Inheritance

• If you define a class based on a previously defined class, the derived (or descendant) class is said to inherit many properties of the base (or ancestor) class.

• Inheritance describes the ability of a class to inherit information from ancestor (super-) classes.

• A class can be based on more than one base class, creating multiple inheritance. We will avoid it in this course. It is complex and can lead to confusion if not used properly. (Java does not support multiple inheritance.)
Why Inheritance?

• Inheritance enables the reuse of existing classes.
• Inheritance reduces the effort needed to add new features to an existing object.
• For example, we can have a class called Person and then create a new class called Student and yet another class called Faculty both of which are derived from Person. Features that are common to both classes are placed in Person, reducing the need for duplication of information in classes.
• An object is considered a member of its own class, and also of all its super classes.
What is inherited?

• All members of the base class (methods and fields) are inherited except the constructors and destructors.
• The derived class can add new members.
• You cannot revise ancestral data members (e.g., change the data type of an inherited field.)
• You can redefine member functions to have a different meaning in a derived class.
  – This is origin of polymorphism
• When an object is created, remember that we need to initialize all variables and this will involve the constructor functions for ancestral classes.
Inheritance vs Access?

- The derived class inherits the private variables of the base class but cannot access them directly. Thus, we need to depend on constructor and member functions to deal with these variables.
- Inheritance does not mean accessibility.
- The member functions of a derived class can call public member functions of the base class.
- The member functions of a derived class cannot call private member functions of the base class.
- What if a function name, say Foo, is common to both classes? An object of a derived class will use the one in derived class and an object of the base class will use the one in base class.
  - Dynamic binding © Matt Evett, 1999
Example

class Sphere
{
private:
    float myRadius;
    Point myPosition;
}
class Ball: public Sphere
{
private:
    string myMaterial;
    float myElasticity;
}
Protected Members

- A class can have protected members in addition to public and private.
- protected members are not accessible to clients of the class.
- protected members are accessible to derived class.
- protected category helps to share information among the classes in a single path of the hierarchy without sharing that information with other clients.
- In general, one should place variables in private section. Place them in protected if it is essential to share them with derived classes.
Kinds of Inheritance

• Public Inheritance:
  – Both public and protected members of the base class retain their access status in the derived class.
  – This is the most common.

• Protected Inheritance:
  – Both public and protected members of the base class become protected members of the derived class.

• Private Inheritance:
  – Both public and protected members of the base class become private members of the derived class.

• Note that the derived class keeps the same access or more stringent access (not more liberal access)
Inheritance Syntax

class Ball: public Sphere
{
    :
}
class Ball: protected Sphere
{
    :
}
class Ball: private Sphere
{
    :
}
Data Initialization

• Because a derived class inherits all data members of its base class, we need to initialize them all.
• When an object of the derived class is created, we call the constructor function of the derived class.
• The constructor function of the derived class will automatically call the constructor function of the base class.
• Thus constructor functions are executed starting from the farthest ancestor to the derived class.
• Thus, a constructor function of a class is only responsible for initializing variables explicitly defined in that class not for those inherited.
Constructor Usage

Ball::Ball(string mat, float elas, float rad, float pos): Sphere(rad, pos), myMaterial(mat), myElasticity(elas)
{
    :
}

Sphere::Sphere(float radius, float pos): myRadius(radius), myPos(pos)
{
    :
}
Is-A Relationship

• If you find that two classes share “is-a” relationship, then it indicates that you can use public inheritance.
• An object of the derived class can be substituted for an object of the base class. (Object Type Compatibility).
• Thus, if a function expects a formal parameter to be of base type class, you can call it with derived class object.
• Object compatibility applies both to reference and value parameters.
• This leads to the issue of static vs dynamic (late) binding.
Has-A relationship

- A ball-point pen has a ball has its point. Wouldn’t want to use a ball whenever a pen is expected!
- Probably don’t want to use inheritance here.
- Make the ball a member of the pen class.
- has-a ==> containment
  - Class Pen { ... private Ball myBall; ... }
- When an object of type pen is created, an object of type ball is indirectly created and hence the constructor function for this should be called. This is done in the initializer list.
  - Pen::Pen(int size):Ball(size) { ... }
- Destructors are executed in reverse order.
  Implicitly, you don’t have to call them.
As-A relationship

• You can implement stack as a list.
• When there is “as-a” relationship, consider private inheritance.
• Both “has-a” and “as-a” relationships are possible when public inheritance is inappropriate.
• With private inheritance, public and protected members of the base class become private and so not accessible to any clients or derived classes.
  – Useful--You don’t want a client to be able to call List::Insert on your Stack objects! Only Stack::Push.
• “has-a” is the simplest to use and is preferable if it is sufficient for your needs.
Private Inheritance

• If class D privately inherits from class B, then we can say that “D is implemented in terms of B”.
• Object of type D cannot be substituted for objects of type B.
• Private Inheritance means nothing during design, only during implementation.
• Private inheritance means that implementation only is inherited; interface should be ignored.
  – Example: Suppose we implement stack in terms of list. Clearly an object of type Stack has a different interface from an object of type List. List is used only to implement Stack.

© Matt Evett, 1999
Protected Inheritance

- Inherited public methods and members become protected.
- Use with as-a implementations when the derived object should, itself, be inheritable.
- Continuing the Stack as-a List example, protected inheritance would be correct if the designer wanted to allow types of Stacks (i.e., subclasses).
  - Ex: suppose a SmartStack is a kind of stack that logs the number of pushes and pops. Stack is a protected inheritor of List. SmartStack is a public inheritor of Stack.
  - Clients of both Stack and SmartStack cannot use List methods.
  - Methods of both Stack and SmartStack can.
Early binding

• Assume that both Sphere and Ball classes have a function called Display:

```cpp
class Sphere{
    void Display(); ... 
};
class Ball:public Sphere{
    void Display(); ... 
}; // Same signature
```
Early binding (cont.)

• Consider the following example:

```cpp
Sphere s1(3); Ball b1(4);
Sphere *p = &s1; Sphere *q = &b1;
s1.Display(); // early binding, Sphere::Display
b1.Display(); // early binding, Ball::Display
p->Display(); // late binding?
q->Display(); // late binding?
```

• Because `p` and `q` are pointers to a `Sphere`, the last two calls will call `Display` in `Sphere`. Even the last one!!

• “Early” == “static”, “late” == “dynamic”
Virtual Functions and Late Binding

• But, if Sphere::Display is a virtual function, then the last line of the example will call the Ball::Display rather than Sphere::Display.
  
  class Sphere{
    virtual void Display(); … }
  
  class Ball:public Sphere{
    void Display(); … }; // Same signature

• Virtual keyword ==> Use the function in derived class if it has one.
  Sphere *p = &s1;   p->Display()

• Calls Sphere::Display because *p is an instance of Sphere, not Ball.
What is Late Binding?

• Thus, the determination of which function to call for a particular statement in the code is done at run time rather than during compiler time. This is known as “late” or “dynamic” binding.

• Dynamic binding is possible for functions declared as virtual.

• Thus a virtual function can be overridden in the derived class.

• The function Display is called “polymorphic”.
  – A polymorphic function has multiple meanings.

• If you want the function in derived class to override the one in base class, don’t forget virtual!
Omission of Virtual keyword

• Omission of *virtual* can have indirect effect.

```cpp
class Sphere{
public:
    double Area() const;
    virtual const Display();
};

class Ball: public Sphere{
public:
    double Area() const;...
};

Ball b(10);
b.Display();
```

• Ball inherits Sphere::Display.
• What happens if Sphere::Display calls area?
Omitting Virtual Keyword

Ball b(10);
b.Display();

- **Suppose Sphere::Display calls Area():**
  ```cpp
double Sphere::Display() { ...  
    x = Area(); ...  }  
  ```
- **Because Sphere::Area is not defined as virtual, the call to Area in Display will be interpreted by the compiler as Sphere::Area, regardless of the type (Sphere or Ball) of the invoking object (static binding).**
- **If we had defined Sphere::Area as virtual, then the Area() function call in Display will use late binding and will call correctly the Area in Ball class.**
Virtual Method Table

• The compiler creates a table for each class in which it keeps track of the pointer to the actual function to be called for each virtual function defined in the class.

• VMT is invisible to the programmer as it is automatically created by the compiler.

• A call to the constructor establishes this table.

• Thus, VMT enables late (or dynamic) binding.
**VMT in action**

Sphere* p = &s1; Sphere* q = &b1; p->Display(); q->Display();

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area</td>
<td>0x1A550</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Sphere VMT</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area</td>
<td>0x2CC18</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Ball VMT</td>
<td></td>
</tr>
</tbody>
</table>
Abstract Class

- When a class defines a virtual function, normally the code for the function is also provided.
- *Virtual* = use the base class’s implementation unless it is overridden in the derived class.
  - The base class virtual function code is the most general use of the function. The implementations in derived classes capture special cases.
- If a base class does not provide the implementation for a virtual function, every derived class *MUST* provide one. Such a function is called a *pure virtual function*.
- A class with at least one pure virtual function is called an *abstract* class. *No instantiation allowed.*
Abstract Class (cont.)

- Abstract classes are useful for enforcing the implementation of a specific interface by derived classes.

- Declare a pure virtual functions with "= 0"
  ```
  virtual int Display() = 0;
  ```
  - Without "=", the linker will report an error: undefined function.

- If a class has at least one virtual function, you should normally declare the destructor as a virtual function, too. (Highly recommended.)
  - Why?

- If a class has no virtual functions, don’t declare destructor function as virtual.
Ex: Pure Virtual Class

• Triangles and Squares are Shapes
• Shouldn’t be able to instantiate Shape, such an object would have no meaning.

Class Shape{
    public:
        virtual void Display() = 0;  // pure virtual
        // This forces derived classes to implement this.
        virtual ~Shape();  // All base classes should have virtual destructors.

Class Triangle: public Shape { ... };
Class Square: public Shape { ... };

© Matt Evett, 1999
Destructors and Inheritance

• What happens if we try to delete a derived class object using a base class pointer, but base::destructor is non-virtual?

class A{...; virtual ~A(); ...};
class B: public A {...; ~B(); ...extra fields};
class C: public B {...; ~C(); ...more extra fields};
B* x = new B;  B* y = new C;
delete x; delete y;
Inheritance & Destructors (cont.)

- The compiler will call the destructors starting only from the base class going up in the hierarchy.
  - `delete x;` // works just fine
  - `delete y;` // clears “extra fields”, but not “more extra fields”

- But, we always want to call destructors starting from the most specific class, in reverse order.
  - The solution is to make the destructor function virtual in base classes.

- One twist: If you want to make a class abstract but you have no pure virtual functions, what do you do? Make the destructor a pure virtual function. But, you must provide code for this destructor.