zippy");
}
```

- The destructor will automatically be called for local objects, such as `zip`.

- For heap based objects, `delete` must be called explicitly.

```
void fun2()
{
    Dog *zip = new Dog("zippy");
    ...
    ...
    delete zip;
}
```

- But which destructor gets called in the following situation.

```
void fun3()
{
    Animal *zip = new Dog("zippy");
    ...
    delete zip;
}
```

- The answer depends upon whether the destructor is `virtual` or not. If it is, then the `Dog` destructor gets called. Once a destructor gets called in a hierarchy, all destructors up the hierarchy will get called as well.

Pure virtual Functions

- In the `Animal` hierarchy, it's clear that a `Dog` and a `Lion` represent concrete entities.
- On the other hand, `Animal` is really an abstraction. You have probably seen many `Dogs` and a few `Lions`, but have you ever seen an `Animal`?
- Classes like `Animal` should be used to specify prototype information for methods that should be implemented in derived classes.
- These methods are not intended to be implemented in the `Animal` class. Rather they are intended as a mandate to be implemented in derived classes.
- When they are implemented in derived classes, they must have the same prototype as their model in the `Animal` class.

Pure virtual Functions

- In C++, the following notation is used for methods that are not intended to be implemented.

```
#include <iostream>
#include <string>
using namespace std;
class Animal
{
    static int ids;
protected:
    int id;
    string name;
public:
    Animal(string);
    int getID() const;
    virtual void speak() const = 0;
    virtual void eat() const = 0;
    virtual void move() const = 0;
};
int Animal::ids = 1;
```

- Methods such as `speak`, `eat`, and `move` are called **pure virtual** functions. The following notation informs the compiler that these methods will not be implemented in this class but must be implemented in derived classes.

```
virtual void speak() const = 0;
virtual void eat() const = 0;
virtual void move() const = 0;
```

Abstract Base Classes

- Any class having one or more pure virtual functions is called an **abstract base class**. This underlies the fact that it is not a real class but an abstraction.
- Since it does not represent a real class, there can be no objects of this type.

```
Animal a;           // error
```

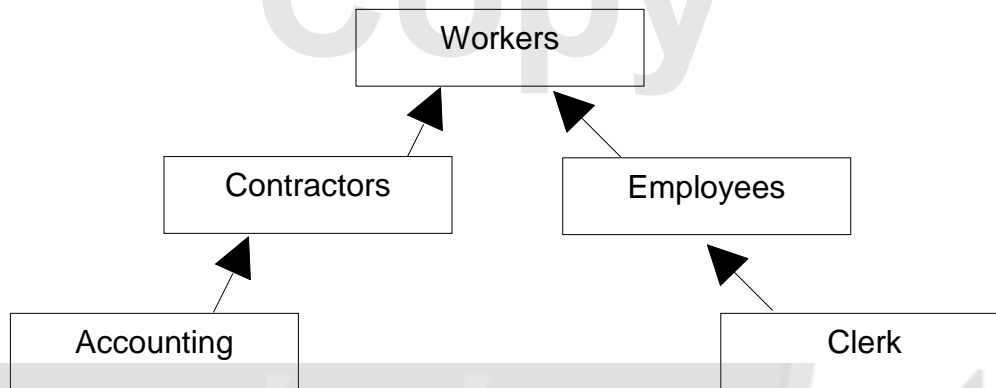
- Its raison d'être is to allow derived classes to inherit a set of interfaces.
- However, there can be pointers to `Animal`.

```
Animal *pt;
```

- The animal pointer can contain the address of any derived object, such as a `Dog` or a `Lion`.
- Abstract base classes make no sense by themselves. They make sense only as a basis for derived classes.
- An abstract base class can be viewed as an interface which hides implementation details.
- An abstract base class can have:
 - ▶ "real" functions
 - ▶ data
 - ▶ constructors, destructor

An Extended Inheritance Example

- We now present a series of classes that descend from an abstract class named `Workers`. First we show a hierarchy of these classes.



`Workers.h`

```
1. // Workers.h
2. //
3. class Workers
4. {
5.     static int ids;
6.     int id;
7.     string name;
8. public:
9.     Workers(string);
10.    int getId();
11.    string getName();
12.    virtual double pay() = 0;
13.    virtual void vacation() = 0;
14. };
```

An Extended Inheritance Example

- Now notice the following two classes, Contractors and Employees.

Contractors.h

```
1. // Contractors.h
2. //
3. #include "Workers.h"
4. class Contractors : public Workers
5. {
6. public:
7.     virtual void review( ) = 0;
8. };
9. // Employees
10. //
11. #include "Workers.h"
12. class Employees : public Workers {
13. public:
14.     virtual double pension( ) = 0;
15.     virtual void bonus( ) = 0;
16. };
```

An Extended Inheritance Example

- Employees and Contractors do not implement any methods from their base class and thus they too are abstract classes.
- Classes derived from Employees must now implement:

```
virtual double pay()           = 0;  
virtual void vacation()       = 0;  
virtual double pension()      = 0;  
virtual void bonus()          = 0;
```

- Classes derived from Contractors must implement:

```
virtual double pay()           = 0;  
virtual void vacation()       = 0;  
virtual void review()          = 0;
```

An Extended Inheritance Example

- Here's an example class that derives from `Contractors`. You will be asked to fill in the details in an exercise.

```
//  
// Accounting.h  
//  
#include "Contractors.h"  
class Accounting : public Contractors  
{  
};
```

- Here's an example class that derives from `Employees`. You will be asked to fill in the details in an exercise.

```
//  
//Clerk.h  
//  
#include "Employees.h"  
class Clerk : public Employees  
{  
};
```

Exercises

1. Implement a `Point` class and then derive `Point3D` from `Point`. Make sure the following program can compile and execute.

```
#include "Point.h"
#include "Point3D.h"
#include <iostream>
using namespace std;
int main()
{
    Point    p1(0,1);
    Point3D  p2(2,3,4);
    cout << p1.getX() << "," << p1.getY() << "\n";
    cout << p2.getX() << ","
        << p2.getY() << ","
        << p2.getZ() << "\n"    ;
    p1.print();
    cout << "\n";
    p2.print();
    cout << "\n";
    return 0;
}
```

Evaluation
Copy

Exercises

2. Create an abstract Shape class that serves as a design specification for some concrete classes such as Circle and Square. Shape should be defined as follows where MyString is a class that you have previously built.

```
class Shape
{
protected:
    MyString shapename;
public:
    Shape(MyString s);
    virtual ~Shape();
    virtual double perimeter() = 0;
    virtual double area() = 0;
    const char *getname();
};
```

```
class Circle : public Shape
{
    double radius;
public:
    Circle(double r, MyString s);
    ~Circle();
    virtual double perimeter();
    virtual double area();
};
```

```
class Square : public Shape
{
    double side;
public:
    Square(double side, MyString s);
    ~Square();
    virtual double perimeter();
    virtual double area();
};
```

Exercises

- ▶ Now create a few `Circle` objects and a few `Square` objects and store their addresses in an array of `Shape` pointers as shown below.

```
const int SIZE = 4;
Shape *pt[SIZE];
Circle  c1(5.0, "mycircle");
Square  s1(10.5, "mysquare");
pt[0] = &s1;
pt[1] = &c1;
...
```

- ▶ Finally, write several loops, one for each function. The loop for the area function is shown below:

```
for (int i = 0; i < SIZE; i++)
    cout << pt[i] -> area() << "\n";
```

3. Complete the class hierarchy below by creating the concrete classes. Add functionality to these concrete classes as you see fit. Make sure that the minimum functionality for these classes satisfies the mandate implied by the three abstract classes below. See the starters directory.

```
// Abstract classes
//
class Workers {};
class Employees : public Workers {};
class Contractors : public Workers {};
//
// Concrete classes
//
class Sales : public Employees {};
class Marketing : public Employees {};
class Legal : public Contractors {};
class Accounting : public Contractors {}
```