Introduction to Polymorphism in Object-Oriented Programming

University of Windsor

March 12, 2020

Live Polling: pollev.com/curtisbright681
Roadmap

In this lecture we’ll design a simple object-oriented toolkit for displaying graphics in a terminal. Along the way we will cover:

- Encapsulation
  - Classes, Variables, Methods, Information hiding

- Inheritance
  - Superclasses, Subclasses, Substitution principle

- Polymorphism
  - Method overloading, Method overriding, Dynamic binding
Encapsulation

encapsulate (verb):

*to enclose in or as if in a capsule*
Encapsulation

encapsulate (verb):

to enclose in or as if in a capsule
There are different types or *classes* of objects. For example, “shape”, “triangle”, “box”, and “text box” are possible classes.
There are different types or *classes* of objects. For example, “shape”, “triangle”, “box”, and “text box” are possible classes.

In C++, a class may be defined using the `class` keyword:

```cpp
class Shape
{
};
```
Some properties of objects differ between two objects of the same class. For example, the heights and widths of a shape. These are *variables* of the shape.
Some properties of objects differ between two objects of the same class. For example, the heights and widths of a shape. These are variables of the shape.

class Shape
{
    int height, width;
};
Objects may also perform actions—such as updating their variables.
Methods

Objects may also perform actions—such as updating their variables.

```java
class Shape {
    int height, width;
    // Initialize height and width
    void init(int h, int w) {
        height = h;
        width = w;
    }
}
```
Information hiding

By default an object’s variables and methods can only be accessed from within the object’s methods.

Using the `public` keyword you can make variables and methods accessible from outside the object.
class Shape
{
    int height, width;
public:
    void init(int h, int w)
    {
        height = h;
        width = w;
    }
};
Consider the following `Account` class:

```cpp
class Account {
    int balance;    // Balance of account
public:
    int getBalance();    // Return balance of account
    void deposit(int amount);    // Add to balance
}
```

Assuming that `A` is of type `Account`, which of the following is a correct way of updating `A`?

A. `balance += 5;`
B. `A.balance += 5;`
C. `deposit(5);`
D. `A.deposit(5);`
E. `A.getBalance() += 5;`
Inheritance

inheritance (noun):
*a physical or mental characteristic inherited from your parents, or the process by which this happens*
Inheritance

inheritance (noun):
a physical or mental characteristic inherited from your parents, or the process by which this happens
Certain classes like shapes can be organized into a hierarchy:
Superclasses and subclasses

A more generic class is called a superclass, while a more specialized class is called a subclass.

Example: Triangle and Box are subclasses of Shape.
What is the most natural relationship between the classes Account, ChequingAccount, and SavingsAccount?

A. Account and ChequingAccount are superclasses of SavingsAccount.
B. Account and ChequingAccount are subclasses of SavingsAccount.
C. SavingsAccount and ChequingAccount are superclasses of Account.
D. SavingsAccount and ChequingAccount are subclasses of Account.
Defining subclasses

To define a new subclass of a given class, the colon operator is used:

```cpp
class Triangle : public Shape
{
};

class Box : public Shape
{
};
```
Inheritance

The methods of a superclass are automatically inherited by any of its subclasses.

In other words, a subclass has the variables and methods of the superclass it was derived from—but the reverse is not true.
Inheritance

Caveat: By default, variables and methods of a superclass are not visible to subclasses. They can be made visible (only to subclasses) by using the protected keyword.
Inheritance

Caveat: By default, variables and methods of a superclass are not visible to subclasses. They can be made visible (only to subclasses) by using the protected keyword.

```cpp
class Shape
{
    protected:
        int height, width;

    public:
        ...
};
```
Substitution principle

Since a subclass is a special case of a superclass, you can always use a subclass to stand in for a superclass if necessary.
In this class hierarchy which classes could not stand in for a Dog?

A. Animal
B. Cat, Calico
C. Animal, Cat, Calico
D. Cat, Calico, Poodle
E. Animal, Cat, Calico, Poodle
Example

class Box : public Shape
{
  public:
    // Draw visual depiction of the box to the standard output
    void draw()
    {
      for(int i=0; i<height; i++)
        cout << string(width, '*') << endl;
    }
};

class Triangle : public Shape
{
  public:
    // Draw visual depiction of the triangle to the standard output
    void draw()
    {
      for(int i=1; i<=height; i++)
        cout << string(i*width/height, '*') << endl;
    }
};
Example

```c
int main()
{
    Triangle t;
    t.init(5,5);
    t.draw();

    Box b;
    b.init(10,10);
    b.draw();
}
```
If \( b \) is a Box and \( s \) is a Shape which line has a problem?

A. Box \( b2 = b; \)
B. Shape \( s2 = s; \)
C. Box \( b2 = s; \)
D. Shape \( s2 = b; \)
Polymorphism
Polymorphism

descriptions term:  
\textit{the condition of occurring in several different forms}
polymorphism (noun):
the condition of occurring in several different forms
Method overloading

Two methods of a class can share the same name, so long as the number of parameters or parameter types are different.
class Shape
{
    ...
    void init(int h, int w)
    {
        height = h;
        width = w;
    }
    void init(int hw)
    {
        height = hw;
        width = hw;
    }
}

int main()
{
    Box b;
    b.init(10);
    b.draw();
    b.init(5,5);
    b.draw();
}
Method overloading

class Shape
{
    ... 
    void init(int h, int w) { 
        height = h; 
        width = w; 
    } 
    void init(int hw) { 
        height = hw; 
        width = hw; 
    } 
} 

int main() {
    Box b;
    b.init(10);
    b.draw();
    b.init(5,5);
    b.draw();
}
Consider the following code:

```cpp
string mystery(int a) { return "A"; }
string mystery(string a) { return "B"; }
string mystery(string a, int b) { return "C"; }
string mystery(int a, string b) { return "D"; }
```

What does `mystery(1, "A")` return?

A. "A"
B. "B"
C. "C"
D. "D"
Method overriding

A subclass can *override* a method of a superclass.

The new method has the same name and parameters as the overridden method but can have a different implementation.
Method overriding

class TextBox : public Box
{
    string text;
public:
    void setText(string s)
    {
        text = s;
    }
    void draw() // Overridden
    {
        for(int i=0; i<height; i++)
        {
            if(i==height/2)
            {
                cout << text << endl;
            }
            else
            {
                cout << string(width, '*') << endl;
            }
        }
    }
};
Method overriding

class Box : public Shape
{
    ...
    void draw()
    { ... }
};

class TextBox : public Box
{
    ...
    void draw()
    { ... }
};

int main()
{
    Box b;
    b.init(5);
    b.draw();
    TextBox tb;
    tb.init(5);
    tb.setText("Hello");
    tb.draw();
}
Method overriding

class Box : public Shape
{
    ...
    void draw()
    {
        ...
    }
};

class TextBox : public Box
{
    ...
    void draw()
    {
        ...
    }
};

int main()
{
    Box b;
    b.init(5);
    b.draw();
    TextBox tb;
    tb.init(5);
    tb.setText("Hello");
    tb.draw();
}
Consider the following:

class Shape { int height, width; };
class Box : public Shape { void draw() { ... } };
class Triangle: public Shape { void draw() { ... } };

Method overriding is used in these simplified class definitions:

A. True
B. False
Consider the following:

```cpp
class Box : public Shape
{
public:
    void init(int hw) { ... }
};

class TextBox: public Shape
{
public:
    void init(int hw, string s) { ... }
};
```

Method overriding is used in these simplified class definitions:

A. True
B. False
Dynamic binding

By default, C++ will determine which method implementation is used at compile time based on the object’s type.

However, a more specific choice could be made at run time because more information is known.
Dynamic binding

Declaring a method as virtual tells the compiler to use “dynamic binding” (on the method in this class and any of its subclasses) and make the choice at run time.

class Box : public Shape { ... virtual void draw() { ... } }; 
class TextBox : public Box { ... void draw() { ... } }; 

void DrawBox(Box& b)
{
    b.draw();
}

Which draw method will be called in DrawBox?
Dynamic binding

Declaring a method as virtual tells the compiler to use “dynamic binding” (on the method in this class and any of its subclasses) and make the choice at run time.

```cpp
class Box : public Shape {
    ... virtual void draw() { ... };
};
class TextBox : public Box {
    ... void draw() { ... }
};

void DrawBox(Box& b)
{
    b.draw();
}
```

Which `draw` method will be called in `DrawBox`?

Depends on the type of object passed to `DrawBox`!
Dynamic binding

```java
int main()
{
    Box b;
    b.init(5);
    DrawBox(b);
}

DrawBox will call the draw method of Box.
```

```java
int main()
{
    TextBox tb;
    tb.init(5);
    tb.setText("Hello");
    DrawBox(tb);
}

DrawBox will call the draw method of TextBox.
```
Consider the following:

class Shape { int height, width; }
class Box : public Shape { ... virtual void draw() { ... } }; 
class TextBox : public Box { ... void draw() { ... } }; 
void DrawBox(Box& b) { b.draw(); } 
Shape s; 
TextBox tb;

What will be the result of running DrawBox(s) and, separately, DrawBox(tb)?

A. Box’s draw run in both cases.
B. Box’s draw run in first case, TextBox’s draw run in second case.
C. TextBox’s draw run in both cases.
D. Error in first case, TextBox’s draw in second case.
E. Error in first case, Box’s draw in second case.
Consider the following:

class Shape { int height, width; }
class Box : public Shape { ... virtual void draw() { ... } };  
class TextBox : public Box { ... void draw() { ... } };  
void DrawShape(Shape& s) { s.draw(); }

int main()
{
    Shape s;
    DrawShape(s);
}

What kind of error arises in this program?

A. Compile time error
B. Run time error
Summary

We discussed three kinds of polymorphism:

1. Method overloading:
   Methods/functions that have the same name but different parameters.
Summary

We discussed three kinds of polymorphism:

1. Method overloading:
   Methods/functions that have the same name but different parameters.

2. Method overriding:
   Methods that have the same name and same parameters, but belong to a superclass and subclass.
Summary

We discussed three kinds of polymorphism:

1. Method overloading:
   Methods/functions that have the same name but different parameters.

2. Method overriding:
   Methods that have the same name and same parameters, but belong to a superclass and subclass.

3. Dynamic binding:
   Method overriding of a function that has been declared virtual.