DigiPen INSTITUTE OF TECHNOLOGY

CS170#12.1

Polymorphism

Vadim Surov

Outline

- Abstract Classes
- Virtual Destructor

Abstract Classes

- The root class `Person` does not seem to have any compelling application in its own right
- The class serves a useful purpose within the logical specification of an inheritance hierarchy, but instances of it (arguably) do not
- An abstract class is simply a class that exists for high-level organizational purposes, but that cannot ever be instantiated

Abstract Classes (contd)

- In C++, a class is abstract if one or more of its member functions are pure virtual

- Pure virtual member functions remain pure virtual in derived classes that do not provide an implementation that overrides the base class prototype

Using The Abstract Class

- Assuming the revised declaration just given, the class `Person` can be derived from, but an attempt to declare an object of type `Person` will generate a compile-time error

- It is, however, legal to declare a pointer to an abstract class, and to use that pointer to store that address of a derived type (as long as it's not also abstract)

- Similarly, you can use references to an abstract class, but the target of the reference will always be some derived type

Virtual Destructor

- Question: What will be deleted in the following code?

```
Base *basePtr = new Derive();
delete basePtr;
```

- Answer: It depends on how the base class destructor is defined. Is it non-virtual or virtual?

- Destruction of a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor

Non-Virtual Destructor Example

```
class Base {
public:
  Base(){ cout << "Constructing Base"; }
  // this is a destructor:
  ~Base(){ cout << "Destroying Base"; }
};

class Derive: public Base {
public:
  Derive(){ cout << "Constructing Derive"; }
  ~Derive(){ cout << "Destroying Derive"; }
};

void main() {
  Base *basePtr = new Derive();
  delete basePtr;
}
```

```
Constructing      Base
Constructing    Derive
Destroying Base
```

Virtual Destructor Example

```
class Base {
public:
  Base(){ cout << "Constructing Base"; }
  // this is a virtual destructor:
  virtual ~Base(){ cout << "Destroying Base"; }
};

class Derive: public Base {
public:
  Derive(){ cout << "Constructing Derive"; }
  ~Derive(){ cout << "Destroying Derive"; }
};

void main() {
  Base *basePtr = new Derive();
  delete basePtr;
}
```

```
Constructing Base
Constructing Derive
Destroying Derive
Destroying Base
```

~~Next Lab~~

- ~~Create a simple "shape" hierarchy: a base class called `Shape` and derived classes called `Circle`, `Square`, and `Triangle`. In the base class, make a virtual function called `draw( )`, and override this in the derived classes.~~
- ~~Make an array of pointers to `Shape` objects that you set with addresses of dynamically created derived objects, and call `draw( )` through the base-class pointers, to verify the behavior of the virtual function.~~
- ~~When you destroy the array of shapes, set the base class destructor as virtual. What does it change in the destruction?~~

~~Next Lab~~

- ~~Modify above so `draw( )` is a pure virtual function. Try creating an object of type `Shape`. Try to call the pure virtual function inside the constructor and see what happens. Leaving it as a pure virtual, give `draw( )` a definition. What happened?~~