Polymorphism

Modified from Section 15.3

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

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

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

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

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
- The Sale class interface and implementation are shown in **Display 15.9** **Display 15.10**

Display 15.9

Back Next

Interface for the Base Class Sale

```
//This is the header file sale.h.
//This is the interface for the class Sale.
//Sale is a class for simple sales.
#ifndef SALE_H
#define SALE_H

#include <iostream>
using namespace std;

namespace salesavitch
{

    class Sale
    {
    public:
        Sale();
        Sale(double the_price);
        virtual double bill() const;
        double savings(const Sale& other) const;
        //Returns the savings if you buy other instead of the calling object.
    protected:
        double price;
    };

    bool operator < (const Sale& first, const Sale& second);
    //Compares two sales to see which is larger.

}//salesavitch

#endif // SALE_H
```

Display 15.10

Back Next

Implementation of the Base Class Sale

```
//This is the implementation file: sale.cpp
//This is the implementation for the class Sale.
//The interface for the class Sale is in
//the header file sale.h.
#include "sale.h"

namespace salesavitch
{

    Sale::Sale() : price(0)
    {}

    Sale::Sale(double the_price) : price(the_price)
    {}

    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());
    }

}//salesavitch
```

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 Sale::operator< can still use function bill from the DiscountSale class
  - The keyword virtual tells C++ to wait until bill is used in a program to get the implementation of bill from the calling object
  - DiscountSale is defined and used in

Display 15.11

Display 15.12

Display 15.11

Back Next

The Derived Class `DiscountSale`

This is the file `discountsale.h`.

```
//This is the interface for the class DiscountSale.
#ifndef DISCOUNTSALE_H
#define DISCOUNTSALE_H
#include "sale.h"

namespace salesavitch
{
    class DiscountSale : public Sale
    {
    public:
        DiscountSale();
        DiscountSale(double the_price, double the_discount);
        //Discount is expressed as a percent of the price.
        virtual double bill() const;
    protected:
        double discount;
    };
}//salesavitch
#endif //DISCOUNTSALE_H
```

The keyword `virtual` *is not required here, but it is good style to include it.*

This is the file `discountsale.cpp`.

```
//This is the implementation for the class DiscountSale.
#include "discountsale.h"

namespace salesavitch
{
    DiscountSale::DiscountSale() : Sale(), discount(0)
    {}

    DiscountSale::DiscountSale(double the_price, double the_discount)
              : Sale(the_price), discount(the_discount)
    {}

    double DiscountSale::bill() const
    {
        double fraction = discount/100;
        return (1 - fraction)*price;
    }
}//salesavitch
```

Display 15.12

Back Next

Use of a Virtual Function

```
//Demonstrates the performance of the virtual function bill.
#include <iostream>
#include "sale.h" //Not really needed, but safe due to ifndef.
#include "discountsale.h"
using namespace std;
using namespace salesavitch;

int main()
{
    Sale simple(10.00);//One item at $10.00.
    DiscountSale discount(11.00, 10);//One item at $11.00 with a 10% discount.

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

    if (discount < simple)
    {
        cout << "Discounted item is cheaper.\n";
        cout << "Savings is $" << simple.savings(discount) << endl;
    }
    else
        cout << "Discounted item is not cheaper.\n";

    return 0;
}
```

Sample Dialogue

```
Discounted item is cheaper.
Savings is $0.10
```

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
  - Virtual functions require considerable overhead so excessive use reduces program efficiency
  - If you want the implementation in the derived class to be called, you must call it from a **pointer** or use a **pass-by-reference parameter** of base class type

Calling Virtual Functions

- If you want the implementation in the derived class to be called by an object of base class, you must call it from
  - a **pointer** of base class type
  - a **pass-by-reference parameter** of base class type

```
void transfer(BankAccount &from, BankAccount &to,
                        double amount)
{
            from.withdraw(amount); ....
```

  - a **member function** of base class

```
void Figure::center()
{          …
            draw(0, 0); …
```

Calling Virtual Functions

- If you cast a derived class object to a base class object, then the virtual class is called from base class, e.g.

```
Figure f;
Rectangle r;
…
r.center(); // will re-draw the rectangle in center
f = r;
f.center(); // will not re-draw the rectangle, because
            // r is casted to Figure and f is not a pointer
            // or reference
```

Calling Virtual Functions

- If a base-class pointer is used, then the virtual class is called from derived class, e.g.

```
Figure *p;
Rectangle r;
…
r.center(); // will re-draw the rectangle in center
p = &r;
p->center(); // will re-draw the rectangle, because
             // the function is called from a pointer
```

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

Type Checking

- C++ carefully checks for type mismatches in the use of values and variables
- This is referred to as strong type checking
  - Generally the type of a value assigned to a variable must match the type of the variable
    - Recall that some automatic type casting occurs
- Strong type checking interferes with the concepts of inheritance

Type Checking and Inheritance

- Consider

```
class Pet
{
    public:
        virtual void print();
        string name;
};

and

class Dog :public Pet
{
    public:
        virtual void print();
        string breed;
};
```

A Sliced Dog is a Pet

- C++ allows the following assignments:

```
vdog.name = "Tiny";
vdog.breed = "Great Dane";
vpet = vdog;
```

- However, vpet will loose the breed member of vdog since an object of class Pet has no breed member
  - This code would be illegal:

```
cout << vpet.breed;
```

- This is the slicing problem

The Slicing Problem

- It is legal to assign a derived class object into a base class variable
  - This slices off data in the derived class that is not also part of the base class
  - Member functions and member variables are lost

Extended Type Compatibility

- It is possible in C++ to avoid the slicing problem
  - Using pointers to dynamic variables we can assign objects of a derived class to variables of a base class without loosing members of the derived class object

Dynamic Variables and Derived Classes

- Example:

```
Pet   *ppet;
Dog *pdog;
pdog = new Dog;
pdog->name = "Tiny";
pdog->breed = "Great
                        Dane";
ppet = pdog;
```

```
void Dog::print( )
{
   cout << "name: "
            <<  name << endl;
   cout << "breed: "
            << breed << endl;
}
```

- ppet->print( );   is legal and produces

```
name:  Tiny
breed:  Great Dane
```

Display 15.13 (1-2)

Display 15.13 (1/2)

Back Next

More Inheritance with Virtual Functions (*part 1 of 2*)

```
//Program to illustrate use of a virtual function
//to defeat the slicing problem.

#include <string>
#include <iostream>
using namespace std;

class Pet
{
public:
    virtual void print();
    string name;
};

class Dog : public Pet
{
public:
    virtual void print();//keyword virtual not needed, but put
                         //here for clarity. (It is also good style!)
    string breed;
};

int main()
{
    Dog vdog;
    Pet vpet;

    vdog.name = "Tiny";
    vdog.breed = "Great Dane";
    vpet = vdog;

    //vpet.breed; is illegal since class Pet has no member named breed

    Dog *pdog;
    pdog = new Dog;
```

More Inheritance with Virtual Functions (*part 2 of 2*)

Display 15.13 (2/2)

Back Next

```
    pdog->name = "Tiny";
    pdog->breed = "Great Dane";

    Pet *ppet;
    ppet = pdog;
    ppet->print(); // These two print the same output:
    pdog->print(); // name: Tiny breed: Great Dane

    //The following, which accesses member variables directly
    //rather than via virtual functions, would produce an error:
    //cout << "name: " << ppet->name << "  breed: "
    //      << ppet->breed << endl;
    //generates an error message: 'class Pet' has no member
    //named 'breed' .
    //See Pitfall section "Not Using Virtual Member Functions"
    //for more discussion on this.

    return 0;
}

void Dog::print()
{
    cout << "name: " << name << endl;
    cout << "breed: " << breed << endl;
}

void Pet::print()
{
    cout << "name: " << endl;//Note no breed mentioned
}
```

Sample Dialogue

```
name: Tiny
breed: Great Dane
name: Tiny
breed: Great Dane
```

Use Virtual Functions

- The previous example:

```
ppet->print( );
```

  worked because print was declared as a virtual function
- This code would still produce an error:

```
cout << "name: " << ppet->name
        << "breed: " << ppet->breed;
```

Why?

- ppet->breed is still illegal because ppet is a pointer to a Pet object that has no breed member
- Function print( ) was declared virtual by class Pet
  - When the computer sees ppet->print( ), it checks the virtual table for classes Pet and Dog and finds that ppet points to an object of type Dog
    - Because ppet points to a Dog object, code for Dog::print( ) is used

Remember Two Rules

- To help make sense of object oriented programming with dynamic variables, remember these rules
  - If the domain type of the pointer p_ancestor is a base class for the domain type of pointer p_descendant, the following assignment of pointers is allowed

    p_ancestor = p_descendant;

    and no data members will be lost
  - Although all the fields of the p_descendant are there, **only virtual functions are required to access them.**