Object Oriented Programming

Bilal Khalid Dar

Inheritance III

Friend Functions and Classes

- **friend** is a **function** or **class** that is **not a member of a class**, but **has access to the private members** of the class.
- **Classes** keep a "**list**" of **their friends**, and only the **external functions or classes** whose **names appear in the list** are **granted access**
- **Friend function Syntax (in Class declaration):**

```
friend ReturnType FunctionName (ParameterTypeList)
```

- Friend is a non-member function.
- Friend function can access private and protected data members of a class which declared it as friend function.

- Friend function is inherited without friendship.
  - It can access private and protected data members of Base class because Base declared it as friend. But Derived class does not automatically make this function a friend, to give access to his private and protected data.

Example

```
class A {
protected:
    int x;

public:
    A() { x = 0; }
    friend void show();
};

// Child Class
class B : public A {
private:
    int y;

public:
    B() { y = 0; }
};
```

```
void show()
{
    B b;
    cout << "The default value of A::x = " << b.x;

    // Can't access private member declared in class 'B'
    cout << "The default value of B::y = " << b.y;
}

int main()
{
    show();
    getchar();
    return 0;
}
```

Output

```
prog.cpp: In function ‘void show()’:
prog.cpp:19:9: error: ‘int B::y’ is private
    int y;
        ^
prog.cpp:31:49: error: within this context
    cout << "The default value of B::y = " << b.y;
                                                 ^
```

Friend Classes

- **Not a good approach→ Declare a complete class as a friend:**
  - **All functions of the friend class <u>can access</u> all private members of the current class**
  - **Only a member function which needs access to private members <u>must be allowed</u>**

Syntax:

```
friend class AuxiliaryOffice;
```

```cpp
#include <iostream>
using namespace std;
class First
{
  // Declare a friend class
  friend class Second;
  public:
     First() : a(0){}
     void print()
     {
        cout << "The result is "<< a << endl;
     }
  private:
     int a;
};
class Second
{
  public:
    void change( First& yclass, int x )
    {
        yclass.a = x;
    }
};
```

```cpp
int main()
{
     First obj1;
     Second obj2;
     obj1.print();
     obj2.change( obj1, 5 );
     obj1.print();
}
```

```
//Output
The result is 0
The result is 5
```

Binding Process

- **Binding** is the **process** to **associate names** with **memory addresses**.

- **Binding** is **done** for **each variable** and **functions**.

- **For functions**, it **means** that **matching** the **call** with the **right function definition** by the compiler.

Compile-time Binding (Static Binding)

- **Compile-time binding** is to **associate** a **function's name** with the **entry point** (start memory address) **of the function** at **compile time** **(also called *early binding*)**

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

void sayHi();
int main(){
    sayHi();      // the compiler binds any invocation of sayHi()
                  // to sayHi()'s entry point.
}
void sayHi(){
    cout << ``Hello, World!\n'';
}
```

In C, only compile-time binding is provided

Start address if sayHi() function

Run-time Binding (Dynamic Binding)

- **Run-time binding** is to **associate** a **function's name** with the **entry point** (start memory address) **of the function** at **run time** **(also called *late binding*)**

- **C++ provides both compile-time** and **run-time bindings**:
  - **Non-Virtual functions** (*you have implemented so far*) are **binded** at **compile time**.
  - **Virtual functions** (in C++) are **binded** at **run-time**.

- **Why virtual functions are used?**
  - To implement **Polymorphism**

Polymorphism

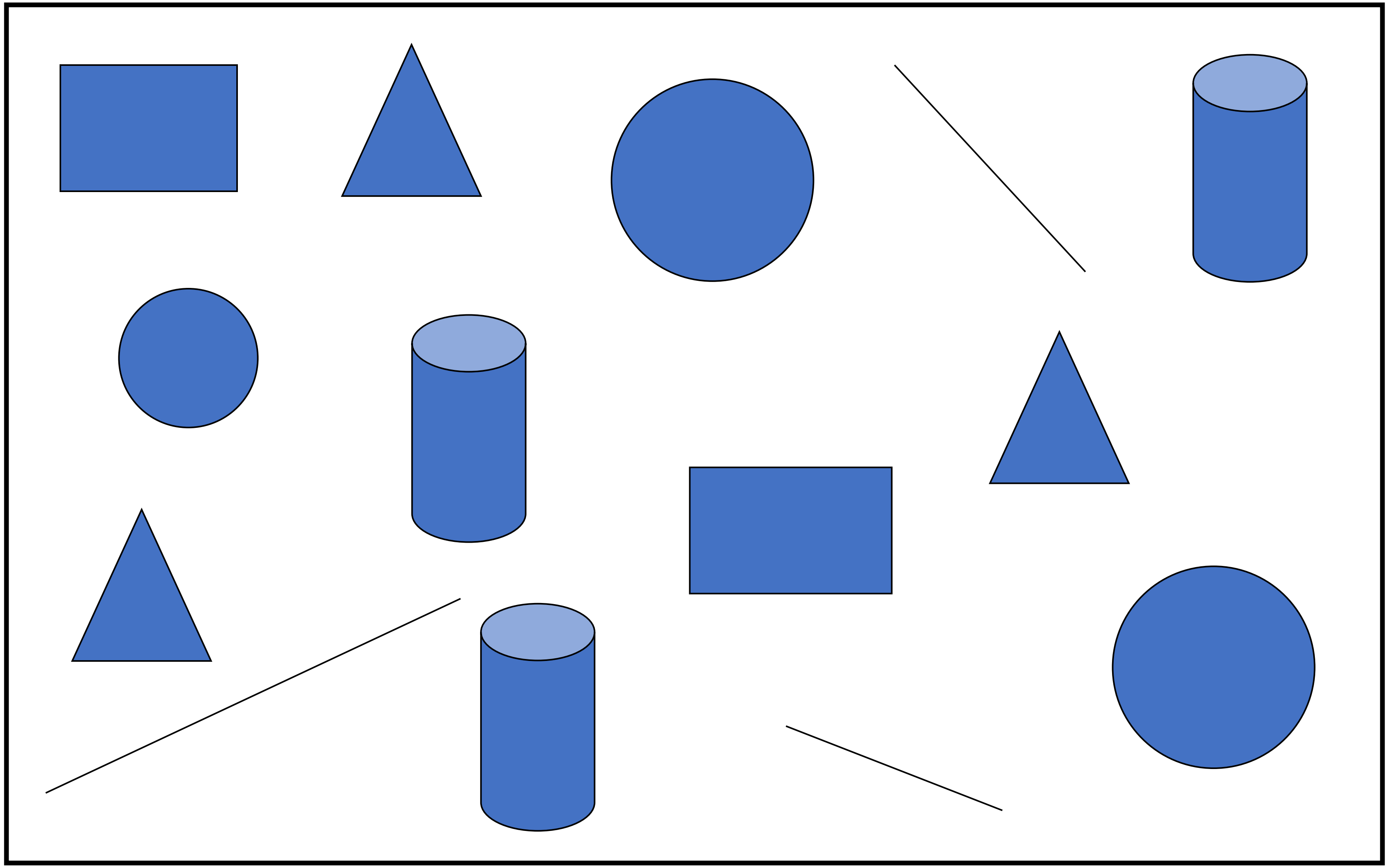
- The **Greek word** ***polymorphism*** means ***one name, many forms***.

- **Two types of Polymorphism in C++:**
  1. **Static polymorphism**: It can be **achieved** by **using overloading**. It is **defined at compilation time (i.e., static binding)**.

  2. **Dynamic polymorphism:** It can be **implemented** by **using inheritance** and **implemented at runtime (i.e., Dynamic Binding)** .

Graphics Drawing Software, *name these items*?

shapes



Graphics Drawing Software Classes

- **Line**
  - **Properties**:- **X-Y Coordinates, Length, Color**
  - **Actions**:- Draw Function, Change Color Function, Get Area Function.
- **Circle**
  - **Properties:- X-Y Coordinates, Radius, Color**
  - **Actions:-** Draw Function, Change Color Function, Get Area Function.
- **Rectangle**
  - **Properties**:- **X-Y Coordinates, Width, Height, Color**
  - **Actions**:- Draw Function, Change Color Function, Get Area Function.
- **Cylinder**
  - **Properties**:- **X-Y Coordinates, Radius, Height, Color**
  - **Actions**:- Draw Function, Change Color Function, Get Area Function.
- **Triangle**
  - **Properties**:- **X-Y Coordinates, Length, Width, Color**
  - **Actions**:- Draw Function, Change Color Function, Get Area Function.

```
class Line
{
  protected:
                int x,y;
  public:
                Line(int ,int );
                void draw();
                int GetArea (void);
};



Line::Line(int a,int b) {
        x=a;
        y=b;
}



void Line::draw( ) {
        cout << "\n Line Drawing code";
}


int Line::GetArea ( ) {
        cout << "\nLine Area "; return 0;
}
```

```
class Circle: public Line {
  protected:
        int radius;
  public:
        Circle(int ,int, int );
        void draw( );
        int GetArea ( );
};



Circle::Circle(int a, int b, int c) : Line (a, b) {
        radius = c;
}



void Circle::draw( ) {
        cout << “Circle drawing code”;
}


int Circle::GetArea ( ) {
        cout << “Circle area code”; return 0;
}
```

```cpp
class Rectangle: public Line {
  protected:
        int Width, Height;
  public:
        Rectangle(int, int , int , int );
        void draw(void);
        int GetArea (void);
};


Rectangle::Rectangle(int a, int b, int c, int d) : Line (a, b ) {
        Width = c;      Height = d;
}


void Rectangle::draw() {
         cout << "Rectangle drawing code";
}


int Rectangle::GetArea () {
        cout << "Rectangle area code"; return 0;
}
```

```cpp
class Triangle: public Line {
  protected:
        int a_axis,b_axis,c_axis;
  public:
        Triangle(int, int , int);
        void draw(void);
        int GetArea (void);
};


Triangle::Triangle(int a, int b, int c) : Line (a, b ) {
        a_axis= a;      b_axis= b; c_axis=c;
}


void Triangle ::draw() {
         cout << “Triangle drawing code”;
}


int Triangle ::GetArea () {
        cout << “Triangle area code”; return 0;
}
```

Shapes.cpp Demo

```
int main ( )
{
   Triangle t1 (3, 4, 5, 19 );
   Circle c1 (3, 4, 5 );
   Rectangle r1 ( 3, 4, 10 , 20 );

   t1.draw ();
   cout << “The area is “ << t1.GetArea ( );

   c1.draw ();
   cout << “The area is “ << c1.GetArea ();

   r1.draw ();
   cout << “The area is “ << r1.GetArea ();

   return 0;
}
```

Polymorphism Scenario in C++

- There is an **inheritance hierarchy**
- The **first class** that **defines** a **virtual function** is the **base class** of the **hierarchy** that **uses dynamic binding** for that **function name** and **signature**.
- **Each** of the **derived classes** in the **hierarchy** **must** have a **virtual function** with **same name and signature.**
- There is a **pointer of base class type** that is **used** to **invoke** **virtual functions** of **derived class**.

Pointers to Derived Classes

- **C++ allows base class pointers** to **point** to **both** **base class object** and **also all derived class objects**.

- **Let's assume:**

```
class Base { … };
class Derived : public Base { … };
```

- **Then, we can write:**

```
Base *p1;  Derived d_obj; p1 = &d_obj;
Base *p2 = new Derived;
```

Pointers to Derived Classes (contd.)

- **While it is allowed for a base class pointer to point to a derived object, the reverse is not true.**

```
base b1;
derived *pd = &b1; // compiler error
```

Pointers to Derived Classes (contd.)

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

- **What is a *Handle:***
  - The **item by which** the **members** of an **object** are **accessed:**
    - An **object name** (i.e., variable, etc.)
    - A **reference to an object**
    - A **pointer to an object**

Pointers to Derived Classes (contd.)

- Using a **base class pointer** (**pointing to a derived class object**) can **access only those members** of the **derived object that were inherited from the base**.

- **This is because the base pointer has knowledge only of the base class.**

- It **knows nothing** about the **members added** by the **derived class**.

DEMO: BasePtr.cpp

Summary – Based and Derived Class Pointers

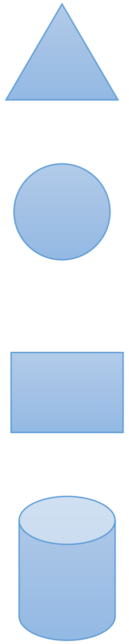
- **Base-class pointer pointing to base-class object**
  - **Straightforward**
- **Derived-class pointer pointing to derived-class object**
  - **Straightforward**
- **Base-class pointer pointing to derived-class object**
  - **Safe**
  - Can **access non-virtual methods** of **only base-class**
  -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

Pointers to Derived Classes

- With the **help of pointers** to **derived classes**, **we can create an array of base class objects**, and that **array can hold objects** of **<u>different derived classes</u>**

```
Line *p[4];
p[0] = new Triangle (3, 4, 5, 19 );
p[1] = new Circle (3, 4, 5 );
p[2] = new Rectangle ( 3, 4, 10 , 20 );
p[3] = new Cylinder ( 3, 4, 5, 10 );

for ( int loop = 0; loop < 4; loop ++ )
{    p[loop]->draw ();
     cout << “The area is “ << p[loop]->GetArea ( );
}
```



Virtual Functions based Shapes

DEMO: vShapes.cpp

```
class Shape{
public:
   virtual void sayHi() { cout <<``Just hi! \n'';}
};
class Triangle : public Shape{
public:
   virtual void sayHi() { cout <<``Hi from a triangle! \n'';}
};
class Rectangle : public Shape{
public:
   virtual void sayHi() { cout <<``Hi from a rectangle! \n; }
};

int main(){
   Shape *p;
   int which;
   cout << ``1 -- shape, 2 -- triangle, 3 -- rectangle\n '';
   cin >> which;
   switch ( which ) {
   case 1:  p = new Shape; break;
   case 2:  p = new Triangle; break;
   case 3:  p = new Rectangle; break;
   }
   p -> sayHi();    // dynamic binding of sayHi()
   delete p;
}
```

Virtual function

- Declaring a **function** virtual will **make** the **compiler** check the type of each object to search more **<u>specific version</u>** of the **virtual function**

- **To** declare **a** function virtual, **we use the Keyword** virtual:

```
class Shape
{
  public:
      virtual void sayHi ()
      {
              cout <<“Just hi! \n”;
      }
};
```

Virtual Functions

- If the member function definition is <u>outside the class</u>, the keyword virtual <u>must not be specified again</u>.

```
class Shape{
public:
  virtual void sayHi ();
};
virtual void Shape::sayHi (){  // error
  cout << ``Just hi! \n'';
}
```

- **Virtual functions can not be stand-alone or static functions.**

- **A virtual function can be inherited from a <u>base class</u> by a <u>derived class</u>, like other class member functions.**

Virtual Functions

- The ***virtualness*** of an **operation** is **always inherited**
- if a **function** is **virtual** in the **base class**, it **must be virtual** in the **derived class**,
- **Even if** the **keyword** "**virtual**" **not specified** (But always use the keyword in children classes for clarity.)
- If **no overridden function** is **provided**, the **virtual function** of **base class** is **used**

Introduction to Virtual Functions

- **Terminology in C++:**
  - **redefine** a **method** that uses **static binding**
  - **override** a **method** that uses **dynamic binding** (i.e., ***virtual functions***)

Virtual Functions

- To **override** a **base class virtual function**, the **virtual function instance** in **derived class must match** the **base class virtual function exactly**.

- **The overriding functions are virtual automatically. The use of keyword virtual is optional in derived classes.**

Virtual Functions

How to declare a member function virtual:

```
class Animal{
 public:
      virtual void id(){cout << "animal";}
};

class Cat : public Animal{
 public:
      virtual void id(){cout << "cat";}
};

class Dog : public Animal{
 public:
      virtual void id(){cout << "dog";}
};
```

Polymorphism Example
(using Base Class's Pointers and References)

```
class Shape{
public:
   virtual void sayHi() { cout <<``Just hi! \n'';}
};
class Triangle : public Shape{
public:
   // overrides Shape::sayHi(), automatically virtual
   void sayHi() { cout <<``Hi from a triangle! \n'';}
};

void print(Shape obj, Shape *ptr, Shape &ref){
   ptr -> sayHi();   // bound at run time
   ref.sayHi();      // bound at run time
   obj.sayHi();      // bound at compile time
}

int main(){
  Triangle mytri;
  print( mytri, &mytri, mytri );
}
```

DEMO: PolyExample2.cpp

Virtual Destructors

- **Constructors cannot be virtual**, but **destructors can be virtual** when a **constructor** of a class is executed there is no **virtual** table in the memory, means
  no **virtual** pointer defined yet.

- **Ensures:** the **derived class destructor** is **called** when a **base class pointer** is **used,** while ***deleting a dynamically created derived class object***.

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

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

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