OOP/Computer Programming

Instructor: Dr. Danish Shehzad

Friend Function Example

- class RectangleOne
- {
- int L,B;
- public:
- RectangleOne(int l,int b)
- {
- L = l;
- B = b;
- }
- friend void Sum(RectangleOne, RectangleTwo);
- };

Example

- class RectangleTwo
- {
- int L,B;
- public:
- RectangleTwo(int l,int b)
- {
- L = l;
- B = b;
- }
- friend void Sum(RectangleOne, RectangleTwo);
- };

```
void Sum(RectangleOne R1,RectangleTwo R2)
{
        cout<<"\n\t\tLength\tBreadth";
        cout<<"\n Rectangle 1  :   "<<R1.L<<"\t  "<<R1.B;
        cout<<"\n Rectangle 2  :   "<<R2.L<<"\t  "<<R2.B;
        cout<<"\n ------------------------------";
        cout<<"\n\tSum   :   "<<R1.L+R2.L<<"\t
"<<R1.B+R2.B;
        cout<<"\n ------------------------------";
}
```

```
void main()
	{
		RectangleOne Rec1(5,3);
		RectangleTwo Rec2(2,6);
		Sum(Rec1,Rec2);
	}
```

Friend class

FRIEND CLASS IN C++ PROGRAMMING

- Similarly, like a friend function, a class can also be made a friend of another class using keyword friend. For example:

```
class B;
class A
{
   // class B is a friend class of class A
   friend class B;
   ... .. ...
}
class B
{
   ... .. ...
}
```

- When a class is made a friend class, all the member functions of that class becomes friend functions.
- In previous program, all member functions of class B will be friend functions of class A. Thus, any member function of class B can access the private and protected data of class A. But, member functions of class A cannot access the data of class B.
- Remember, friend relation in C++ is only granted, not taken.

- For example a LinkedList class may be allowed to access private members of Node.
- class Node
- {
- private:
- int key;
- Node *next;
- /* Other members of Node Class */
-
- friend class LinkedList; // Now class LinkedList can
-                    // access private members of Node
- };

friend Functions and friend Classes

- **`friend`** function and **`friend`** classes
  - Can access **`private`** and **`protected`** members of another class
  - **`friend`** functions are not member functions of class
    - Defined outside of class scope
- Properties of friendship
  - Friendship is granted, not taken
  - Not symmetric (if **`B`** a **`friend`** of **`A`**, **`A`** not necessarily a **`friend`** of **`B`**)

Example

```
#include <iostream>
using namespace std;
class A{
        private:
                int x;
        friend class B;
        public:
                A(){
                        x=20;
                }
};
```

```
class B{
        public:
                void func(A obja){
                cout<<obja.x;
                }
};
int main()
{
        A obja;
        B objb;
        objb.func(obja);
}
```

FRIEND FUNCTIONS AND FRIEND CLASSES

- **`friend`** declarations
  - To declare a **`friend`** function
    - Type **`friend`** before the function prototype in the class that is giving friendship

```
friend int myFunction( int x );
```

    should appear in the class giving friendship
  - To declare a **`friend`** class
  - Type **`friend class Classname`** in the class that is giving friendship
  - if **`ClassOne`** is granting friendship to **`ClassTwo`**,

```
friend class ClassTwo;
```

  - should appear in **`ClassOne`**'s definition

Activity

- Write a program to find maximum out of two numbers using friend function and also note one number is a member of one class and other number is member of some other class[Both variables are private].
- Write a program that has 2 classes called A and B. A has some data-member like x that should be input from a function defined in class B. (Do not use inheritance)

Review — Accessing Members of Base and Derived Classes

```
class B {
public:
  void m();
  void n();
  ...
} // class B


class D: public B {
public
  void m();
  void p();
  ...
} // class D
```

- The following are legal:–

```
B_obj.m() //B's m()
B_obj.n()

D_obj.m() //D's m()
D_obj.n() //B's n()
D_obj.p()


B_ptr->m() //B's m()
B_ptr->n()

D_ptr->m() //D's m()
D_ptr->n() //B's n()
D_ptr->p()
```

Review — Accessing Members of Base and Derived Classes (continued)

```
class B {
public:
   void m();
   void n();
   ...
} // class B

class D: public B {
public
   void m();
   void p();
   ...
} // class D
```

Class **D** *redefines* method **m()**

- The following is legal:–

  B_ptr = D_ptr;

- The following are *not* legal:–

  D_ptr = B_ptr;

  B_ptr->p();

  Even if **B_ptr** *is known to point to an object of class* **D**

Review — Accessing Members of Base and Derived Classes (continued)

- Access to members of a class object is determined by the type of the *handle*.

- Definition: *Handle*
  - The thing by which the members of an object are accessed
  - May be
    - An object name (i.e., variable, etc.)
    - A reference to an object
    - A pointer to an object

Review — Accessing Members of Base and Derived Classes (continued)

- This is referred to as *static binding*

- i.e., the *binding* between handles and members is determined at compile time
  - i.e., statically

What if we need Dynamic Binding?

- What if we need a class in which access to methods is determined at run time by the *type of the object*, not the *type of the handle*

```
class Shape {
public:
   void Rotate();
   void Draw();
   ...
}
```

Need to access the method to draw the right kind of object Regardless of handle!

```
class Rectangle: public Shape {
public:
   void Rotate();
   void Draw();
   ...
}
```

```
class Ellipse: public Shape {
public:
   void Rotate();
   void Draw();
   ...
}
```

SOLUTION – *VIRTUAL FUNCTIONS*

- Define a method as virtual, and the subclass method *overrides* the base class method

- For example

```
class Shape {
public:
  virtual void Rotate();
  virtual void Draw();
  ...
}
```

This tells the compiler to add internal pointers to every object of class **`Shape`** and its derived classes, so that pointers to correct methods can be stored with each object.

What if we need Dynamic Binding?

```
class Shape {
public:
  virtual void Rotate();
  virtual void Draw();
  ...
}
```

```
class Rectangle: public Shape
  {
public:
  void Rotate();
  void Draw();
  ...
}
```

```
class Ellipse: public Shape {
public:
  void Rotate();
  void Draw();
  ...
}
```

- Subclass methods *override* the base class methods
  - (if specified)
- C++ *dynamically* chooses the correct method for the class from which the object was instantiated.

Notes on virtual

- If a method is declared virtual in a class,
  - … it is automatically virtual in *all* derived classes

- It is a really, really good idea to make destructors virtual!

```
virtual ~Shape();
```

  - *Reason:* to invoke the correct destructor, no matter how object is accessed

Virtual Destructors

- Constructors cannot be virtual, but destructors can be virtual.

- It ensures that the derived class destructor is called when a base class pointer is used while deleting a dynamically created derived class object.

Virtual Destructors (contd.)

```
class base {
public:
   ~base() {
      cout <<  "destructing base\n";
   }
};

class derived : public base {
public:
   ~derived() {
      cout << "destructing
     derived\n";
   }
};
```

```
int main()
{
   base *p = new derived;
   delete p;
   return 0;
}
```

Output:

destructing base

Using non-virtual destructor

Virtual Destructors (contd.)

```
class base {

public:
  virtual ~base() {
    cout <<  "destructing base\n";
  }
};

class derived : public base {
public:
  ~derived() {
    cout << "destructing
 derived\n";
  }
};
```

```
int main()
{
  base *p = new derived;
  delete p;

  return 0;
}
```

Output:

    destructing derived
    destructing base

Using virtual destructor

Notes on virtual (continued)

- A derived class may *optionally* override a virtual function
  - If not, base class method is used

```
class Shape {
public:
  virtual void Rotate();
  virtual void Draw();
  ...
}
class Line: public Shape {
public:
  void Rotate();
            //Use base class Draw method
  ...
}
```

Summary – Based and Derived Class Pointers

- Base-class pointer pointing to base-class object
  - *Straightforward*
- Derived-class pointer pointing to derived-class object
  - *Straightforward*
- Base-class pointer pointing to derived-class object
  - Safe
  - Can access non-virtual methods of only base-class
  - Can access virtual methods of derived class
- Derived-class pointer pointing to base-class object
  - *Compilation error*

Abstract and Concrete Classes

- *Abstract Classes*
  - Classes from which it is never intended to instantiate any objects
    - Incomplete—derived classes must define the “missing pieces”.
    - Too generic to define real objects.
  - Normally used as base classes and called *abstract base classes*
    - Provide appropriate base class frameworks from which other classes can inherit.
- *Concrete Classes*
  - Classes used to instantiate objects
  - Must provide implementation for *every* member function they define

Definitions

Pure virtual Functions

- A class is made *abstract* by declaring one or more of its virtual functions to be "pure"
  - By placing **`"= 0"`** in its declaration

- Example

```
virtual void draw() const = 0;
```

  - **`"= 0"`** is known as a *pure specifier*.
  - Tells compiler that there *is no* implementation.

Pure virtual Functions (continued)

- Every *concrete* derived class must override all base-class pure **`virtual`** functions
  - with concrete implementations
- If even one pure virtual function is not overridden, the derived-class will also be *abstract*
  - Compiler will refuse to create any objects of the class
  - Cannot call a constructor

Purpose

- When it does not make sense for base class to have an implementation of a function

- Software design requires *all* concrete derived classes to implement the function
    - Themselves

Why do we want to do this?

- To define a *common public interface* for various classes in a class hierarchy
  - Create framework for abstractions defined in our software system

- The heart of *object-oriented programming*

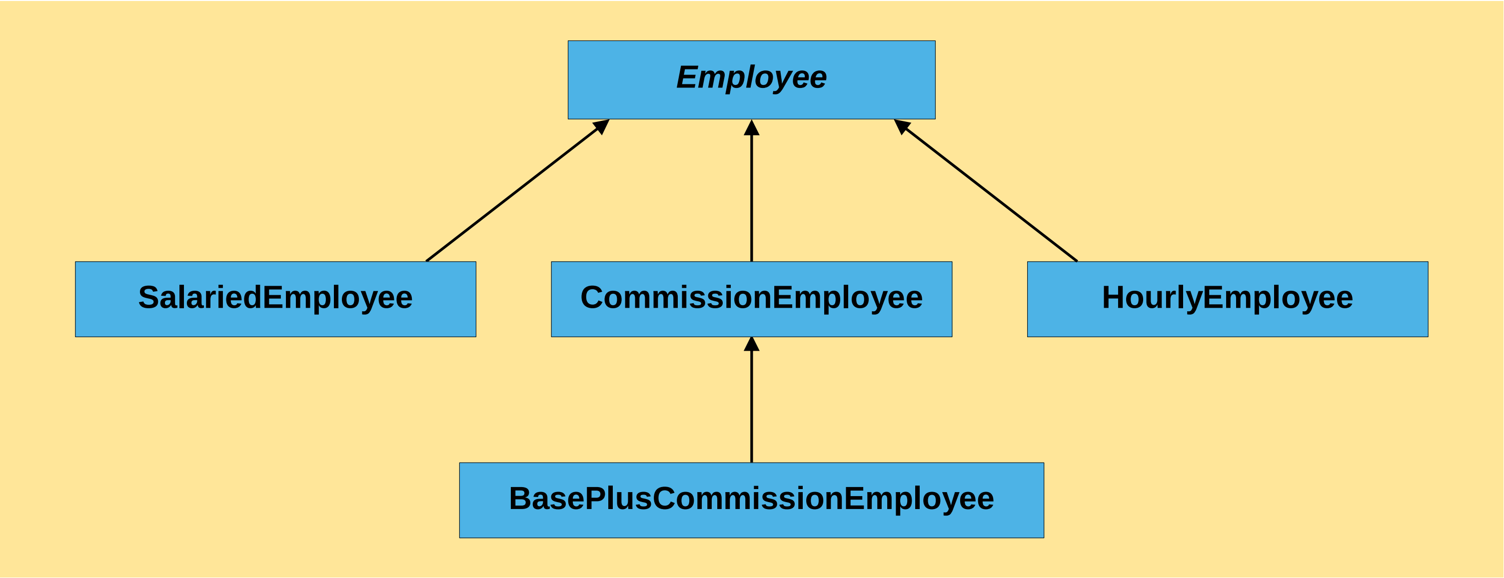
- Simplifies a lot of big software systems
  - Enables code re-use in a major way
  - Readable, maintainable, adaptable code

Case Study: Payroll System Using Polymorphism

- Create a payroll program
  - Use virtual functions and polymorphism
- Problem statement
  - 4 types of employees, paid weekly
    - Salaried (fixed salary, no matter the hours)
    - Hourly workers
    - Commission (paid percentage of sales)
    - Base-plus-commission (base salary + percentage of sales)

Case Study: Payroll System Using Polymorphism

- Base class **`Employee`**
  - Pure virtual function **`earnings`** (returns pay)
    - Pure virtual because need to know employee type
    - Cannot calculate for generic employee
  - Other classes derive from **`Employee`**



Employee Example

```
class Employee {
public:
   Employee(const char *, const char *);
   ~Employee();
          char *getFirstName() const;
          char *getLastName() const;

   // Pure virtual functions make Employee abstract base class.
   virtual float earnings() const = 0; // pure virtual
   virtual void print() const = 0;     // pure virtual

protected:
   char *firstName;
   char *lastName;
};
```

```
Employee::Employee(const char *first, const char *last)
{
        firstName = new char[ strlen(first) + 1 ];
        strcpy(firstName, first);
        lastName = new char[ strlen(last) + 1 ];
        strcpy(lastName, last);
}

// Destructor deallocates dynamically allocated memory
Employee::~Employee() {
  delete [] firstName;   delete [] lastName;
}

// Return a pointer to the first name
char *Employee::getFirstName() const {
return firstName;   // caller must delete memory
}

char *Employee::getLastName() const {
  return lastName;   // caller must delete memory
}
```

```
class SalariedEmployee: public Employee {
public:
        SalariedEmployee(const char *, const char *, float =
0.0);
        void setWeeklySalary(float);
         virtual float earnings() const;
         virtual void print() const;
private:
        float weeklySalary;
};
```

```
// Constructor function for class
SalariedEmployee:: SalariedEmployee(const char *first,
                                    const char *last, float s)
   : Employee(first, last)  // call base-class constructor
{ weeklySalary = s > 0 ? s : 0; }

// Set the SalariedEmployee's salary
void SalariedEmployee::setWeeklySalary(float s)
   { weeklySalary = s > 0 ? s : 0; }

// Get the SalariedEmployee's pay
float SalariedEmployee::earnings() const { return
weeklySalary; }

// Print the SalariedEmployee's name
void SalariedEmployee::print() const
{
   cout << endl << " Salaried Employee: " << getFirstName()
        << ' ' << getLastName();
}
```

```
class CommissionWorker : public Employee {
public:
     CommissionWorker(const char *, const char *, float = 0.0, unsigned =
0);
   void setCommission(float);
   void setQuantity(unsigned);
   virtual float earnings() const;
   virtual void print() const;

private:
   float commission;   // amount per item sold
   unsigned quantity;  // total items sold for week
};
```

```
CommissionWorker::CommissionWorker(const char *first,
        const char *last, float c, unsigned q)
   : Employee(first, last)  // call base-class constructor
{
               commission = c > 0 ? c : 0;
               quantity = q > 0 ? q : 0;
}
void CommissionWorker::setCommission(float c)
   { commission = c > 0 ? c : 0; }
void CommissionWorker::setQuantity(unsigned q)
   { quantity = q > 0 ? q : 0; }
float CommissionWorker::earnings() const
   { return commission * quantity; }
void CommissionWorker::print() const
{
   cout << endl << "Commission worker: " << getFirstName()
        << ' ' << getLastName();
}
```

```
class HourlyWorker : public Employee {
public:
   HourlyWorker(const char *, const char *,
                float = 0.0, float = 0.0);
   void setWage(float);
   void setHours(float);
   virtual float earnings() const;
   virtual void print() const;
private:
   float wage;   // wage per hour
   float hours;  // hours worked for week
};
```

```
HourlyWorker::HourlyWorker(const char *first, const char *last,
                           float w, float h)
   : Employee(first, last)   // call base-class constructor
{
   wage = w > 0 ? w : 0;
   hours = h >= 0 && h < 168 ? h : 0;
}
void HourlyWorker::setWage(float w) { wage = w > 0 ? w : 0; }
// Set the hours worked
void HourlyWorker::setHours(float h)
   { hours = h >= 0 && h < 168 ? h : 0; }
// Get the HourlyWorker's pay
float HourlyWorker::earnings() const { return wage * hours; }
// Print the HourlyWorker's name
void HourlyWorker::print() const
{
   cout << endl << "    Hourly worker: " << getFirstName()
        << ' ' << getLastName();
}
```

```
class BasePlusCommissionEmployee:public
CommissionWorker
{
private:
        float baseSalary;
public:
        BasePlusCommissionEmployee(const char* ,
const char* , float =0.0, unsigned =0,float =0.0);
        void setBaseSalary(float sal)    {
                baseSalary = sal;
        }
        float getBaseSalary(void) const            {
                return baseSalary;
        }
        void print() const;
        float earnings() const;
};
```

```
BasePlusCommissionEmployee::BasePlusCommissionEmployee(const char* first, const char* last, float c,
                    unsigned q,float sal)
                    :CommissionWorker(first,last,c,q)
{
          baseSalary=(sal);
}
void BasePlusCommissionEmployee::print() const
{
          cout << "\nbase-salaried commission employee: ";
          CommissionWorker::print();  // code reuse
} // end function print
float BasePlusCommissionEmployee::earnings() const
{
          return getBaseSalary() + CommissionWorker::earnings();

} // end function earnings
```

```
void main(void)
{
   Employee *ptr;  // base-class pointer

   SalariedEmployee b("Nauman", "Sarwar", 800.00);
   ptr = &b;  // base-class pointer to derived-class object
   ptr->print();                              // dynamic binding
   cout << " earned $" << ptr->earnings(); // dynamic binding
   b.print();                                // static binding
   cout << " earned $" << b.earnings();    // static binding

   CommissionWorker c("Qasim", "Ali", 3.0, 150);
   ptr = &c;  // base-class pointer to derived-class object
   ptr->print();                              // dynamic binding
   cout << " earned $" << ptr->earnings(); // dynamic binding
   c.print();                                // static binding
   cout << " earned $" << c.earnings();    // static binding
```

```
    BasePlusCommissionEmployee p("Mehshan", "Mustafa", 2.5, 200, 1000.0);
    ptr = &p;  // base-class pointer to derived-class object
    ptr->print();                          // dynamic binding
    cout << " earned $" << ptr->earnings(); // dynamic binding
    p.print();                             // static binding
    cout << " earned $" << p.earnings();    // static binding

    HourlyWorker h("Samer", "Tufail", 13.75, 40);
    ptr = &h;  // base-class pointer to derived-class object
    ptr->print();                          // dynamic binding
    cout << " earned $" << ptr->earnings(); // dynamic binding
    h.print();                             // static binding
    cout << " earned $" << h.earnings();    // static binding

    cout << endl;

    return 0;
}
```