2 Polymorphism

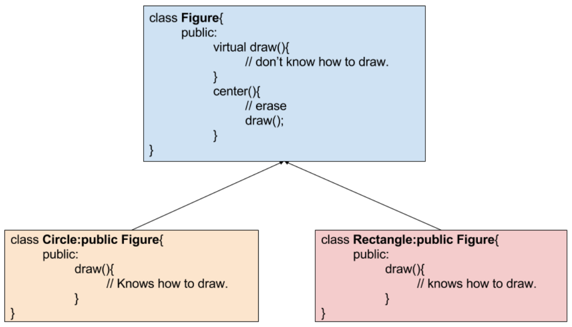
- **Polymorphism** refers to the ability to associate multiple meanings with one function name using a mechanism called **late binding**
- **Polymorphism** is a key component of the philosophy of object oriented programming

2.1 A Late Binding Example

- Imagine a graphics program with several types of figures
  - Each figure may be an object of a different class, such as a circle, oval, rectangle, etc.
  - Each is a descendant of a class **Figure**
  - Each has a function **draw**( ) implemented with code specific to each shape
  - Class **Figure** has functions common to all figures

A Problem

- Suppose that class **Figure** has a function **center**
  - Function **center** moves a figure to the center of the screen by erasing the figure and redrawing it in the center of the screen
  - Function **center** is inherited by each of the derived classes
    - Function **center** uses each derived object's **draw** function to draw the figure
    - The **Figure** class does not know about its derived classes, so it cannot know how to draw each figure



```
class Figure{
        public:
                virtual draw(){
                        // don't know how to draw.
                }
                center(){
                        // erase
                        draw();
                }
}
```

```
class Circle:public Figure{
        public:
                draw(){
                        // Knows how to draw.
                }
}
```

```
class Rectangle:public Figure{
        public:
                draw(){
                        // knows how to draw.
                }
}
```

2.2 Virtual Functions

- Because the **Figure** class includes a method to draw figures, but the **Figure** class cannot know how to draw the figures, **virtual functions** are used
- Making a function **virtual** tells the compiler that you don't know how the function is implemented and to wait until the function is used in a program, then get the implementation from the object.
  - This is called **late binding**

2.3 Virtual Functions in C++

- As another example, let's design a record-keeping program for an auto parts store
  - We want a versatile program, but we do not know all the possible types of sales we might have to account for
    - Later we may add mail-order and discount sales
    - Functions to compute bills will have to be added later when we know what type of sales to add
    - To accommodate the future possibilities, we will make the **bill** function a virtual function

2.3.1 The Sale Class

- All sales will be derived from the base class **Sale**
- The **bill** function of the **Sale** class is virtual

- The member function **savings** and **operator** $<$ each use **bill**

```
#ifndef sale_h
#define sale_h
#include <iostream>

namespace cosc3000
{
    class Sale
    {
    public:
        /// Default constructor
        Sale();
        /// constructor takes the price
        Sale(double the_price);
        /// Returns the bill
        virtual double bill() const;
        /// Returns the savings if you buy other
        /// instead of the calling object.
        double savings(const Sale& other) const;
    protected:
        double price;///< retail price of item
    };

    /// Compares tei sales to see which is larger
    bool operator < (const Sale &first,const Sale &second);
} // cosc3000
#endif /* sale_h */
```

```
#include "sale.h"
namespace cosc3000
{
    Sale::Sale(): price(0)
    {
        /// nothing to do
    }
    Sale::Sale(double the_price) : price(the_price)
    {
        /// nothing to do
    }
    double Sale::bill() const
    {
        return price;
    }
    double Sale::savings(const Sale& other) const
    {
        return (bill() - other.bill());
    }
    bool operator < (const Sale &first, const Sale &second)
    {
        return (first.bill() < second.bill());
    }
}//cosc3000
```

Virtual Function bill

- Because function **bill** is virtual in class **Sale**, function **savings** and **operator** $<$, defined only in the base class, can in turn use a version of **bill** found in a derived class
  - When a **DiscountSale** object calls its **savings** function, defined only in the base class, function savings

calls function **bill**

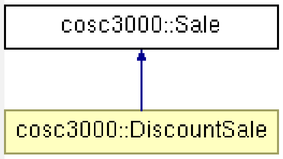
– Because **bill** is a virtual function in class **Sale**, C++ uses the version of **bill** defined in the object that called **savings**

DiscountSale::bill

- Class **DiscountSale** has its own version of virtual function **bill**
- Even though class **Sale** is already compiled, **Sale::savings( )** and **operator**< can still use function **bill** from the **DiscountSale** class
  – Note that those functions are called by reference.
- The keyword **virtual** tells C++ to wait until **bill** is used in a program to get the implementation of **bill** from the calling object

```
#ifndef discountsale_h
#define discountsale_h
#include "sale.h"
namespace cosc3000
{
    class DiscountSale : public Sale
    {
    public:
        /// Default constructor
        DiscountSale();
        /// Discount is expressed as a percent of the price.
        DiscountSale(double the_price,double the_discount);
        /// Returns the bill
        virtual double bill() const;
    protected:
        double discount; ///< percent of discount
    };
}//cosc3000

#endif /* discountsale_h */
```



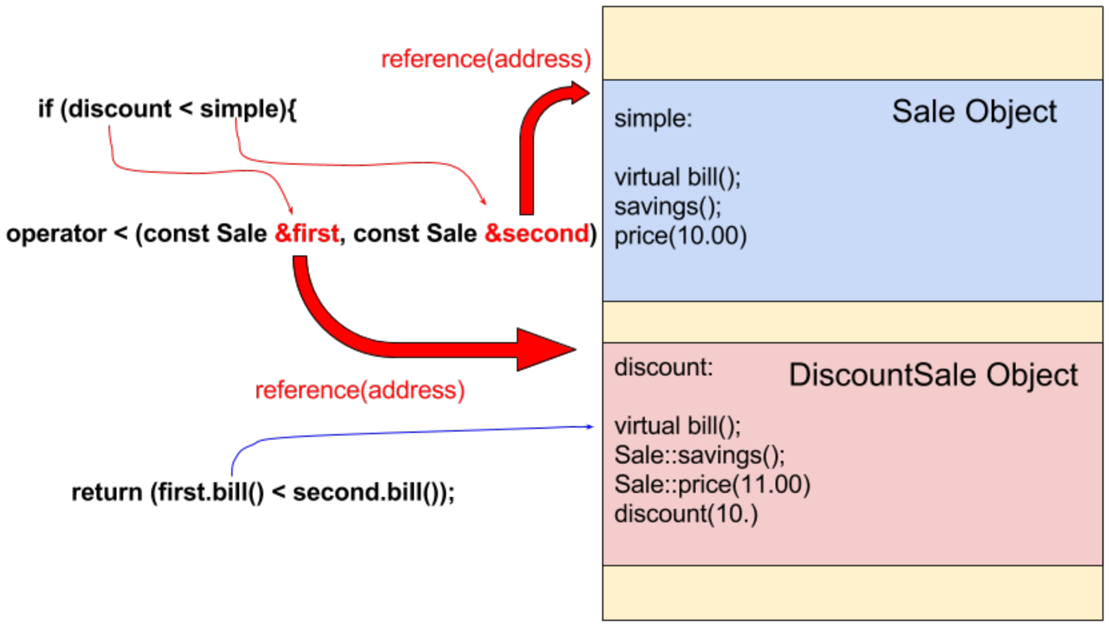
```
#include "discountsale.h"
namespace cosc3000
{
    DiscountSale::DiscountSale():Sale(),discount(0)
    {
        /// nothing to do
    }
    DiscountSale::DiscountSale(double the_price,double the_discount)
    :Sale(the_price),discount(the_discount)
    {
        /// nothing to do
    }
    double DiscountSale::bill() const
    {
        double fraction = discount / 100;
        return (1 - fraction) * price;
    }
}
```

main function

```cpp
///
/// sample49.cpp
///
#include <iostream>
#include "discountsale.h"
int main(int argc, char **argv)
{
    /// One item at $10.00
    cosc3000::Sale simple(10.00);
    /// One item at $11.00 with a 10% discount
    cosc3000::DiscountSale discount(11.00, 10);

    std::cout.setf(std::ios::fixed);
    std::cout.setf(std::ios::showpoint);
    std::cout.precision(2);

    if (discount < simple){
        std::cout << "Discounted item is cheaper.\n";
        std::cout << "Saving is $" << simple.savings(discount)
                        << std::endl;
    }else{
        std::cout << "Discount item is not cheaper.\n";
    }
    return 0;
}
```



2.4 Virtual Details

- To define a function differently in a derived class and to make it virtual
  - Add keyword virtual to the function declaration in the base class
  - virtual is not needed for the function declaration in the derived class, but is often included
  - virtual is not added to the function definition

2.5 Overriding

- Virtual functions whose definitions are changed in a derived class are said to be **overridden**
- Non-virtual functions whose definitions are changed in a derived class are **redefined**