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.
The Problem of Static Binding

- **Static binding (early binding):** function calls are linked to function bodies at **compile time**, i.e. before the program is run.

- **Dynamic binding (late binding):** function pointers are used by the computer instead of hard-coding the link between function calls/bodies, thus we can substitute the necessary function at **run time**.

- How do you change static binding of functions to dynamic binding?
- By making the functions that require late binding **virtual**.
VIRTUAL FUNCTIONS

class Rectangle
{
    // put constructor and other functions here
    virtual double getArea() const;
    void printArea() const;
};

void Rectangle::printArea() const
{
    // because getArea() is virtual,
    // dynamic binding will take place
    cout << "The surface area of this object is ";
    cout << getArea() << endl;
}

- Now the correct version of getArea() will be invoked even on a Box object:

    Box myBox(1,1,1);
    myBox.printArea();
VIRTUAL MEMBER FUNCTIONS

- Virtual member function:
  - function in base class that expects to be redefined in derived class

- Function defined with keyword **virtual**:
  ```
  virtual void Y() {...}
  ```

- Supports **dynamic binding**: functions bound at run time to class-specific function call

- Without virtual member functions, C++ uses **static** (compile time) binding
**Virtual Functions**

- A virtual function is dynamically bound to calls at runtime.
- At runtime, C++ determines the type of object making the call, and binds the function to the appropriate version of the function.
- To make a function virtual, place the `virtual` keyword before the return type in the base class's declaration:

  ```cpp
  virtual double getArea() const;
  ```

- The compiler will not bind the function to calls. Instead, the program will **bind them at runtime**.
Redefining vs. Overriding

- In C++, redefined functions are statically bound and overridden functions are dynamically bound.

- So, a virtual function is overridden, and a non-virtual function is redefined.

- Redefining is a bad idea – you may run into static binding discrepancies any time

- Use overriding instead!
**POLYMORPHISM: “CHANGING SHAPES”**

- So we say that inheritance establishes “is a” relationship between classes...
  - A Student *is a* Person
  - A Box *is a* Rectangle

- How literal is it? In other words, can we say:
  - `Rectangle * someRectangle;`
  - `Box myBox(3,3,3);`
  - `someRectangle = &myBox;` //assign derived to base
  - `cout << someRectangle->getArea() << endl;`

- Will this work?
- What version of `getArea()` is going to be invoked?
Polymorphism Requires References or Pointers

- **Polymorphic behavior** is only possible when an object is referenced by a reference variable or a pointer.
- This will **not work**:
  - `Rectangle rec;`
  - `Box myBox(3,3,3);`
  - `rec = myBox; // compiles just fine!`
  - `cout << rec.getArea() << endl;`

- Will execute Rectangle’s `getArea()`.
- Known as slicing: we create a rectangle from a box, but the unique properties of box are lost.
BASE CLASS POINTERS

- Can define a pointer to a base class object
- Can assign it the address of a derived class object
- Example:
  - Rectangle * varRec = new Box(4,2,1);
- Correct version of getArea() will be called
  - NB: only if getArea() is virtual.
- Question: will you be able to call Box-only functions on varRec? (i.e., getVolume())
- No – only Rectangle interface is available to varRec
- Question: can you do this the other way around?
  - Box * varRec = new Rectangle(4,2);
- No – a Box is a Rectangle, but a Rectangle is not a Box (Polymorphism does not work in reverse)
TRUE BEAUTY OF POLYMORPHISM

- Remember the RPG hierarchy?
TRUE BEAUTY OF POLYMORPHISM

Now the Wizard can heal anyone at all who is a Character!

class Wizard :
public Character {
    public:
    Wizard(string, int, int);
    int getMana();
    void setMana(int);
    void castSpell(Character&);
    void heal(Character&);
    protected:
    int mana;
};
TRUE BEAUTY OF POLYMORPHISM

- Now the Wizard can heal anyone at all who is a Character!

```cpp
Wizard healer("Merlin", 100, 500);
Barbarian ghor("Ghor", 300);
Druid miriel("Miriel", 35, 400);

miriel.castSpell(ghor); // castSpell(Character&)
ghor.fightWith(miriel); // fightWith(Character&)

healer.heal(miriel);  // heal(Character&)
healer.heal(ghor);     // heal(Character&)
```

What if we replaced Character& with Character?
Destructing Polymorphic Objects

What if:

Character * healer = new Wizard("Merlin", 100, 500);
// Go on epic adventures;
// Eventually, delete the object:
delete healer;
// What destructor is this going to call?
// Character’s or Wizard’s?

A destructor is basically a special kind of function that is invoked automatically. Thus, destructors are susceptible to early binding, too.
VIRTUAL DESTRUCTORS

- It's a good idea to make destructors virtual if the class could ever become a base class.
- Otherwise, the compiler will perform static binding on the destructor
  - i.e., only invoke the base class constructor, ignoring derived classes

Why is it important to make destructors virtual:
- because derived classes may use dynamic memory!
- If you fail to invoke their destructors, you will run into a memory leak (see example code)