# Inheritance & Polymorphism

#### Week 7

#### Gaddis: Chapter 15

CS 5301 Spring 2017

Jill Seaman

## Inheritance

- A way to create a new class from an existing class
- The new class is a specialized version of the existing class
- Base class (or parent) the existing class
- Derived class (or child) inherits from the base class
- The derived class contains all the members from the base class (in addition to the ones in the derived class).

class Student {	<pre>class UnderGrad : public Student {</pre>
} Base class	} Derived class 2

## Access to private members

class Grade	class Test : public Grade
<pre>private members: char letter; float score; void calcGrade(); public members: void setScore(float); float getScore(); char getLetter();</pre>	<pre>private members: int numQuestions; float pointsEach; int numMissed; public members: Test(int, int);</pre>
When Test class inherits from Grade class using public class access, it looks like this:	<pre>private members: int numQuestions: float pointsEach; int numMissed; public members: Test(int, int); void setScore(float); float getScore(); float getLetter();</pre>
An instance of Test contains letter and score, but they are <b>not</b> directly accessible from inside (or outside) the Test member functions.	3

# Constructors and Destructors in Base and Derived Classes

- Derived classes can have their own constructors and destructors
- When an object of a derived class is created,
  - 1. the base class's (default) constructor is executed first,
  - 2. followed by the derived class's constructor
- When an object of a derived class is destroyed,
  - 1. the derived class destructor is called first,
  - 2. then the base class destructor

## **Constructors and Destructors:**

example

class BaseClass { public:	
BaseClass()	
	is the BaseClass constructor.\n"; }
~BaseClass()	, , ,
()	is the BaseClass destructor.\n"; }
};	
class DerivedClass : pu	blic BaseClass (
public:	
DerivedClass()	
	is the DerivedClass constructor.\n"; }
~DerivedClass()	is the belived class constructor. (ii ; ;
( )	is the DerivedClass destructor \r".
•	is the DerivedClass destructor.\n"; }
};	
int main() {	
	<pre>w define a DerivedClass object.\n";</pre>
DerivedClass object;	
	n is now going to end.\n";
}	
Output: W	e will now define a DerivedClass object.
	his is the BaseClass constructor.
т	his is the DerivedClass constructor.
T	he program is now going to end.
	his is the DerivedClass destructor

#### This is the DerivedClass destructor. This is the BaseClass destructor.

5

## Passing Arguments to a non-default Base Class Constructor

- Allows programmer to choose which base class constructor is called from the derived class constructor
- Specify arguments to base constructor in the derived constructor function header:

Square::Square(int side) : Rectangle(side, side)
{ // code for Square constr goes here, if any }

 You must specify a call to a base class constructor if base class has no default constructor

# **Redefining Base Class Functions**

- <u>Redefining function</u>: a function in a derived class that has the same name and parameter list as a function in the base class
- Not the same as overloading with overloading, parameter lists must be different
- Objects of base class use base class version of function; objects of derived class use derived class version of function.
- To call the base class version from the derived class version, you must prefix the name of the function with the base class name and the scope resolution operator:

Redefining Base Class Functions:

	1	
class Animal {		
private:		
string species;		
public:		
Animal() { species = "Animal";}	int main() {	
Animal(string spe)	Animal jaspe	er; // Animal()
{ species = spe ;}	Primate fred(	4); // Primate(int)
<pre>void display()</pre>	jasper.displa	ay(); cout << endl;
<pre>{cout &lt;&lt; "species " &lt;&lt; species; }</pre>	fred.display(	); cout << endl;
};	}	
class Primate: public Animal {		
private:		
int heartCham;		Output:
public:		species Animal
Primate() : Animal("Primate") { }		species Primate,
Primate(int in) : Animal ("Primate")		# of heart chambers 4
{ heartCham = in; }		
void display()		
{ Animal::display(); //calls base		
<pre>cout &lt;&lt; ", \n# of heart chambers " &lt;&lt; heartCham;</pre>		
}		8
};		ŭ

#### **Include Guards** #ifndef RECTANGLE H Rectangle.h #define RECTANGLE H class Rectangle private: double width; double length; public: void setWidth(double); void setLength(double); double getWidth() const; double getLength() const; double getArea() const; }; #endif

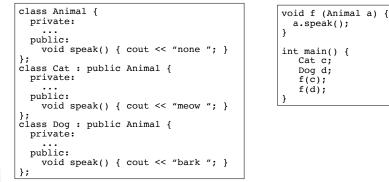
- These preprocessor directives prevent the header file from accidentally being included more than once.
- If you have a base class with 2 derived classes, and the derived classes are both included in a driver<sub>9</sub>...

# Polymorphism

- The Greek word poly means many, and the Greek word morphism means form.
- So, polymorphism means 'many forms'.
- In object-oriented programming (OOP), polymorphism refers to
  - identically named (and redefined) functions
  - that have different behavior depending on the (specific derived) type of object that they are called on.

10

# Example of polymorphism?



- IF the output is "meow bark", yes, polymorphism.
  - The behavior of a in f would depend on its specific (derived) type.
- IF the output is "none none", no it's not.

# Polymorphism in C++

- Polymorphism in C++ is supported through:
  - virtual functions AND
  - pointers to objects OR reference parameters.
- without these, C++ determines which function to invoke at <u>compile time</u> (using the variable type).
- when virtual functions and pointer/references are used together, C++ determines which function to invoke at <u>run time</u> (using the specific type of the instance currently referenced by the variable).

#### Virtual functions

- <u>Virtual member function</u>: function in a base class that expects to be redefined in derived class
- Function defined with key word virtual:

virtual void Y() {...}

- Supports <u>dynamic binding</u>: functions bound at run time to function that they call
- Without virtual member functions, C++ uses <u>static</u> (compile time) <u>binding</u>

# **Example virtual functions**

```
class Animal {
 public:
  virtual void speak();
 int age();
class Cat : public Animal
 public:
 virtual void speak(); //redefining a virtual
                       //redefining a normal function
 int age();
}:
int main()
  Cat morris;
 Animal *pA = &morris; //using a pointer to get dynamic binding
 pA -> age(); // Animal::age() is invoked (base) (not virtual)
 pA -> speak(); // Cat::speak() is invoked (derived)
                                                              14
```

Virtual functions

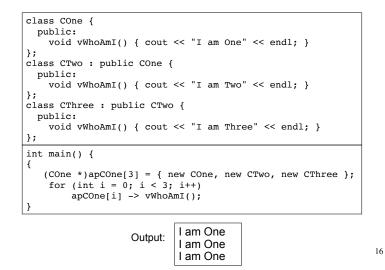
- In compile-time binding, <u>the data type of the</u> <u>pointer</u> resolves which function is invoked.
- In run-time binding, <u>the type of the object</u> <u>pointed to</u> resolves which function is invoked.

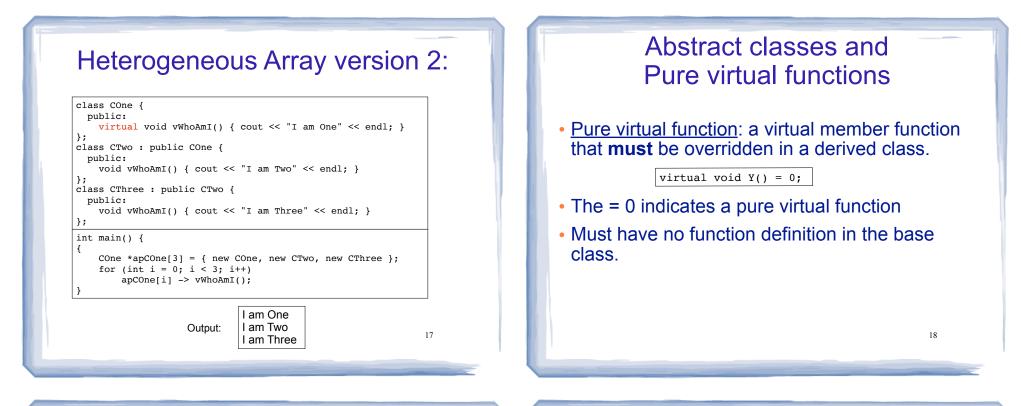
<pre>void f (Animal     a.speak(); }</pre>	<mark>&amp;</mark> a)	{
<pre>int main() {     Cat c;     Dog d;     f(c);     f(d); }</pre>		

• Assuming speak is virtual, since a is passed by reference, the output is:

15

# Heterogeneous Array version 1:

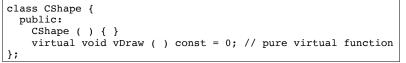




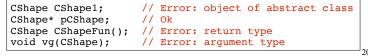
# Abstract classes and Pure virtual functions

- <u>Abstract base class</u>: a class that can have no objects (instances).
- Serves as a basis for derived classes that will have objects
- A class becomes an abstract base class when one or more of its member functions is a pure virtual function.

### Example: Abstract Class



- An abstract class may **not** be used as an argument type, as a function return type, or as the type of an explicit conversion.
- Pointers and references to an abstract class may be declared.



# **Example: Abstract Class**

• Pure virtual functions are inherited as pure virtual functions.

class CAbstractCircle : public CShape { private: int m\_iRadius; public: void vRotate (int) {} // CAbstractCircle ::vDraw() is a pure virtual function };

#### • Or else:

```
class CCircle : public CShape {
 private:
   int m_iRadius;
  public:
   void vRotate (int) {}
    void vDraw(); //define here or in impl file
};
```

21

# Heterogeneous collection: abstract base class

<pre>class Animal {     private:</pre>	<pre>int main() {</pre>
<pre>string name; public: Animal(string n) {name = n;}</pre>	<pre>Animal* animals[ ] = {     new Cat("Charlie"),     new Cat("Scamp"),</pre>
<pre>virtual void speak() = 0; };</pre>	new Cat( Scamp ), new Dog("Penny"), new Cat("Libby"),
class Cat : public Animal { public:	<pre>new Cat("Patches"), new Dog("Milo"),</pre>
<pre>Cat(string n) : Animal(n) { }; void speak() {cout &lt;&lt; "meow "; }</pre>	<pre>new Pig("Wilbur") }; for (int i=0; i&lt; 7; i++) {</pre>
<pre>}; class Dog : public Animal {    public:         Dog(string n) : Animal(n) { };</pre>	animals[i]->speak(); }
<pre>void speak() {cout &lt;&lt; "bark "; } }; class Pig : public Animal {    public:</pre>	Output: meow meow bark meow meow bark oink
<pre>Pig(string n) : Animal(n) { }; void speak() {cout &lt;&lt; "oink "; } };</pre>	22

22