Redefining Base Class Functions

- In C++, we use inheritance to create new classes from existing ones

- Derived classes
  - inherit data/functionality of the base class
  - can add their own data/functionality
Redefining Base Class Functions

- Box is a Rectangle with height...
- But the area of the box should be calculated differently from the Rectangle

We will redefine the getArea() function
Redefining Base Class Functions

- **Redefining** function: function in a derived class that has the **same name and parameter list** as a function in the base class.

- **Why:** to replace a base class function with different actions in derived class.

- **Not the same as overloading:**
  - with overloading, parameter lists must be **different**
  - with redefining, parameter lists must be **the same**

- Objects of **base class** use **base class** version of function; objects of **derived class** use **derived class** version of function.
Redefining Base Class Functions

```cpp
int main()
{
    Rectangle r(2,3); \ create a Rectangle object
    Box b(2,3,4); \ create a Box object

    cout << r.getArea() << endl; \ invoke Rectangle’s
    \ version of getArea()

    cout << b.getArea() << endl; \ invoke Box’s
    \ version of getArea()
}
```

- C++ decides what version of `getArea()` to call by looking at the object types
Redefining Base Class Functions

What if in the derived class you want to access the base class’s version of a redefined function?

double Box::getArea() const
{
    double area = 0;
    area += 2 * height * width;
    area += 2 * height * length;
    area += 2 * Rectangle::getArea();
    return area;
}

Use scope resolution operator to access parent class members that were redefined.
REDEFINING BASE CLASS FUNCTIONS

- Write a function in the base class that uses a redefined function:

```cpp
void Rectangle::printArea() const {
    cout << "The surface area of this object is ";
    cout << getArea() << endl;
}
```

If we invoke `printArea()` on an object of type `Box`, which version of `getArea()` will be invoked: `Rectangle`’s or `Box`’s?
THE PROBLEM OF STATIC BINDING

Write a function in the base class that uses a redefined function:

```cpp
void Rectangle::printArea() const
{
    cout << "The surface area of this object is ";
    cout << getArea() << endl;
}
```

Call it on a Box object:

```cpp
Box myBox(1,1,1);
myBox.printArea();
```

Compiler notices that `printArea()` is defined in the base class, and binds the function call to the base class – we call this static binding, and it is clearly a problem.
A derived class can be derived from another derived class.
CLASS HIERARCHIES

- Consider the following hierarchy:
CLASS HIERARCHIES

- What members are inherited by Wizard?
- What members are accessible to Wizard?

```cpp
class Character {
    public:
        Character(string, int);
        int getHealth();
        void setHealth(int);
        string getName();
        bool isDead();
        bool isFriendly();
    protected:
        string name;
        int health;
};

class Wizard : public Character
```
CLASS HIERARCHIES

- What members are inherited by Alchemist?
- What members are accessible to Alchemist?

```cpp
class Wizard : public Character {
public:
    Wizard(string, int, int);
    int getMana();
    void setMana(int);
    void castSpell(Character&);
    void heal(Character&);
protected:
    int mana;
};
```

class Alchemist: public Wizard
What happens to `castSpell()` function in the Alchemist class?

What happens to the `heal()` function?

class Alchemist : public Wizard
{
public:
    Alchemist(string, int, int);
    ~Alchemist();
    void makePotion();
    void castSpell(Character&); // ?
protected:
    void heal(Character&); // ?
    Potion * potions;
};