Polymorphism (Lecture -1)

(CS 217)

Dr. Muhammad Aleem,

Department of Computer Science,

National University of Computer & Emerging Sciences,
Islamabad Campus

Lecture 1 - Contents

- Binding Process
- Static and Dynamic Binding
- Polymorphism Introduction
- Polymorphism Scenario in C++
- Pointers to Derived Classes

Binding Process

- **Binding** is the **process** to **associate names (variable or function)** with **memory addresses**.

- **Binding** is **performed** for **each variable** and **function** in the program.

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

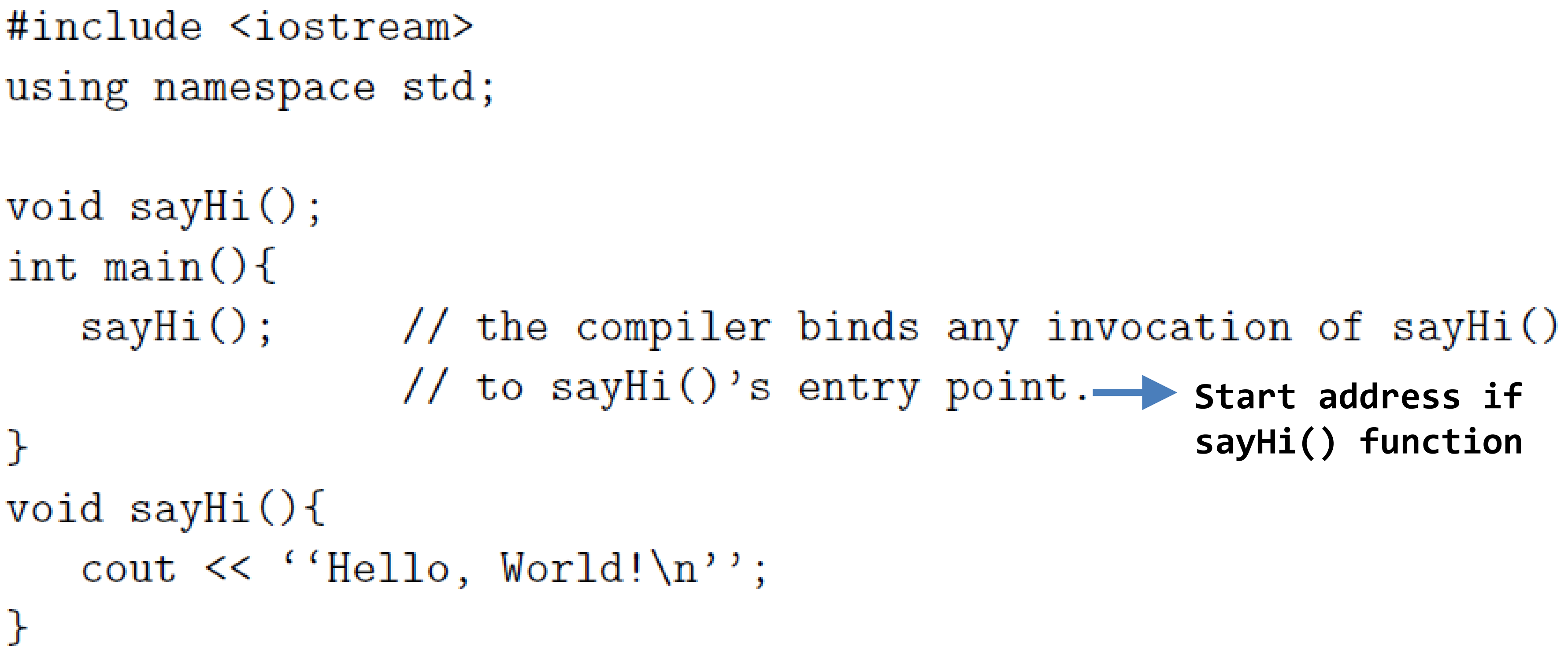
Binding in C and C++

- C provides only compile time binding

- C++ provides both compile and run-time binding

Compile-time Binding (Static Binding)

- **Compile-time binding: associating** 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’’;
}
```

Start address if sayHi() function

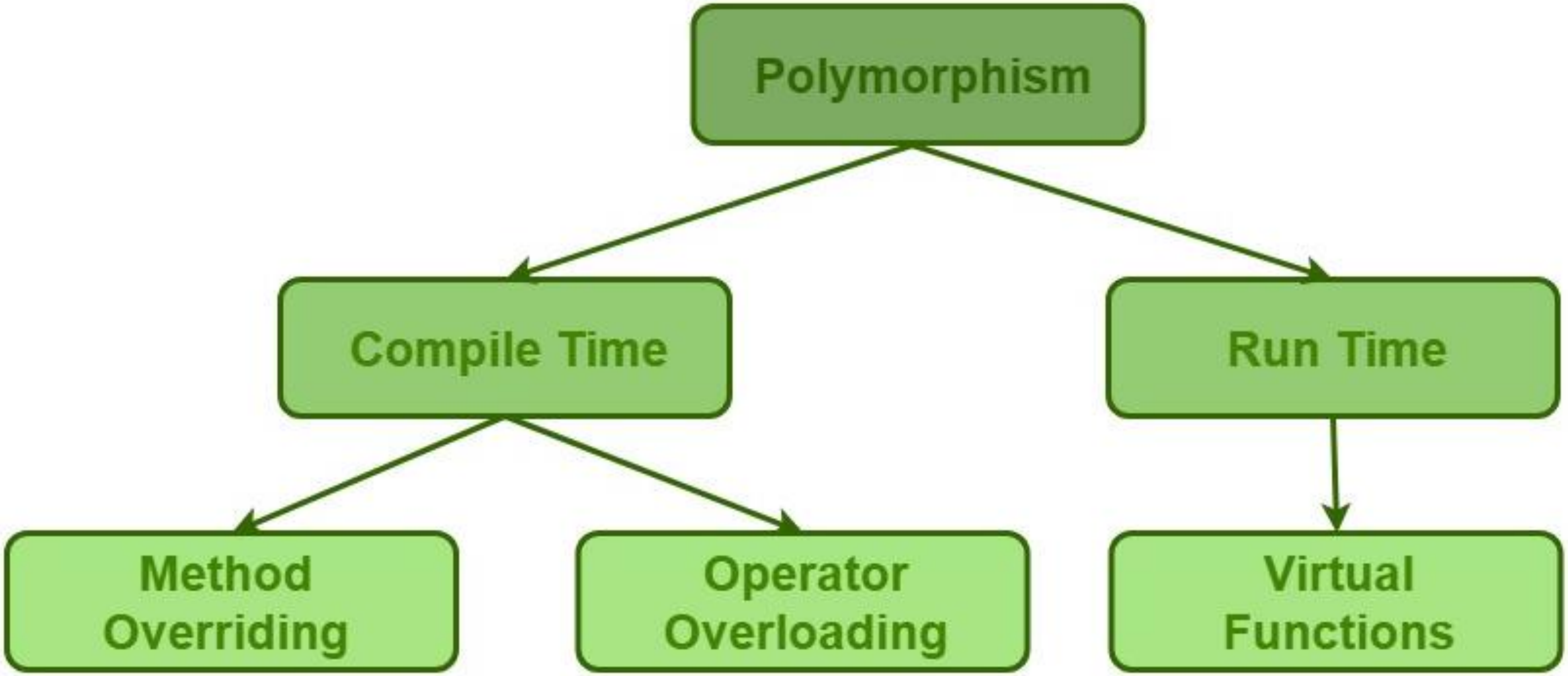
Run-time Binding (Dynamic Binding)

- **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 in C++



Polymorphism

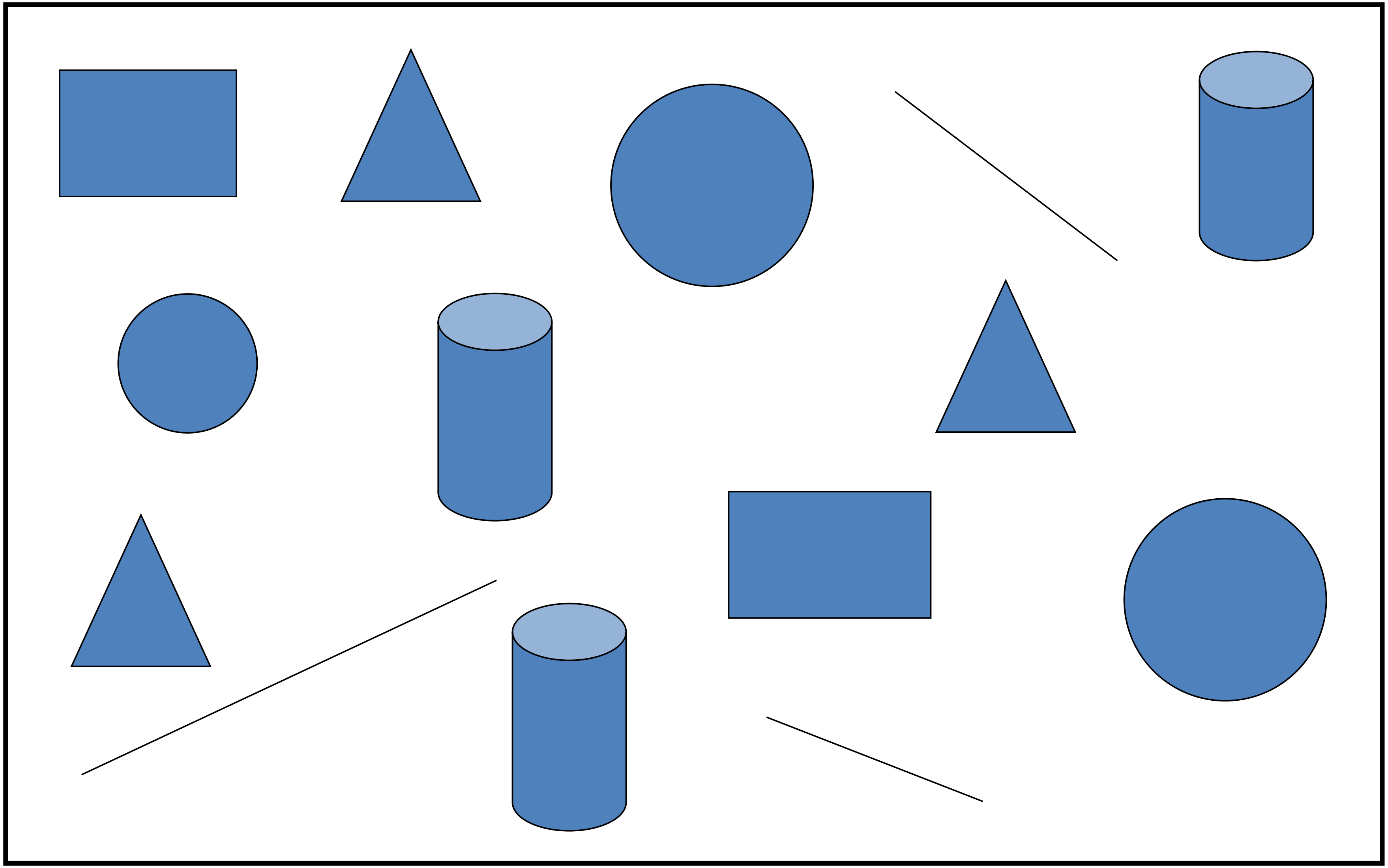
- The **Greek word** ***polymorphism*** means **one name, many forms**.

- **C++ propovide two types of Polymorphism:**

  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);
};
```

DEMO: Shapes.cpp

```
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;
}
```

```
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;
}
```

```
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;
}
```

```
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++

1. There is an **inheritance hierarchy**

2. The **first class** that **defines** a **virtual function** is the **base class** of the **hierarchy** (**dynamic binding** for that **function name**).

3. **Each** of the **derived classes** in the **hierarchy must have** a **virtual function** with **same name and signature (to override).**

4. 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 the** base class object and **also all** derived class objects.

- **Let’s assume:**

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

- **Then, we can write:**

```
Base *bptr;
Derived1 obj1; Derived2 obj2;
bptr = &obj1; // function calls
bptr = &obj2; // function calls
```

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 bObj;
derived *pd = &bObj; // compiler error
```

Pointers to Derived Classes (contd.)

- **Access to members** of a **class object** is **determined** by the **type** 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**) we 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

End of Lecture 1