Announcement

- Lab #3
  - Problems 1 – 5 due 2/19 (Thursday) at 11:59PM.
  - Work on problems 6 – 9 this Friday and due 2/26 (Thursday) at 11:59PM.

- Quiz postponed to 2/26 during lecture

- Reading assignment
  - Ch. 11
Linked List

A → B → C → D → NULL

Linked list
Imagine you need an Object that is slightly different from the existing one.

Instead of re-designing an entire new object from scratch, you can inherit (or derive) the existing object and just “add” the needed modifications.

If Y inherits X, then
- Y is also X.
- Y contains all members of X, plus its own members.
Inheritance

- Class derived from another class
  - Base class (or parent class, or superclass)
  - Derived class (or child class, or subclass)
Inheritance

class derived_class_name: public base_class_name
{
    ...
};

class CRectangle: public CPolygon
{
    ...
};
Suppose class B inherits class A
The classes form a part of a class hierarchy.
  ◦ B is a derived class (subclass, child class) of A, class B inherits from class A.
  ◦ A is a base class (superclass, parent class) of B, class A derives class B.
  ◦ The class immediately above a given class is known as its immediate superclass.
A class inherits all members of the base class (with exceptions).
  ◦ Includes functions/variables inherited by that class.
  ◦ It can add additional variables and functions.
  ◦ It can override (change) the inherited functions.
What is inherited

- Derived classes inherit all accessible members of base class:
  - Base class \textit{A} has member \textit{a}.
  - Derived class \textit{B} has own member \textit{b}, and also member \textit{a}.

- Derived class DOES NOT inherit from base class:
  - Constructor and destructor
  - Operator=() members
  - Friend
Inheritance versus composition

- A class can include members of another class in two ways:

```
class Rect
{
  public :
    float xa, ya, xb, yb; // rectangle min/max coordinates
};

// 1) Example of composition:
class RoundedRectangle1
{
  public :
    Rect r; // rect is included as a member
    float cornerLen; // how much to round on each corner
};

// 2) Example of inheritance:
class RoundedRectangle2 : public Rect // members of Rect are inherited
{
  public :
    float cornerLen; // how much to round on each corner
};

//=> In both examples, the members of RoundedRectangleX are the same.
```
class Rect
{
    public :
    float xa, ya, xb, yb;
};

class RoundedRect : public Rect // 1) we are deriving with public
    //     access to Rect's members
{
    public :
    float cornerDist;
};

void main()
{
    RoundedRect r;
    r.xa = 1.0f; // 2) we have public access to the
                 //     members of the base class
    r.cornerDist = 0.2f; // 3) we also have public access to the
                          //     members of RoundedRect
}
Access control

class Rect
{
    float xa, ya, xb, yb; // 1) => these members are now private
};

class RoundedRectangle : public Rect // 2) we are still deriving with
    // public access
{
    public :
        float cornerDist;
};

void main ()
{
    RoundedRectangle r;
    r.xa = 1.0f; // 3) error: public derivation will not
        // break access control of the base class
    r. cornerDist = 0.2f;
}
Access control

class Rect
{
    protected:
        float xa, ya, xb, yb; // 1) these members are now protected
};

class RoundedRect : public Rect // 2) we are still deriving with
    // public access
{
    public:
        float cornerDist;
        void set ( float f ) { xa = f; } // 3) => set gives access to xa
    }

void main ()
{
    RoundedRect r;
    r.xa = 1.0f; // 4) error: public derivation will not
                   // break access control of the base class
    r.set ( 1.0f ); // 5) ok
    r. cornerDist = 0.2f;
}
class derived_class_name: **public** base_class_name {...}

public members of base are public in derived

class derived_class_name: **protected** base_class_name {...}

public members of base are protected in derived

class derived_class_name: **private** base_class_name {...}

public members of base are private in derived
class X
{
    private: int privx;
    protected: int protx;
    public : int publx;
};

class Y : private X // private derivation
{
    public :
        void privset ( int i ) { privx=i; } // 1) Error
        void protset ( int i ) { protx=i; } // 2) Ok
        void publset ( int i ) { publx=i; } // 3) Ok
};

void main ()
{
 Y y;
 y.privx=1; // 4) error Everything in X is private
 y.protx=2; // 5) error through Y now!
 y.publx=3; // 6) error
}
## Inheritance access matrix

<table>
<thead>
<tr>
<th>Access</th>
<th>Public</th>
<th>Protected</th>
<th>Private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same class member</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Derived class member</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Non-member</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
Constructors of Derived Classes

// Example of typical constructors in a class:
class Rect
{
    public:
        float x, y, w, h; // rectangle upper-left corner (x,y) and size (w,h)

    Rect () { x=y=w=h=0; } // 1) Default constructor declared in-line

    Rect ( const Rect& r ) // 2) Copy constructor, takes in an object of same type
        { x=r.x; y=r.y; w=r.w; h=r.h; }

    Rect ( float rx, float ry, float rw, float rh ) // 3) Another constructor
        { x=rx; y=ry; w=rw; h=rh; }
};
Constructors of Derived Classes

// Constructors in a derived class must call
// the correct constructors of the base class:
class RoundedRect : public Rect
{
  public :
    float cornerLen; // how much to round on each corner

  RoundedRect () { cornerLen=0; } // 1) Default constructor of base class
                          // automatically called

  RoundedRect ( const RoundedRect& r ) // 2) Copy Constructor declaration
    :Rect(r)                               // 3) Calling copy constructor of Rect
  { cornerLen=r.cornerLen; }

  RoundedRect ( float rx, float ry, float rw, float rh, float len )
    :Rect(rx,ry,rw,rh),                  // 4) Calling constructor of base class
      cornerLen(len)                    // 5) Calling float “pseudo-constructor”
  { }
};
Constructors of Derived Classes

- The parenthesis syntax for constructors can be used in several ways:

  ```
  // 1) Example of "pseudo-constructors":
  int i(100); // same as int i=100;
  int* ip = new int(47); // different than new int[47]!

  // 2) Default constructor of an object automatically called:
  Rect r; // no need to use ()

  // 3) Primitive types do not have default constructors!
  int i; // no initialization done here

  // 4) Object initialization will call the copy constructor:
  Rect a; // will call default constructor
  Rect b=a; // will call copy constructor, same as Rect b(a)
  Rect c(a); // will call copy constructor, same as Rect c=a
  ```
The constructor of a base class is always called before the constructor of its derived class.

The same rule applies to long chains of derivation:

```cpp
class A {
};

class B : public A {
};

class C : public B {
};

...
Casting an object type to the type of its base class (as a pointer or reference).

So that classes can work with objects of known behavior (methods), even if an object may actually be of a derived type.

```cpp
class A { };

class B : public A { };

void main ()
{
    B bobject;
    B* bpt = &bobject; // 1) get a pointer to bobject

    A* a = (A*) bpt; // 2) upcast bpt to a pointer to A: always ok since “B is also A”
}
```
Redefining versus overriding methods

- **Redefinition of Methods**
  - Methods with same name in a base and derived classes are disambiguated by the type of the object.

- **Overriding Methods**
  - The virtual keyword allows to call a descendant method even if the object being used is of the base class type.
  - Makes sense only when upcasting is used.

- **Polymorphism**
  - The use of virtual methods is the key concept behind polymorphism.
  - To be covered when we get to Chapter 15.
Redefining versus overriding methods

class A
{
    public:
        void method1 () { cout<<"A::method1()\n"; }
        virtual void method2 () { cout<<"A::method2()\n"; }
};

class B : public A
{
    public:
        void method1 () { cout<<"B::method1()\n"; } // 1) method redefined
        virtual void method2 () { cout<<"B::method2()\n"; } // 2) overridden!
};

void main ()
{
    B b;
    A* a = (A*) &b; // 3) upcast b to a pointer to A
    a->method1(); // 4) will print: "A::method1()"
a->method2(); // 5) will print: "B::method2()"!
}