Object Oriented Programming using C++

CSE-III
CSE206-OOP

Dr. Priyanka Singh

UNIT III: POLYMORPHISM

- Concept of Polymorphism, Function overloading
- Examples and advantages of function overloading
- Pitfalls of function overloading
- Operator overloading
- Overloading unary operations
- Overloading binary operators
- Pitfalls of operators overloading

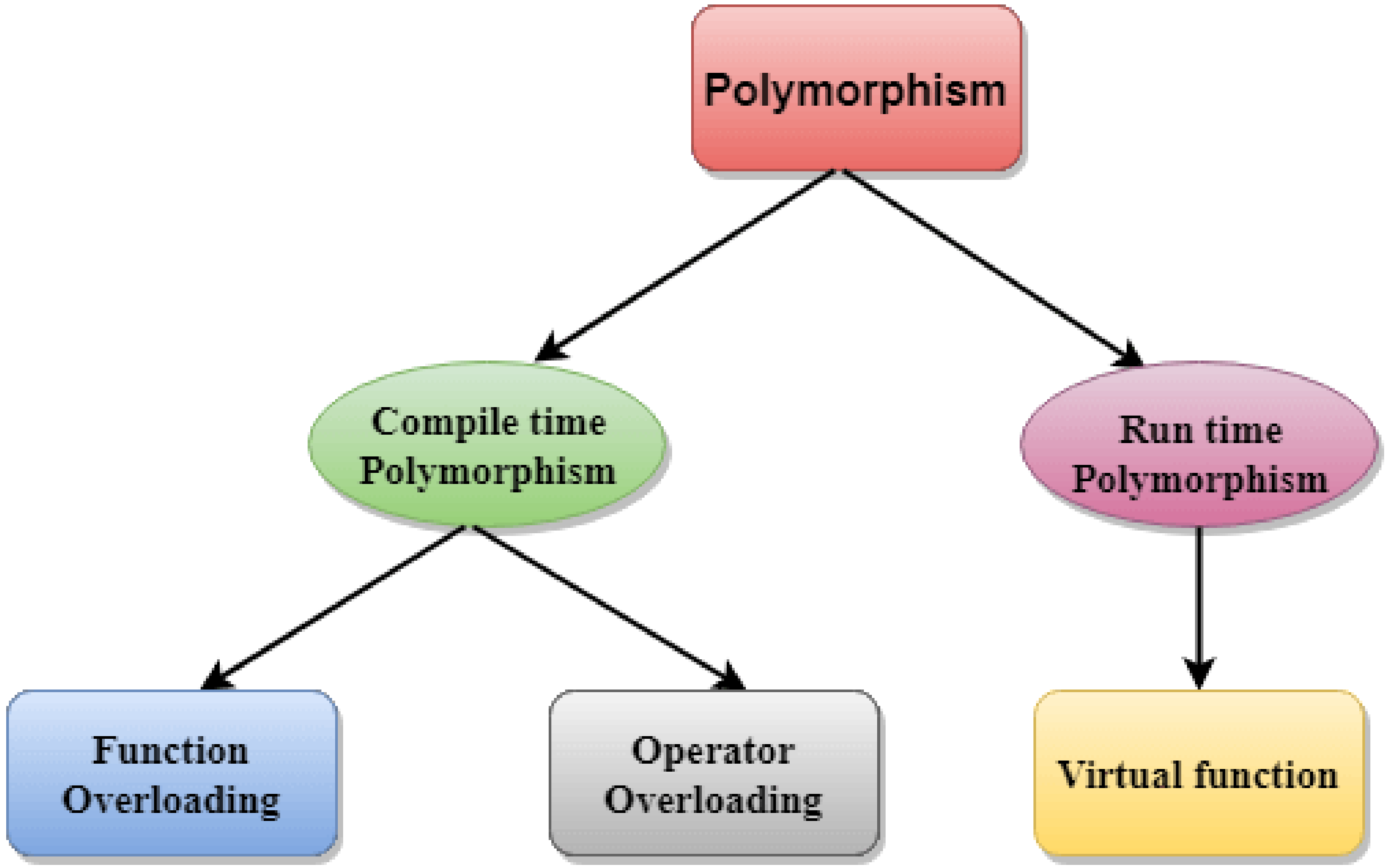
Overview of C++ Polymorphism

- The word **polymorphism** means having many forms.
- Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.
- C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Let's consider a real-life example of polymorphism.

- A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market.
- Here, a single person is behaving differently according to the situations.

Types of polymorphism in C++:



Compile time polymorphism:

- The overloaded functions are invoked by matching the type and number of arguments.
- This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time.
- It is achieved by function overloading and operator overloading which is also known as static binding or early binding.

Function Overloading:

- When there are multiple functions with same name but different parameters then these functions are said to be overloaded.
- Functions can be overloaded by change in number of arguments or/and change in type of arguments.
- The function is redefined by either using different types of arguments or a different number of arguments.
- It is only through these differences that a compiler can differentiate between functions.
- We can overload: methods, constructors, and indexed properties.

Advantages of function Overloading in C++

- The main advantage of function overloading is that it improves code readability and allows code reusability.
- The use of function overloading is to save memory space, consistency, and readability.
- It speeds up the execution of the program
- Code maintenance also becomes easy.
- Function overloading brings flexibility to code.
- The function can perform different operations and hence it eliminates the use of different function names for the same kind of operations.

```
#include <bits/stdc++.h>
using namespace std;

class Function1
{
        public:
        void func(int x)
        {
                cout << "value of x is " << x << endl;
        }
        void func(double x)
        {
                cout << "value of x is " << x << endl;
        }
        void func(int x, int y)
        {
                cout << "value of x and y is " << x << ", " << y << endl;
        }
};
```

```
int main() {
        Function1 obj1;
        obj1.func(7);
        obj1.func(9.132);
        obj1.func(85,64);
        return 0;
}
```

Program of function overloading when number of arguments vary.

```
#include <iostream>
using namespace std;
class Cal {
    public:
static int add(int a,int b)
   {
        return a + b;
    }
static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void) {
    Cal C;                                                   //    class object declaration.
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
   return 0;
}
```

```
OUTPUT
30
55
```

Class work

- WAP to find the volume of cuboid, cone and cylinder using function overloading.

- WAP to find the volume of cuboid, cone and cylinder using constructor overloading.

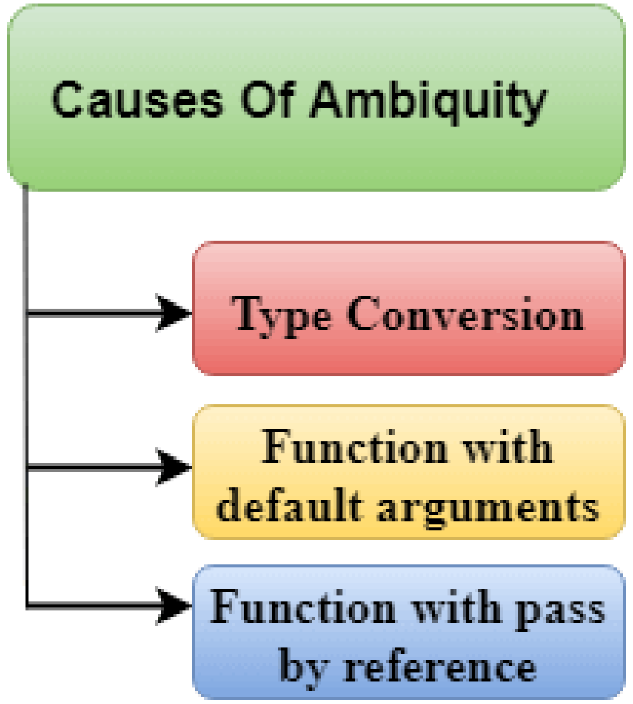
Disadvantages of function Overloading in C++

- Function declarations that differ only by its return type cannot be overloaded with function overloading process.
- Member function declarations with the same parameters or the same name types cannot be overloaded if any one of them is declared as a static member function.

```
class XYZ{
  static void func();
  void func(); // error
  };
```

Function Overloading and Ambiguity

- When the compiler is unable to decide which function it should invoke first among the overloaded functions, this situation is known as function overloading ambiguity.
- The compiler does not run the program if it shows ambiguity error.
- Causes of Function Overloading:
  - Type Conversion.
  - Function with default arguments.
  - Function with pass by reference.


Causes Of Ambiquity
Type Conversion
Function with default arguments
Function with pass by reference

Type Conversion:

```
#include<iostream>
using namespace std;
void fun(int);
void fun(float);
void fun(int i)
{     std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(float j)
{     std::cout << "Value of j is : " <<j<< std::endl;
}
int main()
{   fun(12);
    fun(1.2);
    return 0;  }
```

The example shows an error "**call of overloaded 'fun(double)' is ambiguous**". The fun(12) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

Function with Default Arguments

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int, int);
void fun(int i)
{     std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a, int b=9)
{     std::cout << "Value of a is : " <<a<< std::endl;
   std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{     fun(12);
   return 0;
}
```

The example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

Function with pass by reference

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{  int a=10;
fun(a); // error, which f()?
return 0;
}
void fun(int x)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
}
```

The example shows an error "**call of overloaded 'fun(int&)' is ambiguous**". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

Runtime polymorphism

- This type of polymorphism is achieved by Function Overriding. Late binding and dynamic polymorphism are other names for runtime polymorphism.
- The function call is resolved at runtime in runtime polymorphism.
- In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.
- Function Overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.
- 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.

```
#include <bits/stdc++.h>
using namespace std;
class base {public:
            virtual void print()
            {           cout << "print base class" <<endl;    }
            void show()
            {           cout << "show base class" <<endl;   }
};
class derived : public base {public:
            void print()
            {           cout << "print derived class" <<endl;}
            void show()
            {           cout << "show derived class" <<endl;}
};
int main()
{           base* bptr;
            derived d;
            bptr = &d;
            bptr->print(); // Virtual function, binded at runtime
            bptr->show(); //Non-virtual function, binded at compile time
            return 0;
}
```

- In this code, base class pointer ‘bptr’ contains the address of object ‘d’ of derived class.
- Late binding (Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and
- Early binding (Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be bound at runtime (output is *print derived class* as pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time (output is *show base class* as pointer is of base type).

Virtual Function in C++

- A virtual function is a member function which is declared within a base class and is re-defined(Overridden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class’s version of the function.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

- Virtual functions cannot be static.
- A virtual function can be a friend function of another class.
- Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
- The prototype of virtual functions should be the same in the base as well as derived class.
- They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
- A class may have virtual destructor but it cannot have a virtual constructor.

Question

- Create a class named 'PrintNumber' to print various numbers of different datatypes by creating different functions with the same name 'printn' having a parameter for each datatype.
- A boy has his money deposited $1000, $1500 and $2000 in banks-Bank A, Bank B and Bank C respectively. We have to print the money deposited by him in a particular bank.
  Create a class 'Bank' with a function 'getBalance' which returns 0. Make its three subclasses named 'BankA', 'BankB' and 'BankC' with a function with the same name 'getBalance' which returns the amount deposited in that particular bank. Call the function 'getBalance' by the object of each of the three banks.

Operator Overloading

Operator Overloading

- In C++, it can add special features to the functionality and behavior of already existing operators like arithmetic and other operations.
- This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.
- For example, we can overload an operator ‘+’ in a class like String so that we can concatenate two strings by just using +.
- Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

Advantage of Operator overloading

- The advantage of Operators overloading is to perform different operations on the same operand.
- **Operator that cannot be overloaded are as follows:**
  - Scope operator (::)
  - Sizeof
  - member selector(.)
  - member pointer selector(.*)
  - ternary operator(?:)
- Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

- Where the **return type** is the type of value returned by the function.
- **class_name** is the name of the class.
- **operator op** is an operator function where op is the operator being overloaded, and the operator is the keyword.

Operator overloading

- Operator function must be either non-static (member function) or friend function.
- Operator Overloading can be done by using **three approaches**, they are
  1. Overloading unary operator.
  2. Overloading binary operator.
  3. Overloading binary operator using a friend function.

Implementing of Operator overloading:

- **1.** Member function: It is in the scope of the class in which it is declared.
- **2.** Friend function: It is a non-member function of a class with permission to access both private and protected members.