Unit 7 (Ch 15)

Inheritance & Polymorphism

Prof. Chien-Nan (Jimmy) Liu
Dept. of Electronics & Electrical Engr.
Nat'l Yang Ming Chiao Tung Univ.

Tel: (03)5712121 ext:31211
E-mail: jimmyliu@nycu.edu.tw
http://mseda.ee.nctu.edu.tw/jimmyliu

Overview

- *7.1 Inheritance Basics*
- 7.2 More about Inheritance
- 7.3 Polymorphism

Why Inheritance?

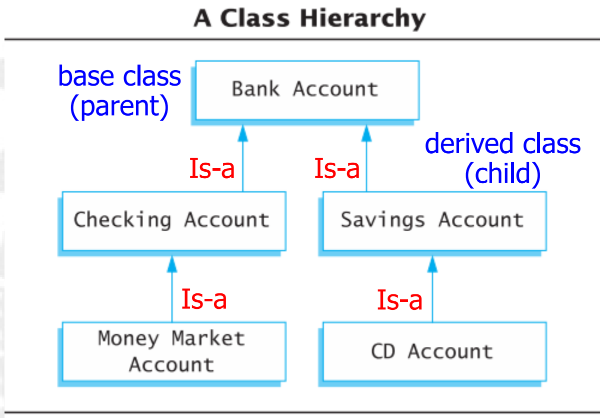
- In OOP, a class is used to represent a concept
  - Polygon, rectangle, ellipse, circle, shape, …
- Concepts don't exist in isolation; they are related
  - Rectangle is a special kind of polygon
  - Circle is a special kind of ellipse
  - They are all shapes
- The same concepts can be inherited from parents → no need to re-write them again
  - Reduce reuse efforts
- Inheritance is one of the key feature of OOP
  - Express such hierarchical relationships
  - Base class vs. derived class (e.g. polygon vs. rectangle)

Inheritance Basics

- Inheritance is the process by which a new class is created from another class
- If class D is inherited from class B
  - The more specific class (D) is a **derived** or **child** class
  - The more general class (B) is the **base**, **super**, or **parent** class
- A derived class automatically has all the member variables and functions of the base class
  - A derived class can add its own member variables and/or member functions
  - **Cannot** be accessed by its parent class

Inheritance Example

- Natural hierarchy of bank accounts
- Most general: A Bank Account stores a balance
- A Checking Account "IS A" Bank Account that allows customers to write checks
- A Savings Account "IS A" Bank Account without checks but higher interest

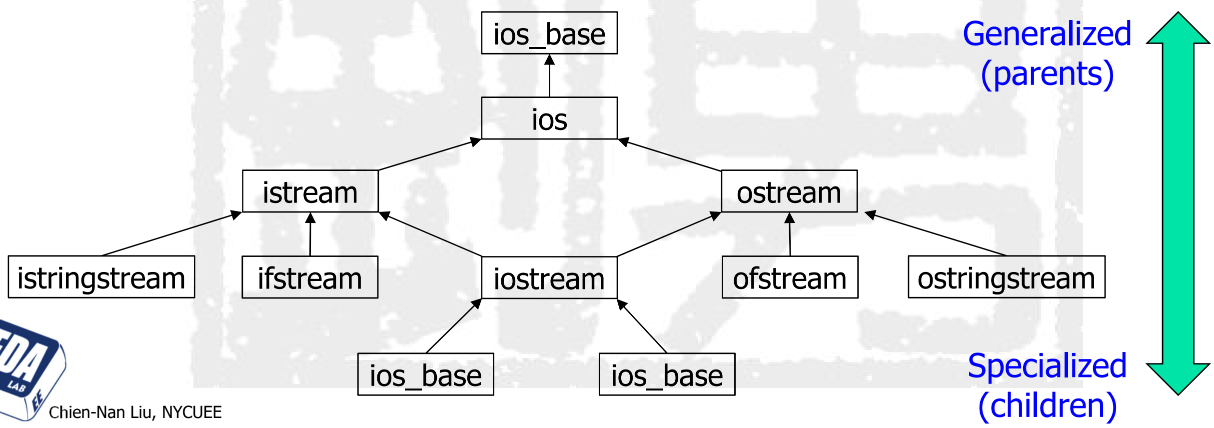


Accounts are more specific as we go down the hierarchy

Each box can be a class

Class Hierarchy

- A derived class can be the base class of another derived class
  class Employee { /* … */ };
  class Manager : public Employee { /* … */ };
  class Director : public Manager { /* … */ };
- Another example: I/O stream



Example: Employee Classes

- To design a record-keeping program with records for salaried and hourly employees…
  - Salaried and hourly employees are all employees → share some common property in "employee" class
- All employees have a name and SSN
  - Functions to manipulate name and SSN are the same for hourly and salaried employees → inheritance
- Different-type employees have different pays
  - Salaried employees is a subset of employees with a fixed wage
  - Hourly employees is another subset of employees who earn hourly wages

A Base Class

- Define a class (Employee) for all employees
  - The base class
- The Employee class will be used to define hourly and salaried employees
  - Two derived classes:

    -- HourlyEmployee

    -- SalariedEmployee

```
class Employee
{
 public:
   Employee( );
   Employee(string theName, string theSSN);
   string getName( ) const;
   string getSSN( ) const;
   double getNetPay( ) const;
   void setName(string newName);
   void setSSN(string newSSN);
   void setNetPay(double newNetPay);
   void printcheck( ) const;
 private:
   string name;
   string ssn;
   double netPay;
};
```

Code for Employee Class (1/2)

Employee.h

```
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

#include <string>
using namespace std;

namespace employeessavitch
{
    class Employee
    {  /* see previous page */ };
} //employeessavitch

#endif //EMPLOYEE_H
```

Employee.cpp

```
#include <string>
#include <cstdlib>
#include <iostream>
#include "employee.h"
using namespace std;

namespace employeessavitch
{
    Employee::Employee( ) : name("No name yet"),
                        ssn("No number yet"), netPay(0)
    {
        //deliberately empty
    }

    Employee::Employee(string theName, string
                                    theNumber)
      : name(theName), ssn(theNumber), netPay(0)
    {
        //deliberately empty
    }
```


Code for Employee Class (2/2)

```
string Employee::getName( ) const
{
    return name;
}

string Employee::getSSN( ) const
{
    return ssn;
}

double Employee::getNetPay( ) const
{
    return netPay;
}

void Employee::setName(string newName)
{
    name = newName;
}
```

```
void Employee::set_ssn(string newSSN)
{
    ssn = newSSN;
}

void Employee::set_netPay (double newNetPay)
{
    netPay = newNetPay;
}

void Employee::printCheck( ) const
{
    cout << "\nERROR: printCheck FUNCTION "
         << "CALLED FOR AN \n"
         << "UNDIFFERENTIATED EMPLOYEE. "
         << "Aborting the program.\n"
         << "Check the program for this bug.\n";
    exit(1);
}

} //employeessavitch
```

Class HourlyEmployee

- HourlyEmployee is derived from Class Employee
  - Inherits all member functions and member variables of Employee
- class HourlyEmployee : public Employee
  - **:public Employee** shows that it is derived from class Employee
  - Declares additional member variables *wageRate & hours*

```
class HourlyEmployee : public Employee
{
 public:
   HourlyEmployee( );
   HourlyEmployee(string theName,
                  string theSSN,
                  double theWageRate,
                  double theHours);
   void setRate(double newWageRate);
   double getRate( ) const;
   void setHours(double hoursWorked);
   double getHours( ) const;
   void printCheck( );
 private:
   double wageRate;
   double hours;
};
```


Public Inheritance

- Public inheritance models "is-a" relationship
  - A derived class inherits all the members of the parent class
  - The derived class does not re-declare or re-define members inherited from the parent
    - Redefines member functions will induce a different definition in the derived class
- The derived class can add member variables and functions
  - The added member functions should be defined in the implementation file for the derived class

```
string name;
string SSN;
double netPay;
……

double wageRate;
double hours;
……
```

Code for HourlyEmployee (1/2)

```
#include <string>
#include <iostream>
#include "hourlyemployee.h"
using namespace std;

namespace employeessavitch
{
    HourlyEmployee::HourlyEmployee( ) :
            Employee( ), wageRate(0), hours(0)
    {
        //deliberately empty
    }

    HourlyEmployee::HourlyEmployee(string
            theName, string theNumber, double
            theWageRate, double theHours) :
            Employee(theName, theNumber),
            wageRate(theWageRate),
            hours(theHours)
    {
        //deliberately empty
    }
```

```
void HourlyEmployee::setRate(double
                                newWageRate)
{
    wageRate = newWageRate;
}

double HourlyEmployee::getRate( ) const
{
    return wageRate;
}

void HourlyEmployee::setHours(double
                                hoursWorked)
{
    hours = hoursWorked;
}

double HourlyEmployee::getHours( ) const
{
    return hours;
}
```


Code for HourlyEmployee (2/2)

```
void HourlyEmployee::printCheck( )   // different to the base class
  {
      setNetPay(hours * wageRate);

      cout << "\n________________________________________________\n";
      cout << "Pay to the order of " << get_name( ) << endl;
      cout << "The sum of " << getNetPay( ) << " Dollars\n";
      cout << "________________________________________________\n";
      cout << "Check Stub: NOT NEGOTIABLE\n";
      cout << "Employee Number: " << getSSN( ) << endl;
      cout << "Hourly Employee. \nHours worked: " << hours
           << " Rate: " << wageRate << " Pay: " << getNetPay( ) << endl;
      cout << "________________________________________________\n";
  }


} //employeessavitch
```

Class SalariedEmployee

- SalariedEmployee is also derived from Employee
- Function *printCheck* is redefined here
  - Have a specific meaning to salaried employees
- SalariedEmployee adds a member variable *salary*
  - Fixed weekly wage

```
class SalariedEmployee : public Employee
{
 public:
   SalariedEmployee( );
   SalariedEmployee (string theName, string
             theSSN, double theWeeklySalary);
   double getSalary( ) const;
   void setSalary(double newSalary);
   void printCheck( );
 private:
   double salary;   //weekly
};
```


Code for SalariedEmployee (1/2)

```
#ifndef SALARIEDEMPLOYEE_H
#define SALARIEDEMPLOYEE_H

#include <iostream>
#include <string>
#include "employee.h"
using namespace std;

namespace employeessavitch
{
   class SalariedEmployee : public Employee
   {  /* see previous page */ };
} //employeessavitch

#endif //SALARIEDEMPLOYEE_H
```

salariedemployee.h

```
#include "salariedemployee.h"
using namespace std;

namespace employeessavitch
{
   SalariedEmployee::SalariedEmployee( ) :
                        Employee( ), salary(0)
   {  /*deliberately empty*/ }

   SalariedEmployee::SalariedEmployee(string
           theName, string theNumber, double
           theWeeklySalary) : Employee(theName,
           theNumber), salary(theWeeklySalary)
   {  /*deliberately empty*/ }

   double SalariedEmployee::getSalary( ) const
   {
      return salary;
   }
```

Code for SalariedEmployee (2/2)

```
void SalariedEmployee::setSalary(double newSalary)
{
    salary = newSalary;
}

void SalariedEmployee::printCheck( )   // different to the base class
{
    setNetPay(salary);
    cout << "\n______________________________________________\n";
    cout << "Pay to the order of " << getName( ) << endl;
    cout << "The sum of " << getNetPay( ) << " Dollars\n";
    cout << "_____________________________________________\n";
    cout << "Check Stub NOT NEGOTIABLE \n";
    cout << "Employee Number: " << getSSN( ) << endl;
    cout << "Salaried Employee. Regular Pay: "
         << salary << endl;
    cout << "_____________________________________________\n";
}
} //employeessavitch
```


Test the Class Inheritance

```
#include <iostream>
#include "hourlyemployee.h"
#include "salariedemployee.h"
using std::cout;
using std::endl;
using namespace employeessavitch;

int main( )
{
    HourlyEmployee joe;
    joe.setName("Mighty Joe");
    joe.setSSN("123-45-6789");
    joe.setRate(20.50);
    joe.setHours(40);
    cout << "Check for " << joe.getName( )
     << " for " << joe.getHours( ) << " hours.\n";
    joe.printCheck( );
    cout << endl;

    SalariedEmployee boss("Mr. Big Shot", "987-
                          65-4321", 10500.50);
    cout << "Check for " << boss.getName( )
         << endl;
    boss.printCheck( );

    return 0;
}
```

Sample Dialogue

```
Check for Mighty Joe for 40 hours.
_________________________________________
Pay to the order of Mighty Joe
The sum of 820 Dollars
_________________________________________
Check Stub: NOT NEGOTIABLE
Employee Number: 123-45-6789
Hourly Employee.
Hours worked: 40 Rate: 20.5 Pay: 820
_________________________________________
Check for Mr. Big Shot
_________________________________________
Pay to the order of Mr. Big Shot
The sum of 10500.5 Dollars
_________________________________________
Check Stub NOT NEGOTIABLE
Employee Number: 987-65-4321
Salaried Employee. Regular Pay: 10500.5
_________________________________________
```

Constructors in Derived Class

- Although a child class inherits all the member of its parent class, the constructor is not inherited
  - The base class constructor can be invoked by the constructor of the derived class at initialization section
  - Besides the parent's constructor, you may also add extra initialization operations in child's constructor

```
HourlyEmployee::HourlyEmployee : Employee( ),
                                 wageRate(0), hours(0)
{ //add code if needed }
```

You can specify any Employee constructor here

Order of Constructors

- If a derived class does not invoke a base class constructor explicitly, use the default constructor
- Assume class B is derived from class A, and class C is derived from class B. When creating an object of class C:
  - The base class A's constructor is the first invoked
  - Class B's constructor is invoked next
  - C's constructor completes execution
- Class objects are destroyed in reverse order of construction
  - Destructors are not inherited, either …

Demo the Order of Constructors

```
class A {
    A( ) { cout << "ctor A" << endl; }
    ~A( ) { cout << "dtor A" << endl; }
};
class B {
    B( ) { cout << "ctor B" << endl; }
    ~B( ) { cout << "dtor B" << endl; }
};
class C : public B {
    A a;
    C( ) { cout << "ctor C" << endl; }
    ~C( ) { cout << "dtor C" << endl; }
};

int main() {
    C c;
    return 0;
};
```

Output:

ctor B	parent
ctor A	local variable
ctor C	object itself
dtor C	reverse order
dtor A	
dtor B	

Redefinition of Member Functions

- When defining a derived class, you only have to list the inherited functions that you wish to change
  - HourlyEmployee and SalariedEmployee each have their own definitions of printCheck
- The parent's class function can still be used, even though a new version is defined in a derived class
  - To specify that you want to use the base class version of the redefined function:

```
HourlyEmployee sallyH;
sallyH.printCheck( );          // new version in child's class
sallyH.Employee::printCheck( );   // original version of parent
```

Redefining or Overloading

- A redefined function in a derived class has the same number and type of parameters
  - The derived class has only one function with the same name as the base class
- An overloaded function has a different number and/or type of parameters than the base class
  - The derived class has two functions with the same name as the base class
    - One is defined in the base class, one in the derived class

Overview

- 7.1 Inheritance Basics
- *7.2 More about Inheritance*
- 7.3 Polymorphism

Inheritance Details

- Some special functions are, for all practical purposes, not inherited by a derived class
  - Copy constructors
  - The assignment operator
  - Destructors
- If those special functions are not defined in derived classes, C++ will generate a default version
  - Often do nothing in the default version
  - The original functions in the base class are not used !!
- If there are pointers and dynamic variables in the base class, special handling is required …
  - You should define your own versions in derived classes

The Assignment (=) Operator

- In implementing an overloaded assignment operator in a derived class:
  - Use the assignment operator from the base class
    - It is written as a member function of the class
  - Assign the member variables introduced in the derived class

Ex:
```
Derived& Derived::operator= (const Derived& rhs)
{
    Base::operator=(rhs)
    ……
```

  - This line handles the assignment of the inherited member variables by calling the base class assignment operator
  - The remaining code handles the new members in the derived class

The Copy Constructor

- Default copy constructor only copies the contents of member variables
  - Not working for pointers and dynamic variables
- Implementing the correct copy constructor in a derived class is similar to that for = operator:

```
Ex: Derived::Derived(const Derived& object)
              :Base(object), <other initializing>
    {…}
```

  - Invoking the copy constructor of its base class sets up the inherited member variables
    - Since object is of type Derived, it is also of type Base

Destructors in Derived Classes

- If the base class has a destructor, defining the destructor for the derived class is relatively easy
  - When the destructor for a derived class is called, the destructor for the base class is automatically called
  - The derived class destructor only need to delete the dynamic variables added in the derived class
- Assume class B is derived from class A, and class C is derived from class B. When an object of class C goes out of scope …
  - The destructor of class C is called
  - Then the destructor of class B
  - Then the destructor of class A

destructors are called in the reverse order of constructor calls

The Protected Qualifier

- The private member variable/function in the parent class is still not accessible to the child class
  - The parent class member functions must be used to access the private members of the parent

```
void HourlyEmployee::printCheck( )
{
      netPay = hours * wageRate;
```

    - netPay is a private member of Employee!
- What if the member variables *name, netPay, and ssn* are listed as **protected** (not private) in the parent
  - Protected members appear to be private outside the class, but are accessible by derived classes
  - This illegal code becomes legal !!

Access Control of Members

- The members of a class can be private, protected, or public
  - Apply to both data members and member functions
- Public members
  - Its name can be used by any functions
- Private members
  - Its name can be used only by member functions and friends of the class in which it declared
- Protected members
  - Its name can be used only by member functions and friends of the class in which it declared, and
  - by member functions and friends of classes derived from this class

Demo the Protected Members

```
class B {
 private:
   int b_priv;
 protected:
   void b_prot( );
 public:
   void b_pub( );
};

class D : public B {
 public:
   void d_pub( );
};
```

```
void D::d_func( ) {   // D's member function
    b_priv = 1;   // Error!! B's private member
    b_prot( );     // OK. Child can access protected member
    b_pub( );      // OK. Allowed in any function
    ……
};

void func(B& b) {   // A global function
    b.b_priv = 1;    // Error!! B's private member
    b.b_prot( );      // Error!! B's protected member
    b.b_pub( );       // OK. Allowed in any function
    ……
};
```

- It's not a good idea to have protected data members
  - Difficult to trace if you have many descendants (sons, sons of sons, sons of sons of sons, …)
- Sometimes useful to have protected member functions

Access Control of Base Classes

- Like a member, a base class can be declared as private, protected, or public

```
class X : public B { /* … */ };          // public inheritance
class Y : protected B { /* … */ };    // protected inheritance
class Z : private B { /* … */ };        // private inheritance
```

Member in base class	Type of Inheritance		
	public	**protected**	**private**
public	public	protected	private
protected	protected	protected	private
private	no access	no access	no access

- Public inheritance models "is-a" relationship
  - This is the most common form of inheritance

Is-a vs. Has-a

- "Is-a" relationship is modeled by public inheritance
- "Has-a" relationship is modeled through composition
  - Class (e.g. Student) has an object from another class (e.g. Date) as a data member
  - Ex: class Student {
    ```
    string name;
    Date birthday;
    ……
    ```
  - Student "is a" Date;   // Wrong !!
    Student "has a" Date called birthday; // Composition
- Both encourage software reuse …

Software Engineering with Inheritance

- Classes are often closely related
  - "Factor out" common attributes and behaviors and place these in a base class
  - Use inheritance to form different derived classes
- If modifications to a base class are necessary
  - Derived classes do not change as long as the public and protected interfaces are the same
  - However, derived classes may need to be recompiled
- Multiple inheritance is allowed in C++, but not encouraged to use until you are an expert
  - A class has more than one direct base classes
    → easy to confuse

Case Study: Point and Circle

- Circle class is derived from the Point class
  - Parent class shares its members in protected section

```
#ifndef POINT2_H
#define POINT2_H
#include <iostream>
using std::ostream;

class Point {
   friend ostream &operator<<(ostream &,
                              const Point &);
 public:
   Point(int=0, int=0);   // default constructor
   void setPoint(int, int);  // set coordinates
   int getX() const { return x; }  // get x value
   int getY() const { return y; }  // get y value
 protected:    // accessible to derived classes
   int x, y;      // coordinates of the point
}; // end class Point
#endif
```

```
#ifndef CIRCLE2_H
#define CIRCLE2_H
using std::ostream;
#include "point2.h"

class Circle : public Point {
   friend ostream &operator<<(ostream &,
                              const Circle & );
 public:
   Circle(double r=0.0, int x=0, int y=0);
   void setRadius( double );     // set radius
   double getRadius() const;     // return radius
   double area() const;          // calculate area
 protected:       // accessible to derived classes
   double radius;    // radius of the Circle
}; // end class Circle
#endif
```


Code for Point Class

point2.cpp

```
#include "point2.h"

// Constructor for class Point
Point::Point( int a, int b ) { setPoint( a, b ); }

// Set the x and y coordinates
void Point::setPoint( int a, int b )
{
    x = a;
    y = b;
} // end function setPoint

// Output the Point
ostream &operator<<( ostream &output,
                     const Point &p )
{
   output << '[' << p.x << ", " << p.y << ']';
   return output;         // enables cascading
} // end operator<< function
```

application.cpp

```
#include <iostream>
using std::cout;
using std::endl;
#include "point2.h"

int main()
{
   Point p( 72, 115 );  // instantiate Point object p

    // protected data of Point inaccessible to main
    // access protected data through member func
   cout << "X coordinate is " << p.getX()
           << "\nY coordinate is " << p.getY();
   p.setPoint( 10, 10 );
   cout << "\n\nThe new location of p is " << p;
   cout << endl;
   return 0;
} // end function main
```

program output

```
X coordinate is 72
Y coordinate is 115

The new location of p is [10, 10]
```

Code for Circle Class

circle.cpp

```
#include <iomanip>
using std::ios;
using std::setiosflags;
using std::setprecision;
#include "circle2.h"

Circle::Circle( double r, int a, int b )
   : Point( a, b )  // call base-class constructor
{ setRadius( r ); }  // initialize its own variable

void Circle::setRadius( double r )
{ radius = ( r >= 0 ? r : 0 ); }

double Circle::getRadius() const
{ return radius; }

double Circle::area() const
{ return 3.14159 * radius * radius; }
```

```
ostream &operator<<( ostream &output,
                     const Circle &c )
{
  output << "Center = " << static_cast<Point> (c)
         << "; Radius = "
         << setiosflags(ios::fixed |
                        ios::showpoint )
         << setprecision( 2 ) << c.radius;
  return output;   //  enables cascaded calls
} // end operator<< function
```

Get the inherited part in object c and turn its type to Class point

Test the Circle Class

```
#include <iostream>
using std::cout;
using std::endl;
#include "point2.h"
#include "circle2.h"
int main()
{
   Circle c( 2.5, 37, 43 );
   cout << "X coordinate is " << c.getX()
          << "\nY coordinate is " << c.getY()
          << "\nRadius is " << c.getRadius();
   c.setRadius( 4.25 );
   c.setPoint( 2, 2 );
   cout << "\n\nThe new location and radius of c are\n"
          << c << "\nArea " << c.area() << '\n';
   Point &pRef = c;
   cout << "\nCircle printed as a Point is: " << pRef << endl;
   return 0;
} // end function main
```

program output

Program output
X coordinate is 37
Y coordinate is 43
Radius is 2.5

The new location and radius of c are
Center = [2, 2]; Radius = 4.25
Area 56.74

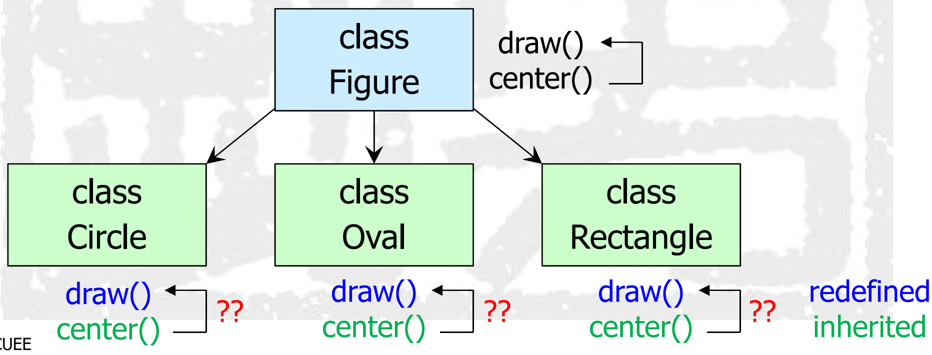
Circle printed as a Point is: [2, 2]

Overview

- 7.1 Inheritance Basics
- 7.2 More about Inheritance
- *7.3 Polymorphism*

A Binding Issue in Inheritance

- Imagine a program with several types of figures
  - Each figure may be an object of a different class, such as a circle, oval, rectangle, etc.
  - Figure is the base class, others are derived classes
  - Each has a function draw( ) specific to each shape
  - Class Figure has a function center( ) inherited to all figures
    - It calls function draw( ) to redraw the figure at the center
    - But which draw( ) is called in the derived classes??



Polymorphism

- If you don't know how the function is implemented, tell the compiler to wait until the function is used
  - Get the correct version based on the actual calling object
  - This mechanism is called **late binding**
  - The unbound function is specified as a **virtual function**
- Polymorphism refers to the ability to associate multiple meanings with one function
  - In C++, polymorphism is achieved through virtual functions, and
  - manipulating objects through pointers or references
- Polymorphism is a key component of the philosophy of object oriented programming

Virtual Functions in C++

- 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
    - Only non-static functions can be made virtual
  - *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

An Example w/o Virtual Function

```
class Employee {
  // data members are omitted
 public:
  void print() const;  // Employee::print()
  ......
}

void Employee::print() const {
  cout << name << '\t' << depar << endl;
}

class Manager : public Employee {
  // data members are omitted
 public:
  void print() const;  // Manager::print()
  ......
}
```

Manager redefines Employee's print()

```
void Manager::print() const {
  Employee::print();   // Manager is also an Employee
  cout << "Level: " << level << endl;
}

void f() {
  Employee ta("TA", 3);
  Manager jim("Jimmy", 3, 1);
  ta.print();              // use Employee::print()
  jim.print();             // use Manager::print()

  Employee *pe = &ta;
  Manager *pm = &jim;
  pe->print();             // use Employee::print()
  pm->print();             // use Manager::print()
  pe = &jim;               // OK. jim is also an Employee
  pe->print();             // use Employee::print()
}                          // but jim is a manager …
```


With Virtual Function, It Becomes …

```
class Employee {
  // data members are omitted
 public:
  virtual void print() const;  // always virtual
  ......                       // in all derived
}                              // classes
       no "virtual" here !!
void Employee::print() const {
  cout << name << '\t' << depar << endl;
}

class Manager : public Employee {
  // data members are omitted
 public:
  void print() const;  // Manager::print()
  ......
}
```

Manager class is unchanged at all

```
void Manager::print() const {
  Employee::print();   // Manager is also an Employee
  cout << "Level: " << level << endl;
}

void f() {
  Employee ta("TA", 3);
  Manager jim("Jimmy", 3, 1);
  ta.print();              // use Employee::print()
  jim.print();             // use Manager::print()

  Employee *pe = &ta;
  Manager *pm = &jim;
  pe->print();             // use Employee::print()
  pm->print();             // use Manager::print()
  pe = &jim;               // OK. jim is also an Employee
  pe->print();             // use Manager::print()
}                          // It knows what pe point to …
```

Virtual vs. Non-Virtual Functions

- For non-virtual (member) functions
  - Function calls are statically bound (i.e. bound at compile time)
- For virtual (member) functions
  - Function calls are dynamically bound (i.e. bound at runtime)

```
class Employee {
  // data members are omitted
 public:
  void print1() const;
  virtual void print2() const;
}
```

```
class Manager : public Employee {
  // data members are omitted
 public:
  void print1() const;   // non-virtual, override
  void print2() const;   // virtual, late binding
}
```

```
void f() {
  Employee ta("TA", 3), *pe = &ta;
  Manager jim("Jimmy", 3, 1), *pm = &jim;
  pm->print1();        // static binding, use Manager::print1()
  pe->print2();        // dynamic binding, use Employee::print2()
  pe = &jim;           // pe is now pointing to Manager
  pe->print1();        // still static binding, use Employee::print1()
  pe->print2();        // dynamic binding, use Manager::print2()
}
```


Another Example: Auto Parts Store

- We want a versatile program for record-keeping in an auto parts store
  - But we do not know all the possible types of sales we might have to account for at this moment …
  - 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 Base Class – Sale Class

- All sales will be derived from the base class Sale
- The bill function of the Sale class is virtual
  - Determined later based on the type of calling object
- Both the member function savings and operator < use this virtual function
  - Changed automatically when the virtual function is bound to a specific type

```
class Sale
{
 public:
    Sale();
    Sale(double thePrice);
    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.
```


Code for Sale Class

sale.h

```
#ifndef SALE_H
#define SALE_H

#include <iostream>
using namespace std;

namespace salesavitch
{
   class Sale
   {  /* see previous page */ };
} //salesavitch

#endif //SALE_H
```

sale.cpp

```
#include "sale.h"

namespace salesavitch
{
   Sale::Sale() : price(0) { /* empty */ }

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

   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
```

The Derived Class -- DiscountSale

- Derived class has its own version of virtual function *bill*
- When a DiscountSale object calls its savings function (inherited from the Sale class)

→ Sale::savings( ) use the function bill from the DiscountSale class

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

```
class DiscountSale : public Sale
{
 public:
    DiscountSale();
    DiscountSale(double thePrice,
                 double theDiscount);
    //Discount is a percent of the price
    virtual double bill() const;
 protected:
    double discount;
};
```


Code for DiscountSale Class

```
#ifndef DISCOUNTSALE_H
#define DISCOUNTSALE_H
#include "sale.h"

namespace salesavitch
{
    class DiscountSale : public Sale
    {  /* see previous page */ };
} //salesavitch

#endif //DISCOUNTSALE_H
```

discountsale.h

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

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

    DiscountSale::DiscountSale(double thePrice, double
        theDiscount) : Sale(thePrice), discount(theDiscount)
    { /* empty */ }

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

} //salesavitch
```

discountsale.cpp

Test Sale & DiscountSale

```
#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."
                << endl;

    return 0;
}
```

Sample Dialogue

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


Redefine vs. Override

- When a derived class D modifies the definition of an inherited non-virtual member function mf
  - We say class D **redefines** mf, or mf is redefined in D

```
class B { public: void mf(); }
class D: public B { public: void mf(); }  // D redefines mf()
```

- When a derived class D modified the definition of a virtual member function mf inherited from class B
  - We say D::mf **overrides** B::mf, or B::mf is overridden by D::mf

```
class B { public: virtual void mf(); }      // D::mf() overrides
class D: public B { public: void mf(); }  // B::mf()
```

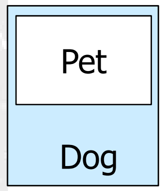
- Fundamental concepts are different between them

Type Checking in Inheritance

- 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, e.g. double d=2.5; int a = d; → a will become 2, not 2.5
- It is legal to assign a derived class object into a base class variable, but some info will be truncated
- Ex:

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

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



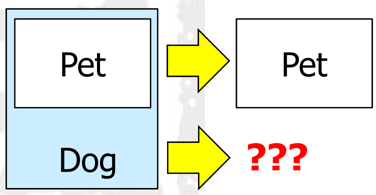
A Sliced Dog is a Pet

- C++ allows the following assignments:

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

- However, vpet will loose the breed member of vdog since this member does not exist in class Pet
  - This code would be illegal:

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



- This is called the slicing problem
- What if you assign a base class object into a derived class variable??

Dynamic Variables & Derived Classes

- It is possible in C++ to avoid the slicing problem
  - Using pointers to dynamic variables !!
  - Will not lose members of the object because you only transfer the memory address, not actual object
- Ex:

```
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

Use Virtual Functions

- In this example, ppet->print( ) worked because print was declared as a virtual function by class Pet
  - The computer checks the virtual table for classes Pet and Dog and finds that ppet points to an object of type Dog
  - Since ppet points to a Dog object, Dog::print( ) is used
- This code would still produce an error:

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

  - ppet is a pointer to a Pet object that has no breed member
- If p_ancestor is a pointer to the base class and p_descendant is a pointer to the derived class
  - p_ancestor = p_descendant is allowed without losing info
  - However, virtual functions are required to access members

Avoid Hiding Inheritance Names (1/2)

```
class Base {
 public:
   virtual void mf1();     // pure virtual function
   virtual void mf1(int); //overload virtual function
   virtual void mf2();    // simple virtual function
   void mf3();            // non-virtual member function
   void mf3(double);      // overloaded non-virtual function
};

class Derived : public Base { public:
   virtual void mf1();   // override Base::mf1
   void mf3();           // redefine mf3
}

void f() {
   Derived d;
   d.mf1();              // OK. Call Derived::mf1()
   d.mf1(10);            // Surprising error!!
   d.mf2();              // OK. Call Base::mf2()
   d.mf3();              // OK. Call Derived mf3()
   d.mf3(10.0);          // Surprising error!!
}
```

Base::mf1() is hidden. Number of parameters does not match to Derived::mf1()

Base::mf3() is hidden. Number of parameters does not match to Derived::mf3()

Avoid Hiding Inheritance Names (2/2)

```
class Base {
  // same as in previous page …
};

class Derived : public Base {
 public:
   using Base::mf1;      // make all mf1 in Base visible in Derived
   virtual void mf1();   // override Base::mf1() only
   void mf3();           // redefine mf3
}

void f() {
   Derived d;
   d.mf1();              // OK. Call Derived::mf1()
   d.mf1(10);            // OK now. Call Base::mf1(int)
   d.mf2();              // OK. Call Base::mf2()
   d.mf3();              // OK. Call Derived mf3()
   d.Base::mf3(10.0); // OK now. Call Base::mf3(double) explicitly
}
```

Clearly specify the function to call.

parameters does not match to mf1(). Call Base::mf1(int).

Virtual Destructors

- Destructors should be made virtual
  - Consider
    ```
    Base *pBase = new Derived;
          …
    delete pBase;
    ```
  - If the destructor in Base is virtual, the destructor for Derived is invoked as pBase points to a Derived object
    - Returning all members in Derived, including the member inherited from Base, to the freestore
- If the Base destructor is not virtual, only the Base destructor is invoked
  - This leaves Derived members, which are not a part of Base, in memory

Demo the Virtual Destructor

```
class Base {
 public:
    ~Base();            // non-virtual dtor
    ……                  // other stuffs
};
class Derived : public Base
{ /* add some members… */ }
void f() {
    Base *pB = new Derived;
    // OK. Call Derived's ctor
    ……
    delete pB;  // Disaster!! Call Base's dtor
};              // since it is non-virtual
```

```
class Base {
 public:
    virtual ~Base();   // virtual dtor
    ……                 // other stuffs
};
class Derived : public Base
{ /* add some members… */ }
void f() {
    Base *pB = new Derived;
    // OK. Call Derived's ctor
    ……
    delete pB;  // OK. Call Derived's dtor
};              // since it is virtual
```

- Declare destructors virtual in polymorphic base classes
  - Otherwise, you may call the wrong destructor and return wrong size of memory
- Don't blindly declare destructors virtual in all classes
  - Incur memory and runtime overhead → there's no free lunch