

CSCI 1061U

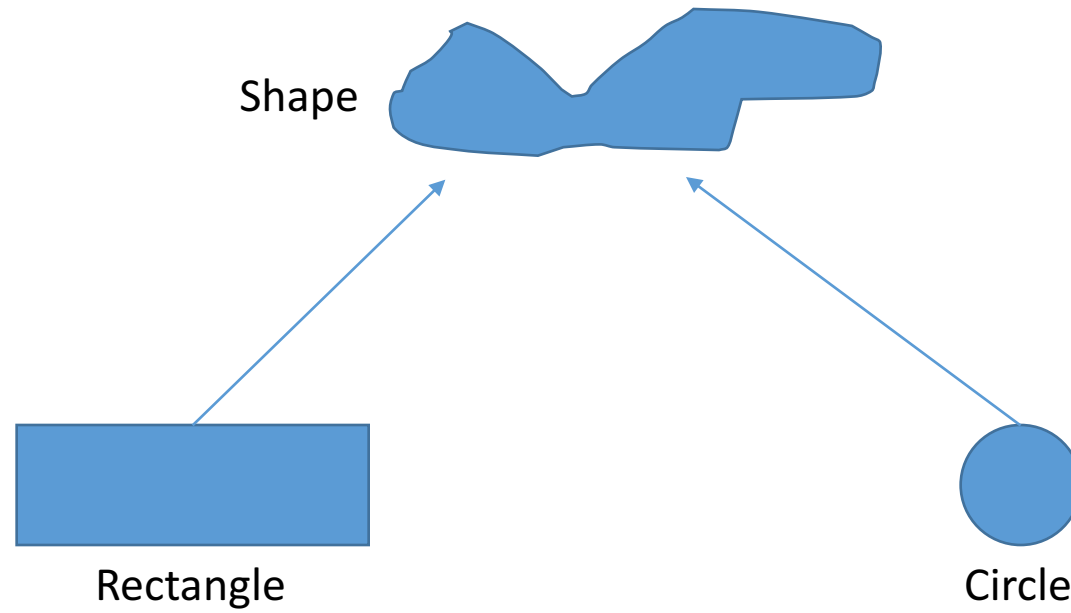
Programming Workshop 2

Virtual Functions

Polymorphism

- Fundamental principle of object-oriented programming
- Associating many meanings to one function
- Virtual functions provide this capability

Polymorphism example



The image shows a C++ IDE with two tabs: `Shapes.h` and `Shapes.cpp`. The `Shapes.h` tab is active, showing the header file with the following code:

```
1 class Shape {
2 public:
3     Shape();
4     ~Shape();
5     void draw(); // draws an empty shape
6 };
7
8 class Rectangle : public Shape
9 {
10 public:
11     Rectangle(int w, int h);
12     ~Rectangle();
13     void draw(); // draws a rectangle
14
15 private:
16     int _w, _h; // width & height
17 };
18
19 class Circle : public Shape
20 {
21 public:
22     Circle(int r);
23     ~Circle();
24     void draw(); // draws a circle
25
26 private:
27     int _r; // radius
28 };
```

The `Shapes.cpp` tab is also visible, showing the implementation of the classes:

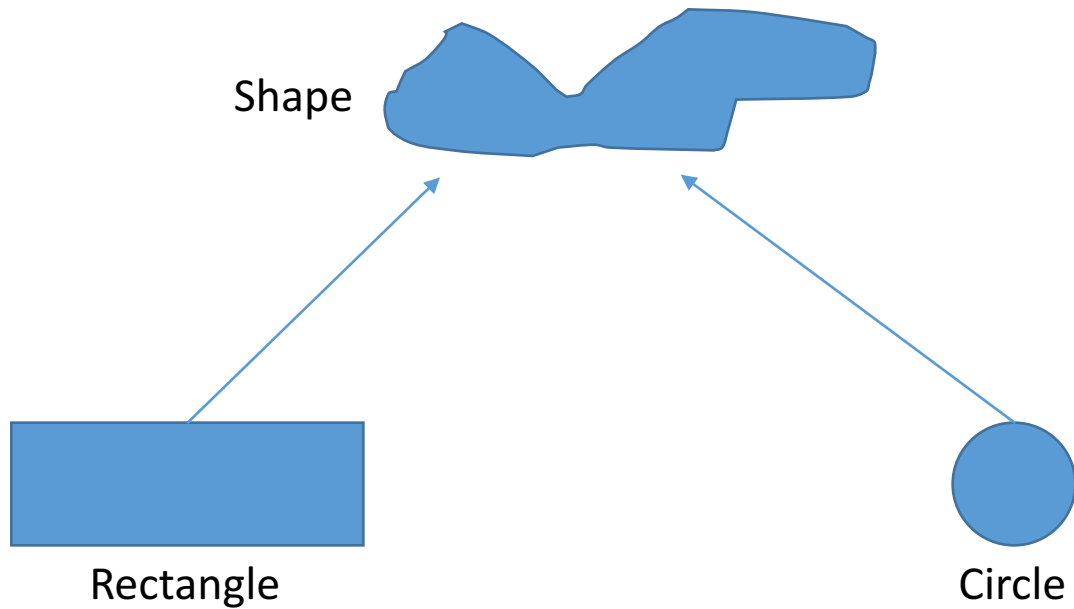
```
1 #include "Shapes.h"
2 #include <iostream>
3 using namespace std;
4
5 Shape::Shape() {}
6
7 Shape::~Shape() {}
8
9 void Shape::draw()
10 {
11     cout << "Drawing an empty shape" << endl;
12 }
13
14 //////////////////////////////////////
15 Rectangle::Rectangle(int w, int h)
16 : _w(w), _h(h)
17 {}
18
19 Rectangle::~Rectangle() {}
20
21 void Rectangle::draw()
22 {
23     cout << "Drawing a rectangle of width " << _w
24         << " and height " << _h
25         << endl;
26 }
27
28 //////////////////////////////////////
29 Circle::Circle(int r)
30 : _r(r)
31 {}
32
33 Circle::~Circle() {}
34
35 void Circle::draw()
36 {
37     cout << "Drawing a circle of radius " << _r << endl;
38 }
```

The IDE status bar at the bottom shows "Line 28, Column 3", "0 misspelled words", "Spaces: 4", and "C++".

Shapes.h

Shapes.cpp

Polymorphism example

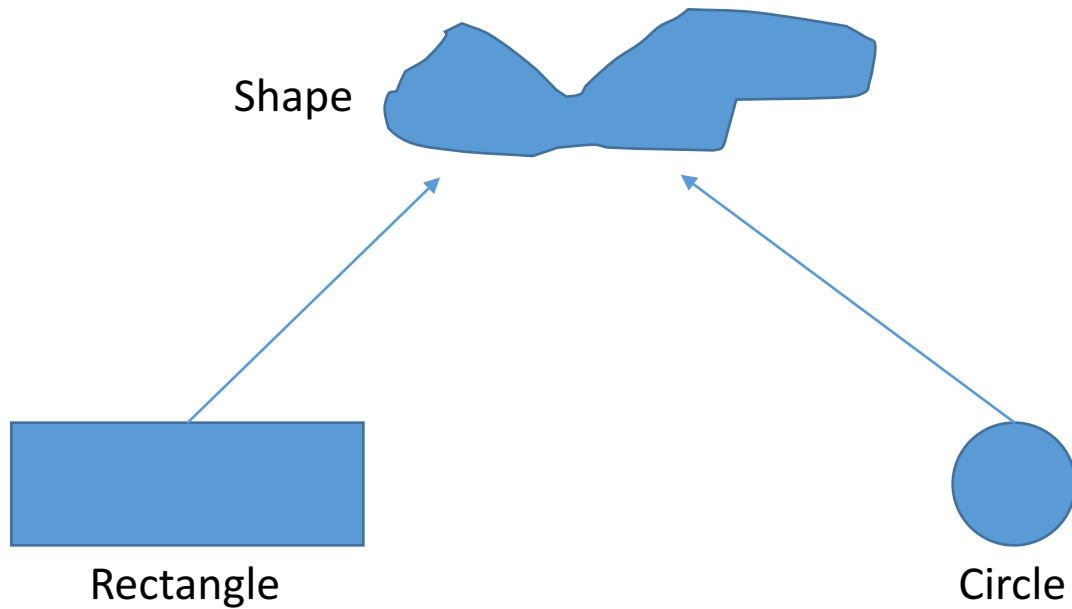


```
main.cpp UNREGISTERED
Shapes.h * main.cpp * main2.cpp *
1  #include "Shapes.h"
2
3  int main()
4  {
5      Circle* c = new Circle(2);
6      c->draw(); // this draws circle
7
8      Rectangle* r = new Rectangle(3,6);
9      r->draw(); // this draws rectangle
10
11     delete c; delete r;
12 }
```

Line 12, Column 2 0 misspelled words Spaces: 4 C++

This works

Polymorphism example

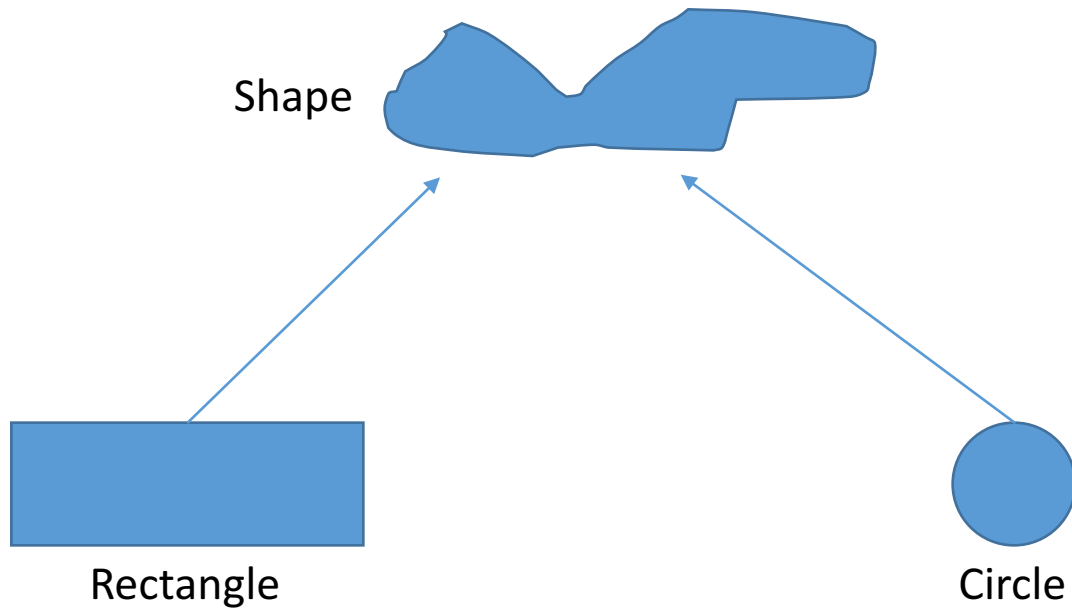


```
main2.cpp UNREGISTERED
Shapes.h x main.cpp x main2.cpp Shap.cpp x
1  #include "Shapes.h"
2
3  int main()
4  {
5      Shape* s[2]; // a list of different kind of shapes
6      s[0] = new Circle(4);
7      s[1] = new Rectangle(4,5);
8
9      for (int i=0; i<2; ++i) {
10         s[i]->draw();
11     }
12
13     for (int i=0; i<2; ++i) {
14         delete s[i];
15     }
16 }
```

Line 16, Column 2 0 misspelled words Spaces: 4 C++

Will this work?

Polymorphism example

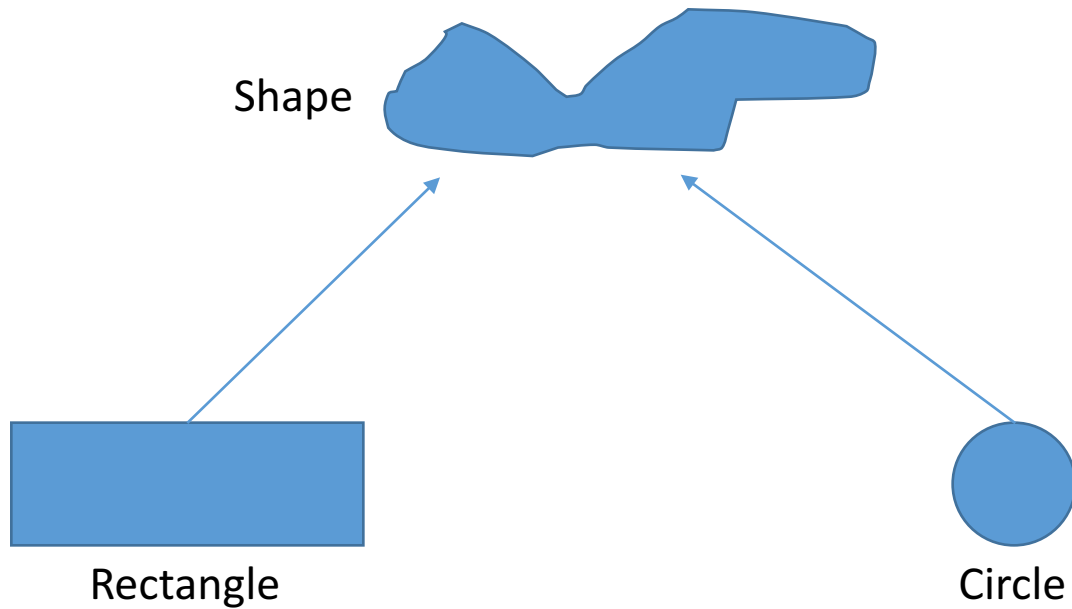


```
main2.cpp UNREGISTERED
Shapes.h x main.cpp x main2.cpp Shap.cpp x
1  #include "Shapes.h"
2
3  int main()
4  {
5      Shape* s[2]; // a list of different kind of shapes
6      s[0] = new Circle(4);
7      s[1] = new Rectangle(4,5);
8
9      for (int i=0; i<2; ++i) {
10         s[i]->draw();
11     }
12
13     for (int i=0; i<2; ++i) {
14         delete s[i];
15     }
16 }
```

Line 16, Column 2 0 misspelled words Spaces: 4 C++

Will this work? **NO**

Polymorphism example



main2.cpp UNREGISTERED

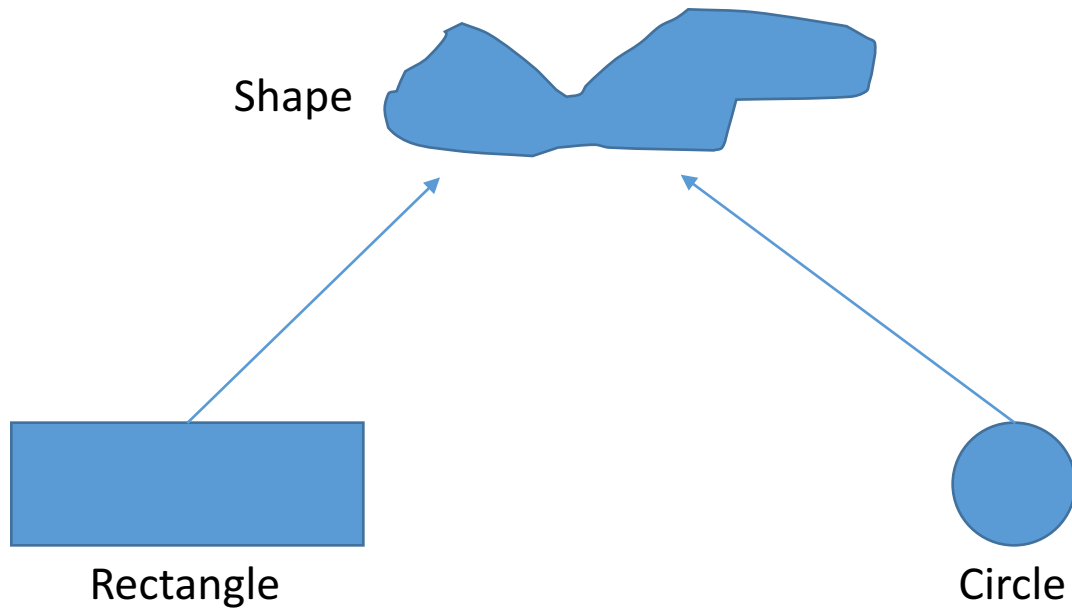
```
1  #include "Shapes.h"
2
3  int main()
4  {
5      Shape* s[2]; // a list of different kind of shapes
6      s[0] = new Circle(4);
7      s[1] = new Rectangle(4,5);
8
9      for (int i=0; i<2; ++i) {
10         s[i]->draw();
11     }
12
13     for (int i=0; i<2; ++i) {
14         delete s[i];
15     }
16 }
```

This invokes the draw() function in Shape class.

Line 16, Column 2 0 misspelled words Spaces: 4 C++

Will this work? **NO**

Polymorphism example



```
main2.cpp UNREGISTERED
Shapes.h x main.cpp x main2.cpp ● Shapes.cpp x
1  #include "Shapes.h"
2
3  int main()
4  {
5      Shape* s[2]; // a list of different kind of shapes
6      s[0] = new Circle(4);
7      s[1] = new Rectangle(4,5);
8
9      for (int i=0; i<2; ++i) {
10         s[i]->draw();
11     }
12
13     for (int i=0; i<2; ++i) {
14         delete s[i];
15     }
16 }
```

Potential memory leak.
Invoking destructor of
Shape class.

Line 16, Column 2 0 misspelled words Spaces: 4 C++

Will this work? **NO**

Solution: Virtual Functions

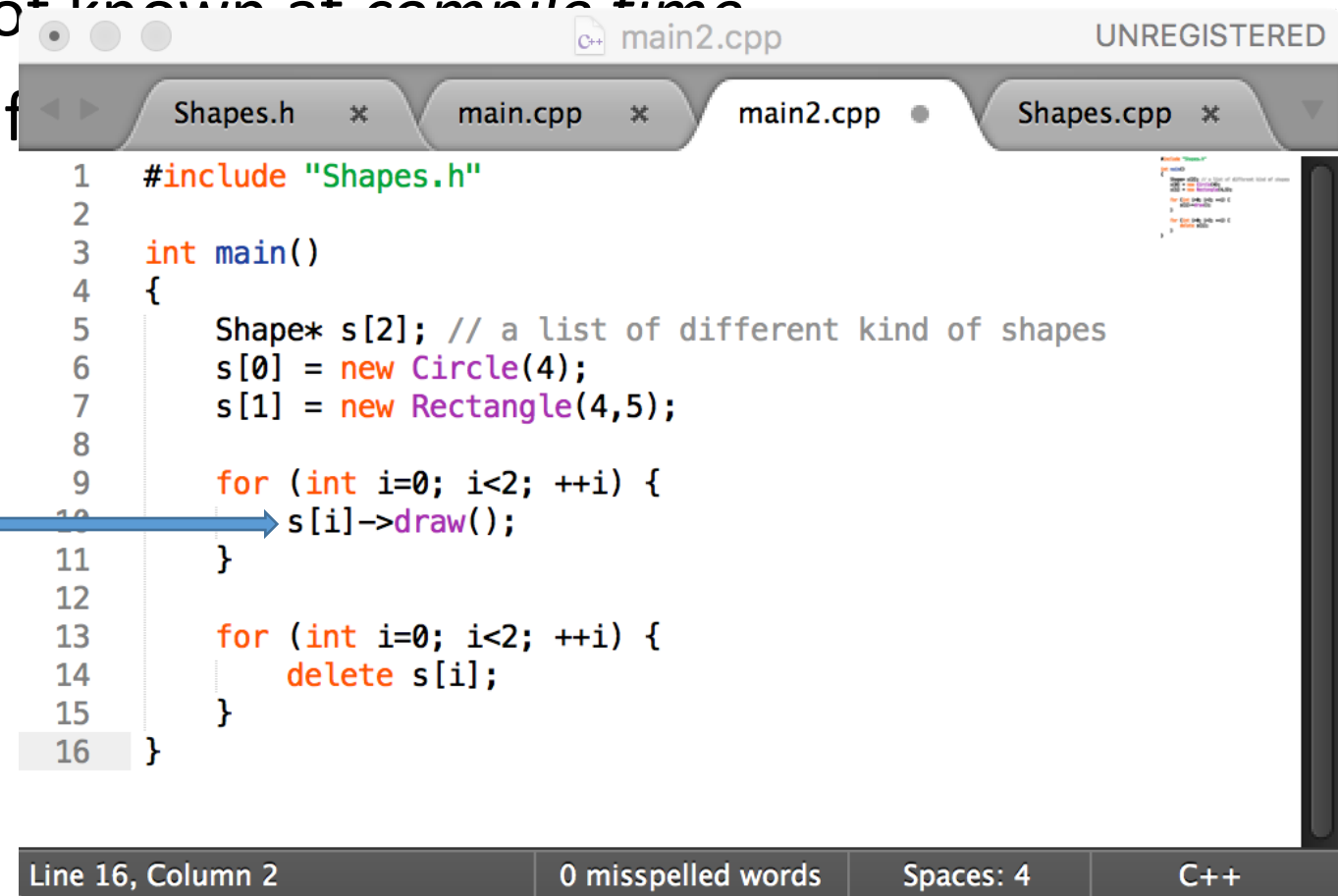
- Function implementation is not known at *compile time*
- Function implementation is inferred from object instance at *run time*
- Late or dynamic binding

Solution: Virtual Functions

- Function implementation is not known at compile time
- Function implementation is inferred at runtime
- Late or dynamic binding

Use Circle draw() function if s[i] is of type Circle, use Rectangle draw() function if s[i] is of type Rectangle, etc.

The type of s[i] is not known at compile time.



```
1  #include "Shapes.h"
2
3  int main()
4  {
5      Shape* s[2]; // a list of different kind of shapes
6      s[0] = new Circle(4);
7      s[1] = new Rectangle(4,5);
8
9      for (int i=0; i<2; ++i) {
10         s[i]->draw();
11     }
12
13     for (int i=0; i<2; ++i) {
14         delete s[i];
15     }
16 }
```

Line 16, Column 2 0 misspelled words Spaces: 4 C++

Solution: Virtual Functions

- Function implementation
- Function implementation
- Late or dynamic binding

Type of s is not known until run time

```
main3.cpp UNREGISTERED
Shapes.h * main.cpp * main2.cpp * main3.cpp * Shapes.cpp *
1  #include "Shapes.h"
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7      Shape* s = 0; // a pointer to hold a shape
8                  // initially set to null
9
10     char choice;
11     cout << "Enter Circle (1) or Rectangle (2): ";
12     cin >> choice;
13
14     switch (choice)
15     {
16     case '1': s = new Circle(4); break;
17     case '2': s = new Rectangle(4,7); break;
18     default: cout << "Invalid shape entered." << endl; return 1;
19     }
20
21     s->draw(); delete s;
22     return 0;
23 }
```

User selects a Circle or a Rectangle at run time

Line 14, Column 1 0 misspelled words Spaces: 4 C++

Textbook Slides

Learning Objectives

- Virtual Function Basics
 - Late binding
 - Implementing virtual functions
 - When to use a virtual function
 - Abstract classes and pure virtual functions
- Pointers and Virtual Functions
 - Extended type compatibility
 - Downcasting and upcasting
 - C++ "under the hood" with virtual functions

Virtual Function Basics

- Polymorphism
 - Associating many meanings to one function
 - Virtual functions provide this capability
 - Fundamental principle of object-oriented programming!
- Virtual
 - Existing in "essence" though not in fact
- Virtual Function
 - Can be "used" before it's "defined"

Figures Example

- Best explained by example:
- Classes for several kinds of figures
 - Rectangles, circles, ovals, etc.
 - Each figure an object of different class
 - Rectangle data: height, width, center point
 - Circle data: center point, radius
- All derive from one parent-class: Figure
- Require function: draw()
 - Different instructions for each figure

Figures Example 2

- Each class needs different *draw* function
- Can be called "draw" in each class, so:
Rectangle r;
Circle c;
r.draw(); //Calls Rectangle class's draw
c.draw(); //Calls Circle class's draw
- Nothing new here yet...

Figures Example: center()

- Parent class Figure contains functions that apply to "all" figures; consider:
center(): moves a figure to center of screen
 - Erases 1st, then re-draws
 - So Figure::center() would use function draw() to re-draw
 - Complications!
 - Which draw() function?
 - From which class?

Figures Example: New Figure

- Consider new kind of figure comes along:
Triangle class
 derived from Figure class
- Function center() inherited from Figure
 - Will it work for triangles?
 - It uses draw(), which is different for each figure!
 - It will use Figure::draw() → won't work for triangles
- Want inherited function center() to use function Triangle::draw() NOT function Figure::draw()
 - But class Triangle wasn't even WRITTEN when Figure::center() was! Doesn't know "triangles"!

Figures Example: Virtual!

- Virtual functions are the answer
- Tells compiler:
 - "Don't know how function is implemented"
 - "Wait until used in program"
 - "Then get implementation from object instance"
- Called late binding or dynamic binding
 - Virtual functions implement late binding

Virtual Functions: Another Example

- Bigger example best to demonstrate
- Record-keeping program for automotive parts store
 - Track sales
 - Don't know all sales yet
 - 1st only regular retail sales
 - Later: Discount sales, mail-order, etc.
 - Depend on other factors besides just price, tax

Virtual Functions: Auto Parts

- Program must:
 - Compute daily gross sales
 - Calculate largest/smallest sales of day
 - Perhaps average sale for day
- All come from individual bills
 - But many functions for computing bills will be added "later"!
 - When different types of sales added!
- So function for "computing a bill" will be virtual!

Class Sale Definition

- class Sale
 - {
 - public:
 - Sale();
 - Sale(double thePrice);
 - double getPrice() const;
 - virtual*** double bill() const;
 - double savings(const Sale& other) const;
 - private:
 - double price;
 - };

Member Functions

savings and operator <

- `double Sale::savings(const Sale& other) const`
 {
 return (bill() – other.bill());
 }
- `bool operator < (const Sale& first,`
 `const Sale& second)`
 {
 return (first.bill() < second.bill());
 }
- Notice BOTH use member function bill()!

Class Sale

- Represents sales of single item with no added discounts or charges.
- Notice reserved word "virtual" in declaration of member function *bill*
 - Impact: Later, derived classes of Sale can define THEIR versions of function bill
 - Other member functions of Sale will use version based on object of derived class!
 - They won't automatically use Sale's version!

Derived Class DiscountSale Defined

- ```
class DiscountSale : public Sale
{
public:
 DiscountSale();
 DiscountSale(double thePrice,
 double the Discount);
 double getDiscount() const;
 void setDiscount(double newDiscount);
 double bill() const;
private:
 double discount;
};
```

# DiscountSale's Implementation of bill()

- `double DiscountSale::bill() const`  
  {  
    double fraction = discount/100;  
    return (1 – fraction)\*getPrice();  
  }
- Qualifier "virtual" does not go in actual function definition
  - "Automatically" virtual in derived class
  - Declaration (in interface) not required to have "virtual" keyword either (but usually does)

# DiscountSale's Implementation of bill()

- Virtual function in base class:
  - "Automatically" virtual in derived class
- Derived class declaration (in interface)
  - Not required to have "virtual" keyword
  - But typically included anyway, for readability

# Derived Class DiscountSale

- DiscountSale's member function bill() implemented differently than Sale's
  - Particular to "discounts"
- Member functions *savings* and "<"
  - Will use this definition of bill() for all objects of DiscountSale class!
  - Instead of "defaulting" to version defined in Sales class!

# Virtual: Wow!

- Recall class Sale written long before derived class DiscountSale
  - Members savings and "<" compiled before even had ideas of a DiscountSale class
- Yet in a call like:  
DiscountSale d1, d2;  
d1.savings(d2);
  - Call in savings() to function bill() knows to use definition of bill() from DiscountSale class
- Powerful!

# Virtual: How?

- To write C++ programs:
  - Assume it happens by "magic"!
- But explanation involves late binding
  - Virtual functions implement late binding
  - Tells compiler to "wait" until function is used in program
  - Decide which definition to use based on calling object
- Very important OOP principle!

# Overriding

- Virtual function definition changed in a derived class
  - We say it's been "overridden"
- Similar to redefined
  - Recall: for standard functions
- So:
  - Virtual functions changed: ***overridden***
  - Non-virtual functions changed: ***redefined***




# C++11 **override** keyword

- C++11 includes the **override** keyword to make it clear if a function is overridden or redefined

```
class Sale
{
 public:
 ...
 virtual double bill() const;
 ...
};

class DiscountSale : public Sale
{
 public:
 ...
 double bill() const override;
 ...
};
```

Makes it explicit  
that this  
function  
overrides **bill()**  
in the Sale class



# C++11 **final** keyword

- C++11 includes the **final** keyword to prevent a function from being overridden. Useful if a function is overridden but don't want a derived classes to override it again.

```
class Sale
{
public:
 ...
 virtual double bill() const final;
 ...
};
```

Cannot override

```
class DiscountSale : public Sale
{
public:
 ...
 double bill() const;
 ...
};
```

Results in  
compiler error

# Virtual Functions: Why Not All?

- Clear advantages to virtual functions as we've seen
- One major disadvantage: overhead!
  - Uses more storage
  - Late binding is "on the fly", so programs run slower
- So if virtual functions not needed, should not be used

# Pure Virtual Functions

- Base class might not have "meaningful" definition for some of its members!
  - Its purpose solely for others to derive from
- Recall class Figure
  - All figures are objects of derived classes
    - Rectangles, circles, triangles, etc.
  - Class Figure has no idea how to draw!
- Make it a pure virtual function:  
`virtual void draw() = 0;`

# Abstract Base Classes

- Pure virtual functions require no definition
  - Forces all derived classes to define "their own" version
- Class with one or more pure virtual functions is: abstract base class
  - Can only be used as base class
  - No objects can ever be created from it
    - Since it doesn't have complete "definitions" of all its members!
- If derived class fails to define all pure's:
  - It's an abstract base class too

# Extended Type Compatibility

- Given:  
Derived is derived class of Base
  - Derived objects can be assigned to objects of type Base
  - But NOT the other way!
- Consider previous example:
  - A DiscountSale "is a" Sale, but reverse not true

# Extended Type Compatibility Example

- class Pet  
  {  
  public:  
    string name;  
    virtual void print() const;  
  };  
  class Dog : public Pet  
  {  
  public:  
    string breed;  
    virtual void print() const;  
  };

# Classes Pet and Dog

- Now given declarations:  
Dog vdog;  
Pet vpet;
- Notice member variables name and breed are public!
  - For example purposes only! Not typical!



# Using Classes Pet and Dog

- Anything that "is a" dog "is a" pet:
  - `vdog.name = "Tiny";`  
`vdog.breed = "Great Dane";`  
`vpet = vdog;`
  - These are allowable
- Can assign values to parent-types, but not reverse
  - A pet "is not a" dog (not necessarily)

# Slicing Problem

- Notice value assigned to vpet "loses" it's breed field!
  - `cout << vpet.breed;`
    - Produces ERROR msg!
  - Called slicing problem
- Might seem appropriate
  - Dog was moved to Pet variable, so it should be treated like a Pet
    - And therefore not have "dog" properties
  - Makes for interesting philosophical debate

# Slicing Problem Fix

- In C++, slicing problem is nuisance
  - It still "is a" Great Dane named Tiny
  - We'd like to refer to it's breed even if it's been treated as a Pet
- Can do so with pointers to dynamic variables

# Slicing Problem Example

- ```
Pet *ppet;  
Dog *pdog;  
pdog = new Dog;  
pdog->name = "Tiny";  
pdog->breed = "Great Dane";  
ppet = pdog;
```
- Cannot access breed field of object pointed to by ppet:

```
cout << ppet->breed;    //ILLEGAL!
```

Slicing Problem Example

- Must use virtual member function:
ppet->print();
 - Calls print member function in Dog class!
 - Because it's virtual
 - C++ "waits" to see what object pointer ppet is actually pointing to before "binding" call

Virtual Destructors

- Recall: destructors needed to de-allocate dynamically allocated data
- Consider:
Base *pBase = new Derived;
...
delete pBase;
 - Would call base class destructor even though pointing to Derived class object!
 - Making destructor ***virtual*** fixes this!
- Good policy for all destructors to be virtual

Casting

- Consider:
Pet vpet;
Dog vdog;
...
vdog = static_cast<Dog>(vpel); //ILLEGAL!
- Can't cast a pet to be a dog, but:
vpel = vdog; // Legal!
vpel = static_cast<Pet>(vdog); //Also legal!
- Upcasting is OK
 - From descendant type to ancestor type

Downcasting

- Downcasting dangerous!
 - Casting from ancestor type to descended type
 - Assumes information is "added"
 - Can be done with `dynamic_cast`:

```
Pet *ppet;  
ppet = new Dog;  
Dog *pdog = dynamic_cast<Dog*>(ppet);
```

 - Legal, but dangerous!
- Downcasting rarely done due to pitfalls
 - Must track all information to be added
 - All member functions must be virtual

Inner Workings of Virtual Functions

- Don't need to know how to use it!
 - Principle of information hiding
- Virtual function table
 - Compiler creates it
 - Has pointers for each virtual member function
 - Points to location of correct code for that function
- Objects of such classes also have pointer
 - Points to virtual function table

Summary 1

- Late binding delays decision of which member function is called until runtime
 - In C++, virtual functions use late binding
- Pure virtual functions have no definition
 - Classes with at least one are abstract
 - No objects can be created from abstract class
 - Used strictly as base for others to derive

Summary 2

- Derived class objects can be assigned to base class objects
 - Base class members are lost; slicing problem
- Pointer assignments and dynamic objects
 - Allow "fix" to slicing problem
- Make all destructors virtual
 - Good programming practice
 - Ensures memory correctly de-allocated