CS-1201 Object Oriented Programming

Polymorphism

Arbish Akram

Department of Computer Science
Government College University

Function Overriding

- Function overriding allows a child class to provide a new implementation of a function that is already defined in its parent class.
- This feature enables the child class to modify or extend the behavior of the inherited function.
- A child class inherits data members and member functions from the parent class.
- To override a function in the child class, the function signature in the child class must match the one in the parent class.
- Overriding is like creating a new version of an old function in the child class.

Function Overriding I

Invoking derived-class function

```
1  class Base {
2     public:
3      void print() {
4          cout << "Base Function" << endl;
5      }
6  };
7  class Derived : public Base {
8     public:
9      void print() {
10         cout << "Derived Function" << endl;
11     }
12 };
13 int main() {
14     Derived derived1;
15     derived1.print();
16     return 0;
17 }
```

Derived Function

Function Overriding II

```
1  class Base {
2     public:
3      void print() {
4          cout << "Base Function" << endl;
5      }
6  };
7  class Derived : public Base {
8     public:
9      void print() {
10         cout << "Derived Function" << endl;
11     }
12 };
13 int main() {
14     Derived derived1, derived2;
15     derived1.print();
16     // access print() function of the Base class
17     derived2.Base::print();
18     return 0;
19 }
```

Derived Function
Base Function

Function Overriding III

```
1  class Base {
2     public:
3      void print() {
4          cout << "Base Function" << endl;
5      }
6  };
7  class Derived : public Base {
8     public:
9      void print() {
10         cout << "Derived Function" << endl;
11
12         // call overridden function
13         Base::print();
14     }
15 };
16 int main() {
17     Derived derived1;
18     derived1.print();
19     return 0;
20 }
```

Derived Function
Base Function

Function Overriding IV

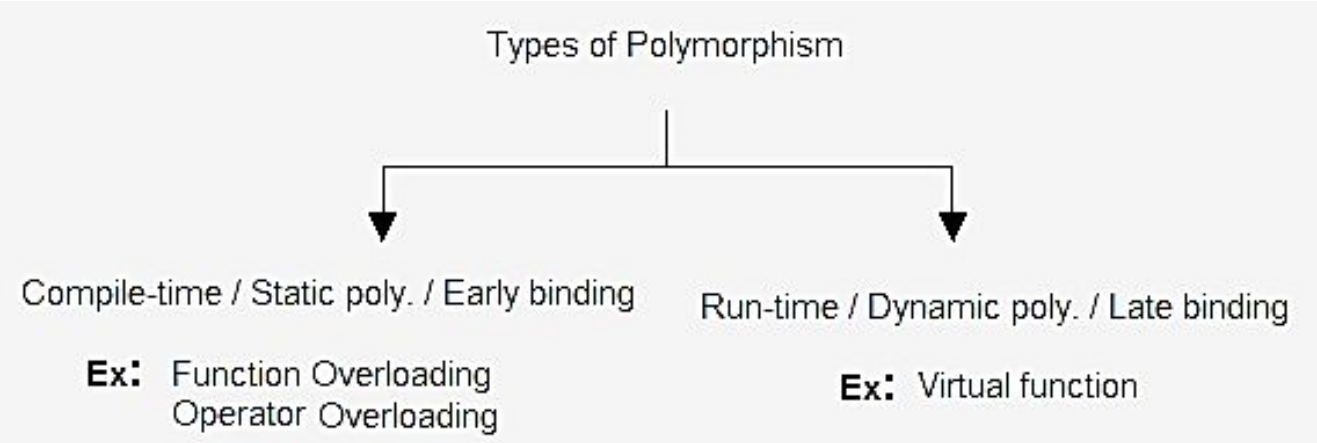
```
1  class Base {
2     public:
3      void print() {
4          cout << "Base Function" << endl;
5  }};
6  class Derived : public Base {
7     public:
8      void print() {
9          cout << "Derived Function" << endl;
10 }};
11 int main() {
12     Derived derived1;
13     // pointer of Base type that points to derived1
14     Base* ptr = &derived1;
15     // call function of Base class using ptr
16     ptr->print();
17     return 0;
18 }
```

Output: Base Function

If `ptr` points to a `Derived` object but is of type `Base*`, it calls the member function of `Base`.

Polymorphism

- A single interface can have multiple implementations.
- Objects of different classes, related by inheritance, can respond differently to the same member function call.
- More than one function with the same name can exist, each with different functionality.
- Polymorphism can be categorized into two main types:
  - **Compile-time Polymorphism** (Static Polymorphism)
  - **Runtime Polymorphism** (Dynamic Polymorphism)



Static vs Dynamic Binding

- Binding refers to the process of associating identifiers (such as variable and function names) with addresses.
- Binding is performed for both variables and functions.
- It is the link between a function call and its corresponding function definition.
- **Early Binding** (Compile-time Polymorphism): As the name suggests, the compiler (or linker) directly associates an address with the function call. It replaces the call with a machine-language instruction.
- **Late Binding** (Run-time Polymorphism): In this case, the compiler adds code to identify the object type at runtime, then matches the call with the correct function definition.

Static Binding

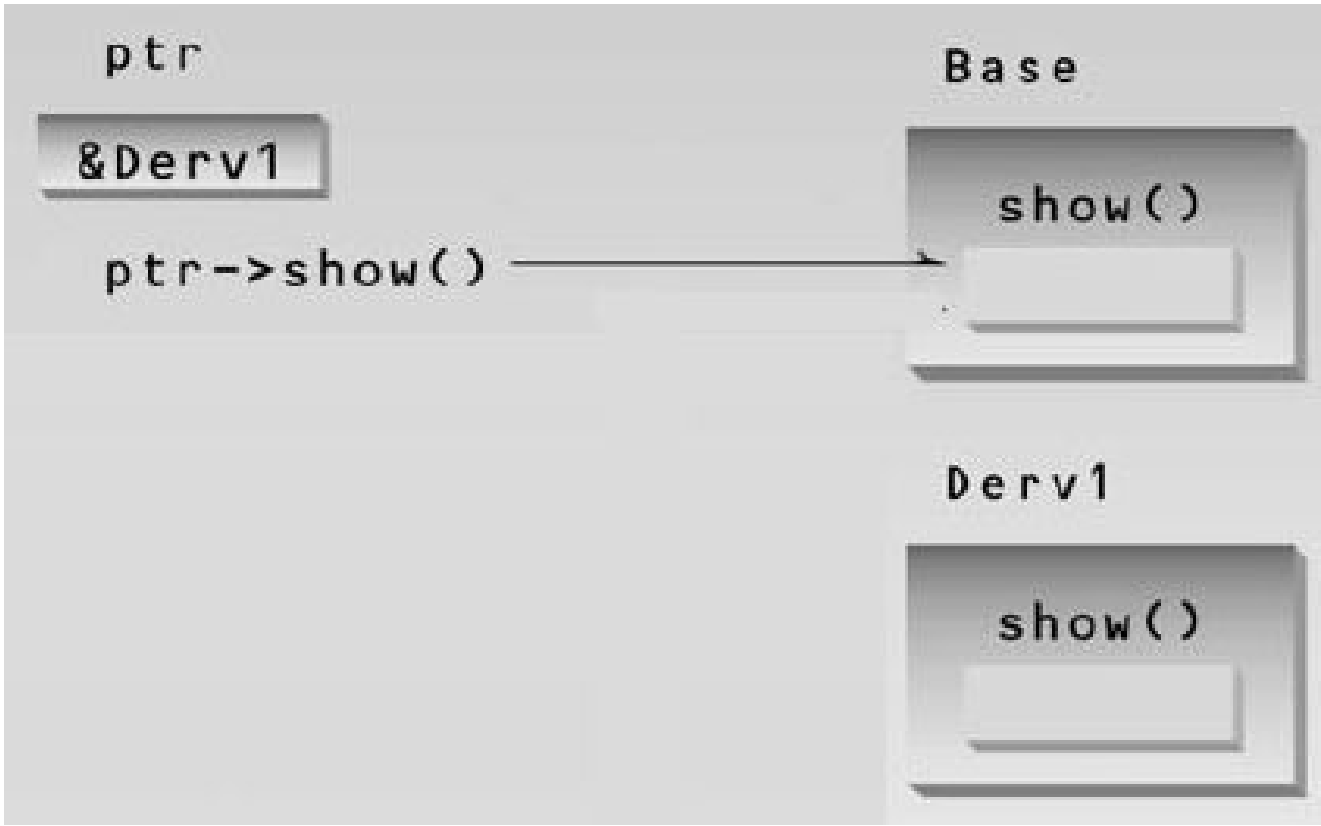
```
1  class ComputeSum
2  {
3    public:
4      int sum(int x, int y)
5      {
6          return x + y;
7      }
8      int sum(int x, int y, int z)
9      {
10         return x + y + z;
11     }
12 };
13 int main()
14 {
15     ComputeSum obj;
16     cout << "Sum is " << obj.sum(10, 20) << '\n';
17     cout << "Sum is " << obj.sum(10, 20, 30) << '\n';
18     return 0;
19 }
```

Even with the same function name, the call to sum() binds to the correct function based on the parameters, determined statically at compile time.

Invoking Base-Class Functions from Derived-Class Objects

```
1  class B
2  {
3      public:
4      void f() {
5          cout << " In Base \n";
6          }
7  };
8  class D : public B
9  {
10     public:
11     void f() {
12         cout << "In Derived \n";
13         }
14 };
15 int main()
16 {
17     B *ptr;
18     D dv1;
19     ptr = &dv1; // Assign address of dv1 to the pointer
20     ptr->f();   // Invoke function via base class pointer
21     return 0;
22 }
```

Invoking Base-Class Functions from Derived-Class Objects



Invoking Base-Class Functions from Derived-Class Objects

- Even though the base-class pointer points to a derived-class object, the base class's member function is invoked (rather than the derived class's member function).
- The output of the function call demonstrates that the invoked functionality depends on the type of the pointer (or reference) used to invoke the function, not the type of the object for which the member function is called.
- To invoke the derived class's functionality, rather than the base class's functionality, you can use virtual functions (dynamic binding), which will be discussed later.

Invoking Derived-Class Functions from Base-Class object

```
1  class B
2  {
3      public:
4      void f() {
5          cout << " In Base \n";
6          }
7  };
8  class D : public B
9  {
10     public:
11     void f() {
12         cout << "In Derived \n";
13         }
14 };
15 int main()
16 {
17     B bv1;
18     D *ptr;
19     ptr = &bv1; // Assign address of bv1 to the pointer
20     ptr->f();   // Invoke function via derived class pointer
21     return 0;
22 }
```

Derived-Class Member-Function Calls via Base-Class Pointers

- Using base class pointer or object, the compiler allows us to invoke only base-class member functions.
- A compilation error will occur if an attempt is made to access a derived-class-only member function.

Virtual Functions

- **Virtual functions**: With virtual functions, the type of the object, not the type of the handle (pointer or reference) used to invoke the member function, determines which version of a virtual function to invoke.
- A virtual function is defined as a member function within a base class that you expect to override (redefine) in derived classes.
- To create a virtual function, you must precede the function's declaration in the base class with the `virtual` keyword.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call.
- They are mainly used to achieve **Runtime Polymorphism**.

Rules for Virtual Functions

- Virtual functions cannot be **static** and cannot be a **friend function** of another class.
- Virtual functions should be accessed using a **pointer** or **reference** of the base class type to achieve runtime polymorphism.
- The prototype of virtual functions should be the same in both the **base class** and the **derived class**.
- Virtual functions are always defined in the base class and overridden in derived classes. It is **not mandatory** for the derived class to override the virtual function; if not, the base class version will be used.
- A class may have a **virtual destructor**, but it cannot have a **virtual constructor**.

Dynamic Binding

```
1  class base {
2      public:
3      virtual void print() {
4          cout << "print base class" << endl;
5      }
6      void show() {
7          cout << "show base class" << endl;
8      }};
9  class derived : public base {
10 public:
11     void print() {
12         cout << "print derived class" << endl;
13     }
14     void show() {
15         cout << "show derived class" << endl;
16     }};
17 int main() {
18     base* bptr;
19     derived d;
20     bptr = &d;
21     bptr->print(); // virtual function, binded at runtime
22     bptr->show(); // Non-virtual function, binded at compile time
23 }
```

Notes

- Runtime polymorphism is achieved only through a pointer (or reference) of base class type.
- Also, a base class pointer can point to the objects of base class as well as to the objects of derived class.

Pure Virtual Functions

- A **pure virtual function** (or **abstract function**) is a virtual function with no implementation.
- It is only declared, not defined.
- A pure virtual function is specified by placing "= 0" in its declaration as follows:

```
class Box
{
    public:
        // pure virtual function
        virtual double getVolume() = 0;
    private:
        double length;
        double breadth;
        double height;
};
```

Pure Virtual Functions

```
1  class Shape {
2      protected:
3          float dimension;
4      public:
5          void getDimension() {
6              cin >> dimension;
7          }
8      // pure virtual Function
9      virtual float calculateArea() = 0;
10 };
11 class Square : public Shape {
12    public:
13         float calculateArea() {
14             return dimension * dimension;
15     }
16 };
```

Pure Virtual Functions

```
1  class Circle : public Shape {
2     public:
3      float calculateArea() {
4          return 3.14 * dimension * dimension;
5      }
6  };
7  int main() {
8      Shape *ptr1, *ptr2;
9      Square square;
10     Circle circle;
11     ptr1 = &square;
12     ptr2 = &circle;
13
14     cout << "Enter the length of the square: ";
15     ptr1->getDimension();
16     cout << "Area of square: " << ptr1->calculateArea() << endl;
17
18     cout << "\nEnter radius of the circle: ";
19     ptr2->get dimension();
20     cout << "Area of circle: " << ptr2->calculateArea() << endl;
21     return 0;
22 }
```

What is an Abstract Class?

- An **abstract class** is a class that cannot be instantiated directly.
- It is used as a base class for other derived classes.
- An abstract class contains at least one **pure virtual function**.
- A pure virtual function is a function declared in the abstract class without an implementation. It must be implemented by derived classes.
- Abstract classes are useful for defining a common interface for a group of related classes.
- Example: A `Shape` class can be an abstract class with a pure virtual function `draw()`, which must be implemented by every derived class (like `Circle`, `Square`).

Cannot Instantiate Objects of Abstract Classes

- Attempting to instantiate an object of an abstract class results in a compile-time error.

```
class Shape {
    public:
        // pure virtual function
        virtual void draw() = 0;
};
int main() {
   // Error: cannot instantiate an abstract clas
    Shape s;
}
```

Abstract Class vs Interface in C++

Abstract Class:

- Can have both abstract (pure virtual) and non-abstract functions (with implementations).
- Can have member variables (data members).
- Can have constructors and destructors.
- Derived classes must implement pure virtual functions to be instantiated.

Interface:

- An interface is a class that only contains pure virtual functions (no implementation).
- Cannot have member variables (except for static constants).
- Cannot have constructors or destructors.
- Interfaces are typically implemented by abstract classes that only contain pure virtual functions.