Programming Language

Polymorphism *(Day 5 Part 1)*

By: Bhaskar Ghosh
bjghosh@gmail.com

Updated: 6 August 2024

C++ Programming Language

Object Oriented Programming (Day 5 Part 1)

- Polymorphism in C++
- Compile-time Polymorphism
  - Function Overloading
  - Operator Overloading
- Run-time Polymorphism – Virtual Functions
- Multiple Inheritance – Diamond Problem
- Abstract Classes and Pure Virtual Functions in C++
- Problem Solving using C++ [Level 4]

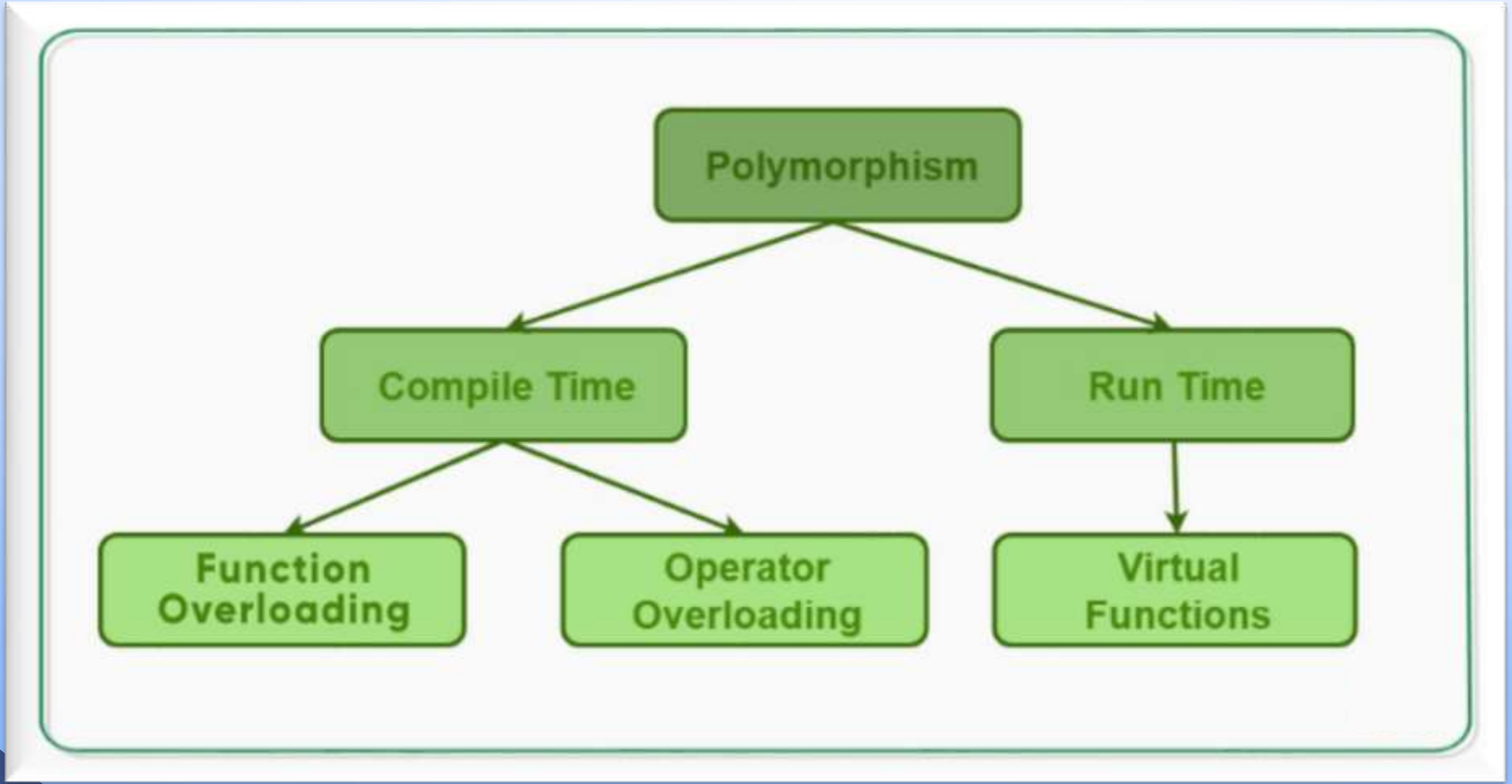
C++ Programming Language

Polymorphism

- “poly” means “many, “morph” means “forms”
- So, Polymorphism means many forms.
- It is an object-oriented programming concept that refers to the ability of a variable, function, or object to take on multiple forms
  - i.e., the behavior of the same object or function is different in different contexts.
- Polymorphism can occur within the class and when multiple classes are related by inheritance.

C++ Programming Language

Polymorphism – Types



C++ Programming Language

Compile-time Polymorphism – Function Overloading

- When we have two functions with the same name but different parameters (numbers or types), then different functions are called depending on the number and data types of parameters.
- This is known as function overloading.
- Cases of Function Overloading
  - The names of the functions and return types are the same but differ in the type of arguments.
  - The name of the functions and return types are the same, but differ in the number of arguments.

C++ Programming Language

Compile-time Polymorphism – Operator Overloading

- If we want an existing operator to work for objects of our user defined class, then we need to overload the operator for that class.
- This is known as operator overloading.
- To use operator overloading, at least one operand must be a user-defined data type.
- ".", "::", typeid, size, ".*", and C++'s single ternary operator, "?:", are the operators that cannot be overloaded.

C++ Programming Language

Run-time Polymorphism – Virtual Functions

- Runtime polymorphism occurs when functions are resolved at runtime rather than compile time.
  - i.e., when a call to an overridden method is resolved dynamically at runtime rather than compile time.
- Also known as ***late binding*** or ***dynamic binding***.
- Achieved using a combination of **function overriding and virtual functions**
- A **virtual function** in C++ is a base class member function declared using the keyword virtual.
- The class with ***atleast one virtual*** function is a ***polymorphic class***.
- Calling a virtual function makes the compiler execute a version of the function suitable for the object in question, when the object is accessed by a pointer or a reference to the base class!

C++ Programming Language

Run-time Polymorphism – Virtual Functions

- If we create a pointer of Base class type to point to an object of Derived class and call a member function, it calls the member function (with the same name) of the Base class.
- To avoid this, we declare the member function of the Base class as virtual by using the virtual keyword.

```
class Base{
   public:
      virtual void print(){...}
};

Class Derived : public Base {
   public:
      void print(){...}
}
```

C++ Programming Language

Virtual Destructors

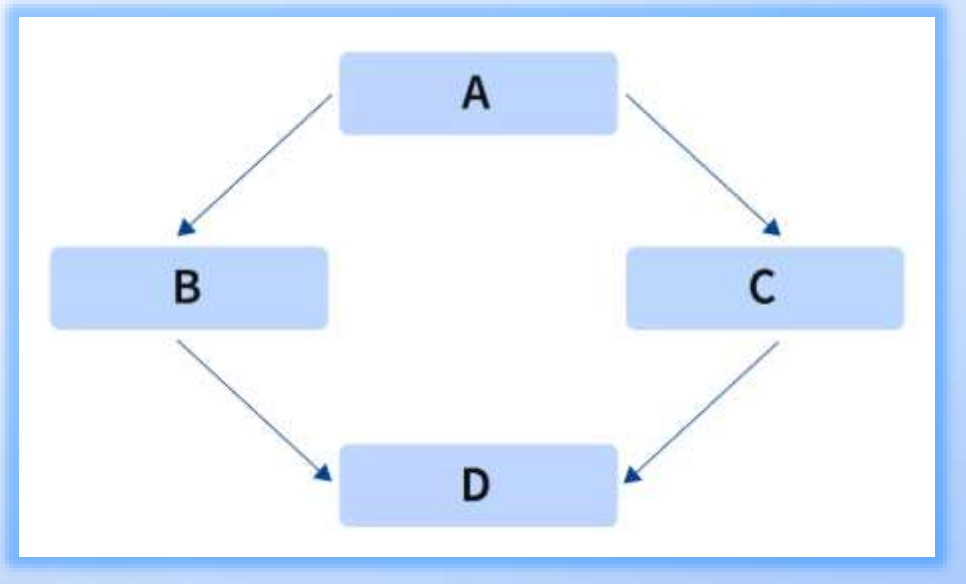
- Base class destructors should always be virtual.
- Suppose we use delete with a base class pointer to a derived class object to destroy the derived-class object.
- If the base-class destructor is not virtual then delete, like a normal member function, calls the destructor for the base class, not the destructor for the derived class.
- This will cause only the base part of the object to be destroyed.

```
class Base{
   public:
      virtual ~Base(){...}
};

Class Derived : public Base {
   public:
      ~Derived{...}
}

int main()
{
   Base* pBase = new Derv;
   delete pBase;
   return 0;
}
```

C++ Programming Language

Multiple Inheritance – Diamond Problem

- With multiple inheritance, a problem can occur during function overriding.
- Suppose two base classes have the same function which is not overridden in the derived class.
- If we try to call the function using the object of the derived class, the compiler shows error.
- Because the compiler doesn't know which function to call.



C++ Programming Language

Virtual Inheritance – Solution to Diamond Problem

- The solution to the diamond problem is Virtual inheritance.
- It is a technique that ensures that only one copy of the base classes or base class member variables is inherited by the second-level derivatives that are grandchild.
- If we use virtual inheritance, then only one copy of the member of the base class is passed into the 2nd level derived class.
- So, there is no ambiguity.
- We use the following syntax to inherit from a virtual base class.

```
class DerivedClass : virtual public BaseClass { ... };
```

C++ Programming Language

Abstract Classes and Pure Virtual Functions in C++

- We can skip the definition for a virtual member function in a Base class by declaring the function as ***pure virtual function***.
- For this, we have to add the expression = 0 after the function declaration statement inside the Base class.

```
virtual void print() = 0;
```

- Since, the definition of such a Base class is incomplete, we can not create objects of this Base class.
- Hence, we call such a Base class as ***Abstract class***.
- We must define the pure virtual function inside a Derived class.
- Only then, can we create objects by using the Derived class.

Programming Language

Generics – Templates in C++

(Day 5 Part 2)

By: Bhaskar Ghosh
bjghosh@gmail.com

Updated: 6 August 2024

C++ Programming Language

Object Oriented Programming (Day 5 Part 2)

- Generic Programming – Templates in C++
- Function Templates
- Class Templates
- Class Templates and Static Variables
- Class Template and Inheritance
- Template Argument Deduction
- Problem Solving using C++ [Level 5]

C++ Programming Language

Generic Programming – Templates in C++

- Generic Programming enables the programmer to write a general algorithm which will work with all data types.
- Generics can be implemented in C++ using Templates.
- Templates make it possible to use one function or class to handle many different data types.
- In templates, we specify a placeholder instead of the actual data type, and that placeholder gets replaced with the data type used during the compilation.

C++ Programming Language

Function Templates

- Function templates are similar to normal functions.
- Normal functions work with only one data type, but a function template code can work on multiple data types.
- Functional templates are more powerful than overloading a normal function as we need to write only one program, which can work on all data types.

```
template <class T> T function-name(T args)
{
    // body of function
}
```

- Note: We can also use the keyword "typename " in place of "class "

C++ Programming Language

Function Templates

- Declaration and Definition of Function templates ***must be in the same file***.
- The templates are not normal functions. They only get compiled when we call a template function with actual data values.
- So, templates are compiled only when required.
- i.e., the compiler generates the exact functionality with the provided arguments and template.

C++ Programming Language

Class Templates

- Like function templates, we can also use templates with the class to make it compatible with more than one data type.

```
template <class T>
class className {
 // Class Definition.
 // We can use T as a type inside this class.
};
```

- When a class uses the concept of template in C++, then the class is known as a generic class.
- Some pre-defined examples of class templates in C++ are vector, LinkedList, Stack, Queue, Array, etc.

C++ Programming Language

Class Templates and Static Variables

- As we know, classes in C++ can contain two types of variables, static and non-static(instance).
- Each object of the class consists of non-static variables.
- But the static variable remains the same for each object means it is shared among all the created objects.
- So, the static variable in template classes remains shared among all objects of the same type.
- i.e., for different data types, static variables have different values.
- i.e., every type has a separate copy of the static variable.

C++ Programming Language

Class Templates and Inheritance – Scenarios

1. Base Class is not a Template class, but a Derived class is a Template class.
   - We can derive from the non-template class and add template members to the derived class.
2. Base Class is a Template class, but Derived class is not a Template class.
   - If we don't want our derived class to be generic, we can use the Base class by providing the template parameter type.
3. Base Class is a Template class, and the Derived class is also a Template class.
   - If we want our derived class to be generic, then it should be a template that can pass the template parameter to the base class.
4. Base Class is a Template Class, and derived class is a Template class with different Types.
   - Additional template types can be included in the template parameter of the derived class template.

C++ Programming Language

Template Argument Deduction (1)

- Template argument deduction automatically deduces the data type of the argument passed to the class or function templates.
- This allows us to instantiate the template without explicitly specifying the data type.

```
template <class T> T add(T num1, T num2)
{
    return num1 + num2;
}
```

```
add<int> (2,3);
```

or

```
add(2,3);
```

Note: Function template argument deduction has been since the C++98

C++ Programming Language

Template Argument Deduction (2)

- The class template argument deduction was added in C++17.
- It allows us to create the class template instances without explicitly definition the types just like function templates.

```
template <class T>
class Exam {
 // Class Definition.
 // We can use T as a type inside this class.
};
```

```
Exam <int> obj(60);
```

or

```
Exam obj(60);
```

Note: Class template argument deduction has been since the C++17

Problem Solving using C++

logic Building and Debugging
Design & Implementation

C++ Programming Language

Problem Solving - Templates

1. Write a template function that performs following action:
   - 1st case when two strings are given , print the smallest of the two strings.
   - 2nd case when two integers are given, print the smallest of the two integers.
   - 3rd case when two char are given , print the smallest of the two characters(lowercase).
2. Design a Banking System
   - Think about the various classes that will be needed, and how the classes will be related.
   - Draw a class diagram to depict these relations.