
Lecture 6: Inheritance

Dragan Mirkovic
Department of Computer Science
University of Houston

D. Mirkovic, C++ Programming, Spring 2005

Announcements

- Today:
 - Inheritance and derived classes in OOL
 - Ch 6. In Irvine.
 - Quiz #3, Tuesday 4/5/2005
 - Designing classes, Ch. 5 in text.
 - Project #2 Designing classes
 - Problem 5.7.3 pp. 174.

D. Mirkovic, C++ Programming, Spring 2005

Introduction

- Inheritance is one of the important concepts in OOL
- Simplifies construction of similar classes
- Topics
 - Creation of hierarchies
 - Constructors and destructors of classes related by inheritance
 - Single inheritance
 - Multiple inheritance
 - Examples

D. Mirkovic, C++ Programming, Spring 2005

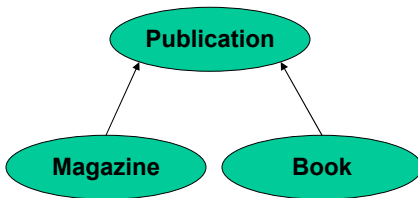
Single inheritance

- Definition:
 - Inheritance is an is-a relationship between classes
 - New class (called *derived class*) is derived from an existing class
 - Derived class includes all data members and operations of the *base class*
 - Extension of capabilities of an existing class
 - Derived class includes new data members and operations

D. Mirkovic, C++ Programming, Spring 2005

Example

- Base class:
 - Publication
- Derived classes:
 - Magazine
 - Book



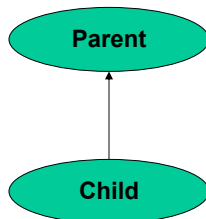
```
• In C++:
class Publication {
public:
    void SetPublisher(const char *s);
    void SetDate(unsigned long dt);
private:
    FString publisher;
    unsigned long date;
};

class Magazine :public Publication {
public:
    void SetIssuesPerYear (unsigned n);
    void setCirculation(unsigned long n);
private:
    unsigned issuesPerYear;
    Unsigned long circulation;
};
```

D. Mirkovic, C++ Programming, Spring 2005

Classifying Objects

- In their attempts to model the real world, designers classify objects. Inheritance implies a hierarchical relationship.

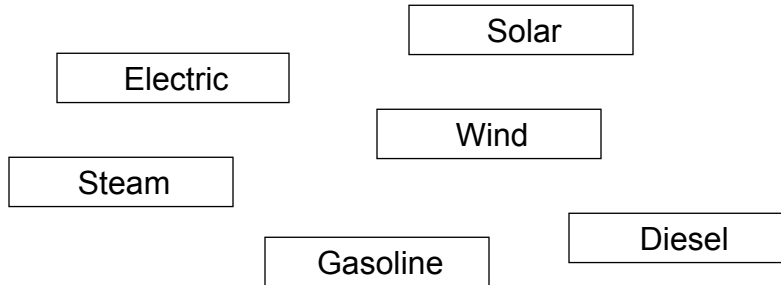


As in biology, the child inherits characteristics from the parent

D. Mirkovic, C++ Programming, Spring 2005

Types of Motors

- Clearly, all these classes are related. What might all of these motors have in common?



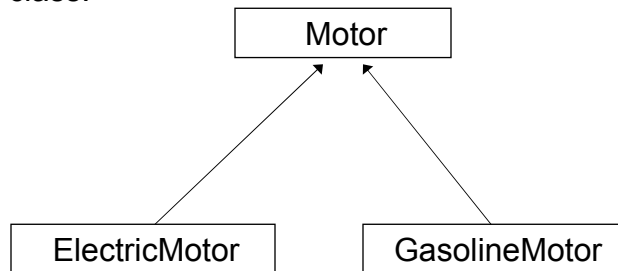
Motors

- The common information can be described in a base class named Motor:

Motor class:
Serial number
Manufacturer
Date of Manufacture
Model Number
etc.

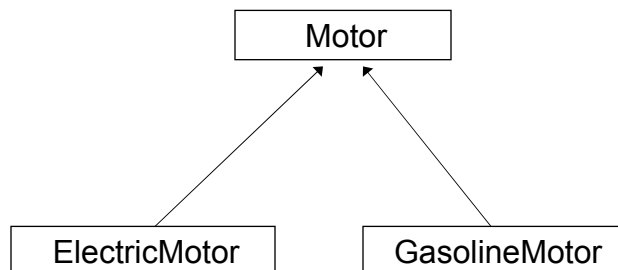
Inheritance Relationships

- The arrows show that the ElectricMotor and GasolineMotor classes are derived from the Motor class.



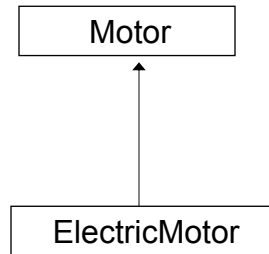
Inheritance Relationships...

- The Motor class is called the base class, and the other two are called derived classes.



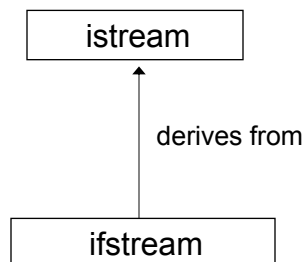
Inheritance (Is-A)

- Inheritance is often called an *is-a* relationship. In other words, "an ElectricMotor is a Motor."



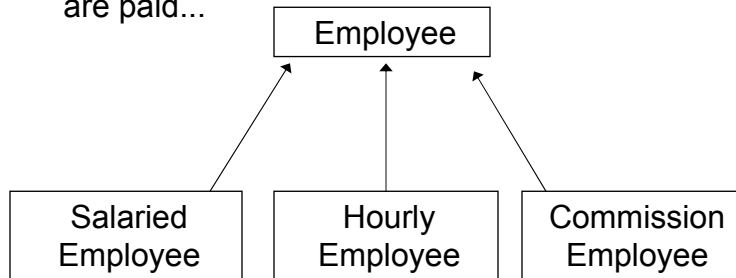
Inheritance Relationships...

- Many other classes also have this relationship, such as the istream and ifstream classes.



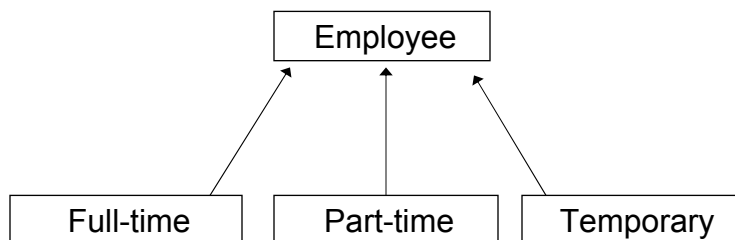
Example: Employees

- We could categorize employees based on how they are paid...



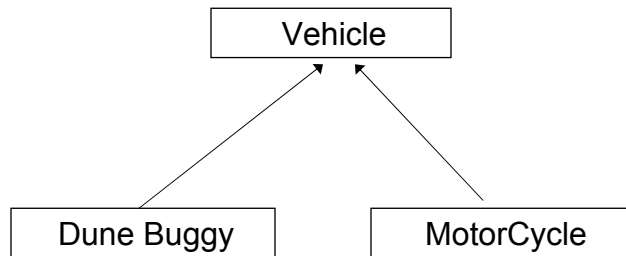
Types of Employees

- ...or we might categorize based on their employment status:



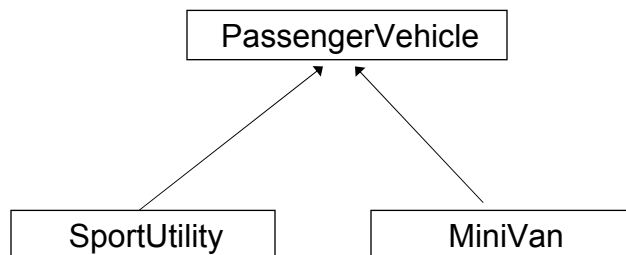
Inheritance Relationships...

- Some relationships are rather remote....

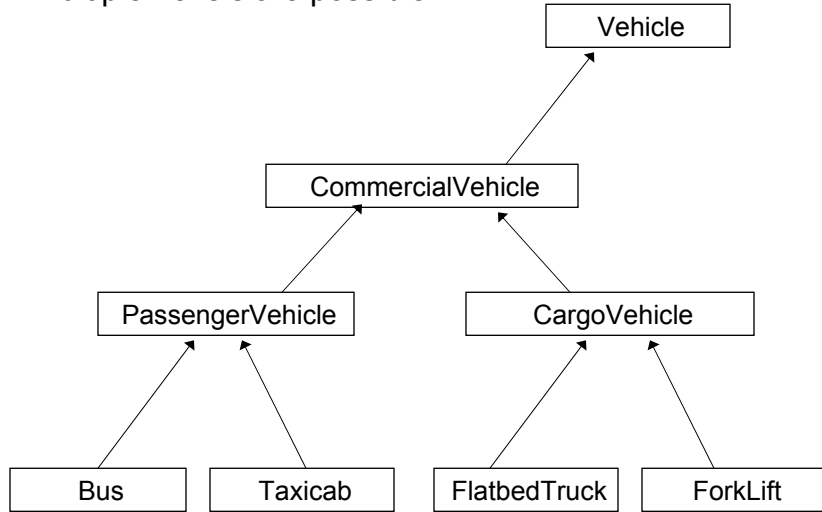


Inheritance Relationships...

- Others are very specific:

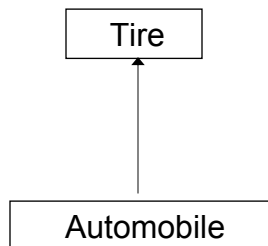


Multiple Levels are possible



Other Relationships

- Here's a relationship that is not based on inheritance. What would you call it?



- This is a *has-a* relationship, also called aggregation.

Declaring Derived Classes

- The general form:

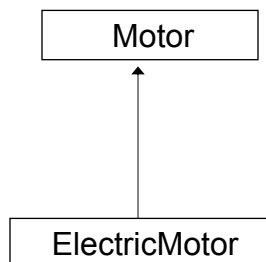
```
class class-name : access-specifieropt base-class {  
    member-list  
};
```

- *access-specifier*: public, private, protected
- *base-class*: the name of the class from which the current class is derived.
- *member-list*: data and function members.
- Private data members in the base class can only be accessed by functions in the base class and designated friends.

D. Mirkovic, C++ Programming, Spring 2005

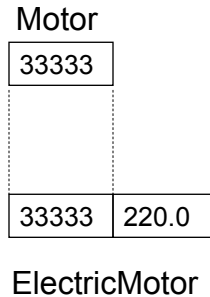
Motor and ElectricMotor

- Let's implement two classes that have something in common.



Motor and ElectricMotor

- An ElectricMotor object contains the same ID number as a Motor, plus the voltage.



CMotor Class

- The CMotor class definition:

```
class CMotor {
public:
    CMotor() { }
    CMotor( const string & id );

    string get_ID() const;
    void set_ID(const string & s);

    void Display() const;
    void Input();

private:
    string m_sID;
}; // more...
```

```

CMotor::CMotor( const string & id )
{ set_ID(id); }

string CMotor::get_ID() const
{ return m_sID; }

void CMotor::set_ID(const string & s)
{ m_sID = s; }

void CMotor::Display() const
{ cout << "[CMotor] ID=" << m_sID; }

void CMotor::Input()
{
    string temp;
    cout << "Enter Motor ID: ";
    cin >> temp;
    set_ID(temp);
}

```

Creating a Derived Class

- The base class must appear first. When the derived class is declared, it names the base class.

```

class base {
    ...
};

class derived : public base1, base2, ...
{
    ...
};

```

- Any number of classes can be derived from the same base class.

CElectricMotor Class

```
class CElectricMotor : public CMotor {
public:
    CElectricMotor();
    CElectricMotor(const string & id,
                  double volts);

    void Display() const;
    void Input();
    void set_Voltage(double volts);
    double get_Voltage() const;

private:
    double m_nVoltage;
};
```

Base Class Initializer

- A base class initializer calls the constructor in the base class. In this example, the ID number of the motor is passed to the CMotor constructor.

```
CElectricMotor::CElectricMotor(const string & id,
                               double volts) : CMotor(id)
{
    m_nVoltage = volts;
}
```

Calling Base Functions

- The Input function exists in both the CMotor and CElectricMotor classes. Rather than duplicate code already written, call the corresponding function in the base class:

```
void CElectricMotor::Input()
{
    CMotor::Input(); // call base class first

    double volts;
    cout << "Voltage: ";
    cin >> volts;
    set_Voltage(volts);
}
```

Calling Base Functions

- This is the Input function in the CMotor class:

```
void CMotor::Input()
{
    string temp;
    cout << "Enter Motor ID: ";
    cin >> temp;
    set_ID(temp);
}
```

Display Function

- The Display function works the same way. It calls `CMotor::Display` first.

```
void CElectricMotor::Display() const
{
    // call base class function first
    CMotor::Display();

    cout << " [CElectricMotor]"
         << " Voltage=" << m_nVoltage <<
    endl;
}
```

Testing the Classes

- The test program can create instances of both `CMotor` and `CElectricMotor` objects.

```
CMotor mot("33333");
mot.Display();
cout << endl;

CElectricMotor elec("40000",220.0);
elec.Display();
cout << endl;
```

Testing the Classes

- When using a derived object, functions may be called from both the base class and the derived class.

```
CElectricMotor elec;           // CElectricMotor  
  
elec.set_ID("40000");         // CMotor  
  
elec.set_Voltage(220.0);     // CElectricMotor
```

Testing the Classes

- When the same function name exists in both classes, C++ automatically calls the function in the derived class. This is the case with the Input and Display functions:

```
elec.Input();                 // CElectricMotor  
  
elec.Display();              // CElectricMotor
```

Assigning Objects

- You can assign a derived class object to a base class variable. This is called object slicing.

```
CMotor mot;
CElectricMotor elec;

mot = elec;          // sliced down to a motor
elec.get_Voltage(); // ok
mot.get_Voltage();  // error
```

Assigning Objects

- But you cannot assign a base class object to a derived variable. That would permit references to nonexistent class members.

```
CMotor mot;
CElectricMotor elec;

elec = mot;          // error

elec.set_Voltage( 220 ); // ???
```

Protected Access

- Class members designated as protected are visible to both the current class, as well as all derived classes. (*but not to anyone else*)
- Two typical cases:
 - Direct access for the derived class to data members of the base class that would otherwise be private
 - Restrict access to base class members that would otherwise be public

Protected Access: Expansion

- Here is one example of extension of access:

```
class Point {  
    public:  
        //  
    protected:  
        int x;  
        int y;  
};
```
- Now derived classes can have direct access to **x** and **y**
 - Not always a good idea, but necessary in some situations

Protected Access: Restriction

- Here's an example that uses the protected qualifier to limit the visibility of `get_ID` and `set_ID`:

```
class CMotor {
public:
    CMotor() { }
    CMotor( const string & id );

protected:
    string get_ID() const;
    void set_ID(const string & s);
```

Protected Access

- The main program cannot call `set_ID` and `get_ID` because they are protected:

```
CMotor M;

M.set_ID("12345");           // error

M.get_ID();                  // error
```

Protected Access

- But functions in CElectricMotor still can access set_ID:

```
CElectricMotor::CElectricMotor(  
    const string & id, double volts)  
{  
    m_nVoltage = volts;  
    set_ID(id);  
}
```

Protected Inheritance

- Let's assume for the moment that CMotor still uses public for all member functions:

```
class CMotor {  
public:  
    CMotor() { }  
    CMotor( const string & id );  
  
    string get_ID() const;  
    void set_ID(const string & s);  
  
    //...
```

Protected Inheritance

- We can use the `protected` specifier when creating a derived class.
- All public functions in the base class become protected in the derived class.

```
class CElectricMotor : protected CMotor
{
    //...
};
```

Protected Inheritance

- For example, the main program cannot call `set_ID` and `get_ID` on an electric motor because the functions are no longer public:

```
CElectricMotor EM;

EM.set_ID("12345");           // error

EM.get_ID();                  // error
```

Protected Inheritance

- It might be that the CElectricMotor class author does not want users to know about the motor's ID number.
- Functions in CElectricMotor still can access public functions in CMotor. Here is an example:

```
CElectricMotor::CElectricMotor(  
    const string & id, double volts)  
{  
    m_nVoltage = volts;  
    set_ID(id);  
}
```

Private Inheritance

- Private inheritance causes all functions declared in the base class to be private when a derived class object is created.
- At first, there seems to be no difference from protected inheritance: Functions in CElectricMotor can still access member functions in CMotor...

```
class CElectricMotor : private CMotor {  
  
    //...  
  
};
```

Private Inheritance

- But when we derive a new class (CPumpMotor) from CElectricMotor, the difference shows: CPumpMotor functions cannot access public members of CMotor.

```
class CPumpMotor : public CElectricMotor {
public:
    void Display() {
        CMotor::Display();           // not
        accessible!
        CElectricMotor::Display();   // this is
        OK
    }
};
```

Nested Class Scope

- Member functions have direct access to all public and protected data and function members within the current class
- Public and protected base class members can be hidden by the same name in the derived class
 - Override by using the scope resolution operator (::)
- Example:

```
class Parent { public: void Print() const;};
class Child {
public:
    void Print() const{
        Parent::Print();
        cout << age << '\n' << school << endl;
    }
private: int age; FString school;
};
```

Friends in Derived Classes

- Friend function can access all public, private and protected members of a class
- Friend of a base class is not automatically a friend to its derived classes
- Example:

```
- Employee and SalariedEmployee
class Employee {
public:
    friend float CalcPay ( Employee & E );
    //...
};
```

D. Mirkovic, C++ Programming, Spring 2005

Example (cont.)

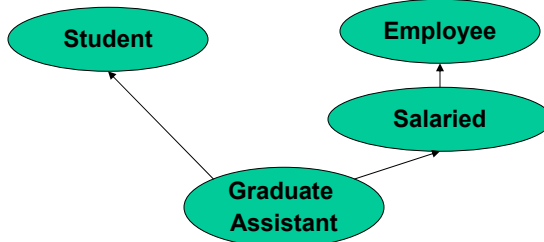
- **SalariedEmployee** requires its own **CalcPay** function, which is very possibly different from the one in the base class
- Example:

```
class SalariedEmployee : public Employee {
public:
    friend float CalcPay ( SalariedEmployee & E );
    //...
};
```

D. Mirkovic, C++ Programming, Spring 2005

Multiple Inheritance

- A derived class may contain attributes and properties inherited from two or more base classes
 - Multiple inheritance
- Example: graduate assistant in a university is both a student and a salaried employee



D. Mirkovic, C++ Programming, Spring 2005

Example

- Use base class list for the `GradAssistant` declaration:

```
class GradAssistant : public Student, public Salaried
```
- Requires name resolution if both base classes contain functions with the same name, like for example, `SetAge` (see. text)

```
GradAssistant GA.  
GA.SetAge(22);           //error ambiguous  
GA.Student::SetAge(22) // ok.
```

D. Mirkovic, C++ Programming, Spring 2005

Summary

- Single inheritance
- Derived classes, base class
- Inheritance tree
- Multiple inheritance
- Private and protected inheritance
- Constructor-initializers

D. Mirkovic, C++ Programming, Spring 2005

Homework

- Exercises pp 212
- 6.6.1 - 1 Time Zones
- 6.6.4 Graphics Classes

D. Mirkovic, C++ Programming, Spring 2005