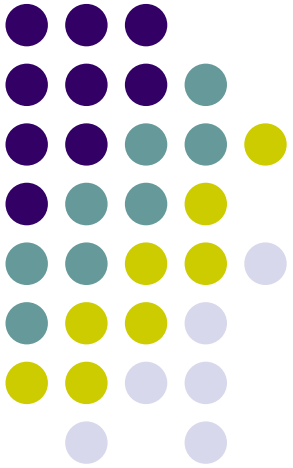
Polymorphism

Hugh C. Lauer
Adjunct Professor

(Slides include materials from *The C Programming Language*, 2nd edition, by Kernighan and Ritchie and from *C: How to Program*, 5th and 6th editions, by Deitel and Deitel)



Accessing Members of Base and Derived Classes

```
class B {
public:
   void m();
   void n();
   ...
} // class B

class D: public B {
public
   void m();
   void p();
   ...
} // class D
```

- The following are legal:–

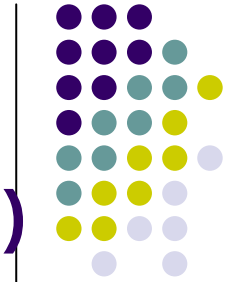
```
B_obj.m() //B's m()
B_obj.n()

D_obj.m() //D's m()
D_obj.n() //B's n()
D_obj.p()

B_ptr->m() //B's m()
B_ptr->n()

D_ptr->m() //D's m()
D_ptr->n() //B's n()
D_ptr->p()
```

Accessing Members of Base and Derived Classes (continued)



```
class B {
public:
   void m();
   void n();
   ...
} // class B

class D: public B {
public
   void m();
   void p();
   ...
} // class D
```

Class **D** *redefines* method **m()**

- The following are legal:–

  ```
  B_ptr = D_ptr;
  ```

- The following are *not* legal:–

  ```
  D_ptr = B_ptr;
  B_ptr->p();
  ```

  Even if **B_ptr** *is known to point to an object of class* **D**

Accessing Members of Base and Derived Classes (continued)

- Access to members of a class object is determined by the type of the *handle.*

- Definition: *Handle*
  - The thing by which the members of an object are accessed
  - May be
    - An object name (i.e., variable, etc.)
    - A reference to an object
    - A pointer to an object

Accessing Members of Base and Derived Classes (continued)

- This is referred to as *static binding*
- I.e., the *binding* between handles and members is determined at compile time
  - I.e., statically

What if we need Dynamic Binding?

- I.e., what if we need a class in which access to methods is determined at run time by the *type of the object*, not the *type of the handle*

```
class Shape {
public:
   void Rotate();
   void Draw();
   ...
}
```

```
class Rectangle: public
   Shape {
public:
   void Rotate();
   void Draw();
   ...
}
```

```
class Ellipse: public
   Shape {
public:
   void Rotate();
   void Draw();
   ...
}
```

What if we need Dynamic Binding?

- I.e., what if we need a class in which access to methods is determined at run time by the *type of the object*, not the *type of the handle*

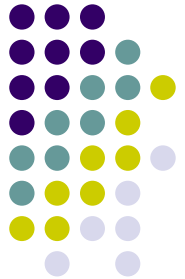
```
class Shape {
public:
   void Rotat
   void D
}
```

Need to access the method to draw the right kind of object Regardless of handle!

```
class Rectangle: public
   Shape {
public:
   void Rotate();
   void Draw();
   ...
}
```

```
class Ellipse: public
   Shape {
public:
   void Rotate();
   void Draw();
   ...
}
```

Solution – *Virtual Functions*



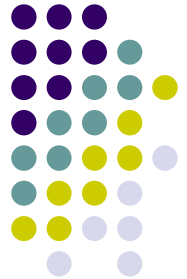
- Define a method as virtual, and the subclass method *overrides* the base class method

- E.g.,

```
class Shape {
public:
   virtual void Rotate();
   virtual void Draw();
   ...
}
```

This tells the compiler to add internal pointers to every object of class **Shape** and its derived classes, so that pointers to correct methods can be stored with each object.

What if we need Dynamic Binding?



```
class Shape {
public:
    virtual void Rotate();
    virtual void Draw();
    ...
}
```

- I.e., subclass methods *override* the base class methods
  - (if specified)
- C++ *dynamically* chooses the correct method for the class from which the object was instantiated.

```
class Rectangle: public Shape {
public:
    void Rotate();
    void Draw();
    ...
}

class Ellipse: public Shape {
public:
    void Rotate();
    void Draw();
    ...
}
```

Notes on `virtual`

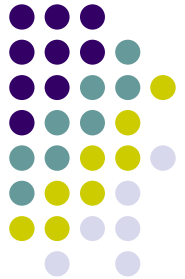
- If a method is declared virtual in a class,
  - … it is automatically virtual in *all* derived classes

- It is a really, really good idea to make destructors virtual!

```
virtual ~Shape();
```

  - *Reason:* to invoke the correct destructor, no matter how object is accessed

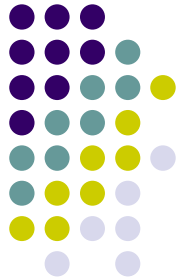
Notes on `virtual` (continued)



- A derived class may *optionally* override a virtual function
  - If not, base class method is used

```
class Shape {
public:
   virtual void Rotate();
   virtual void Draw();
   ...
}
class Line: public Shape {
public:
   void Rotate();
                //Use base class Draw method
   ...
}
```

Polymorphism



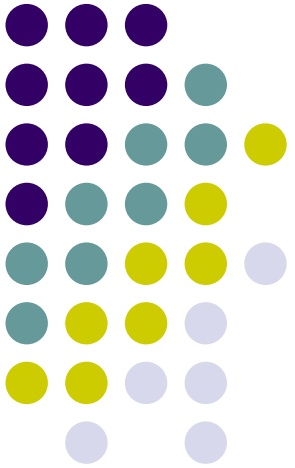
- The ability to declare functions/methods as **`virtual`** is one of the central elements of polymorphism in *C*++

- *Polymorphism:*– from the Greek “having multiple forms”
  - In programming languages, the ability to assign a different meaning or usage to something in different contexts

Summary – Based and Derived Class Pointers

- Base-class pointer pointing to base-class object
  - *Straightforward*
- Derived-class pointer pointing to derived-class object
  - *Straightforward*
- Base-class pointer pointing to derived-class object
  - Safe
  - Can access non-virtual methods of only base-class
  - Can access virtual methods of derived class
- Derived-class pointer pointing to base-class object
  - *Compilation error*

Questions?



Abstract and Concrete Classes

- *Abstract Classes*
  - Classes from which it is never intended to instantiate any objects
    - Incomplete—derived classes must define the “missing pieces”.
    - Too generic to define real objects.
  - Normally used as base classes and called *abstract base classes*
    - Provide appropriate base class frameworks from which other classes can inherit.
- *Concrete Classes*
  - Classes used to instantiate objects
  - Must provide implementation for *every* member function they define

Definitions

Pure `virtual` Functions

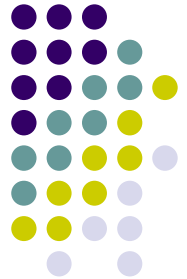
- A class is made *abstract* by declaring one or more of its virtual functions to be “pure”
  - I.e., by placing **`"= 0"`** in its declaration

- Example

```
virtual void draw() const = 0;
```

  - **`"= 0"`** is known as a *pure specifier*.
  - Tells compiler that there *is no* implementation.

Pure `virtual` Functions (continued)



- Every *concrete* derived class must override all base-class pure **`virtual`** functions
  - with concrete implementations

- If even one pure virtual function is not overridden, the derived-class will also be *abstract*
  - Compiler will refuse to create any objects of the class
  - Cannot call a constructor

Purpose

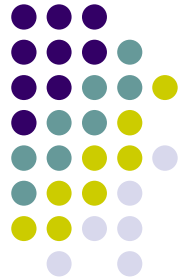
- When it does not make sense for base class to have an implementation of a function

- Software design requires *all* concrete derived classes to implement the function
  - Themselves

Why Do we Want to do This?

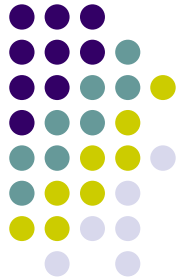
- To define a *common public interface* for the various classes in a class hierarchy
  - Create framework for abstractions defined in our software system
- The heart of *object-oriented programming*
- Simplifies a lot of big software systems
  - Enables code re-use in a major way
  - Readable, maintainable, adaptable code

Abstract Classes and Pure `virtual` Functions



- *Abstract* base class can be used to declare pointers and references referring to objects of any derived concrete class
- Pointers and references used to manipulate derived-class objects polymorphically
- Polymorphism is particularly effective for implementing layered software systems – e.g.,
  1. Reading or writing data from and to devices.
  2. *Iterator* classes to traverse all the objects in a container.

Example – Graphical User Interfaces



- All objects on the screen are represented by derived classes from an abstract base class
- Common windowing functions
  - Redraw or refresh
  - Highlight
  - Respond to mouse entry, mouse clicks, user actions, etc.
- Every object has its own implementation of these functions
  - Invoked polymorphically by windowing system

Questions?