SRM UNIVERSITY AP
Amaravati

Object Oriented Programming using C++

CSE-III

CSE206-OOP

Dr. Priyanka Singh

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.

Difference between Member and friend function:

- **Member function:**
  The number of parameters to be passed is reduced by one, as the calling object is implicitly supplied is an operand.
  - Unary operators tales no explicit parameters.
  - Binary operators take only one explicit parameter.

- **Friend Function:**
  More number of parameters can be passed.
  - Unary operators take one explicit parameter.
  - Binary operators take two explicit parameters.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains at least one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

Overloading Unary Operator

- Let us consider to overload (-) unary operator. In unary operator function, no arguments should be passed. It works only with one class objects. It is overloading of an operator operating on a single operand.

- **Example**:
  Assume that class Distance takes two-member object i.e. feet and inches, create a function by which Distance object should decrement the value of feet and inches by 1 (having single operand of Distance Type)

Unary operator overloading

```
#include <iostream>
using namespace std;
class Distance {
public:
        int feet, inch;
        Distance(int f, int i)
        {
                feet = f;
                inch = i;
        }
        void operator-() // Overloading(-) operator to perform decrement  operation of Distance object
        {       feet--;
                inch--;
                cout << "\nFeet & Inches(Decrement): " << feet << "'" << inch;
        }
};
int main()
{       Distance d1(8, 9);p
        -d1; // Use (-) unary operator by single operand
        return 0;
}
```

this pointer

- Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.
- Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

Using this pointer

```
#include <iostream>
using namespace std;
class Distance {
public:
        int feet, inch;
        Distance(int f, int i)
        {
                this->feet = f;
                this->inch = i;
        }
        void operator-() // Overloading(-) operator to perform decrement  operation of Distance object
        {       feet--;
                inch--;
                cout << "\nFeet & Inches(Decrement): " << feet << "'" << inch;
        }
};
int main()
{       Distance d1(8, 9);p
        -d1; // Use (-) unary operator by single operand
        return 0;
}
```

Overloading Binary Operator

In binary operator overloading function, there should be one argument to be passed. It is overloading of an operator operating on two operands.

```
#include <iostream>
using namespace std;
class A
{   int x;
    public:
    A(){}
    A(int i)
    {       x=i;
    }
    void operator+(A);
    void display();
};
void A :: operator+(A a)
{
     int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;

}
```

```
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

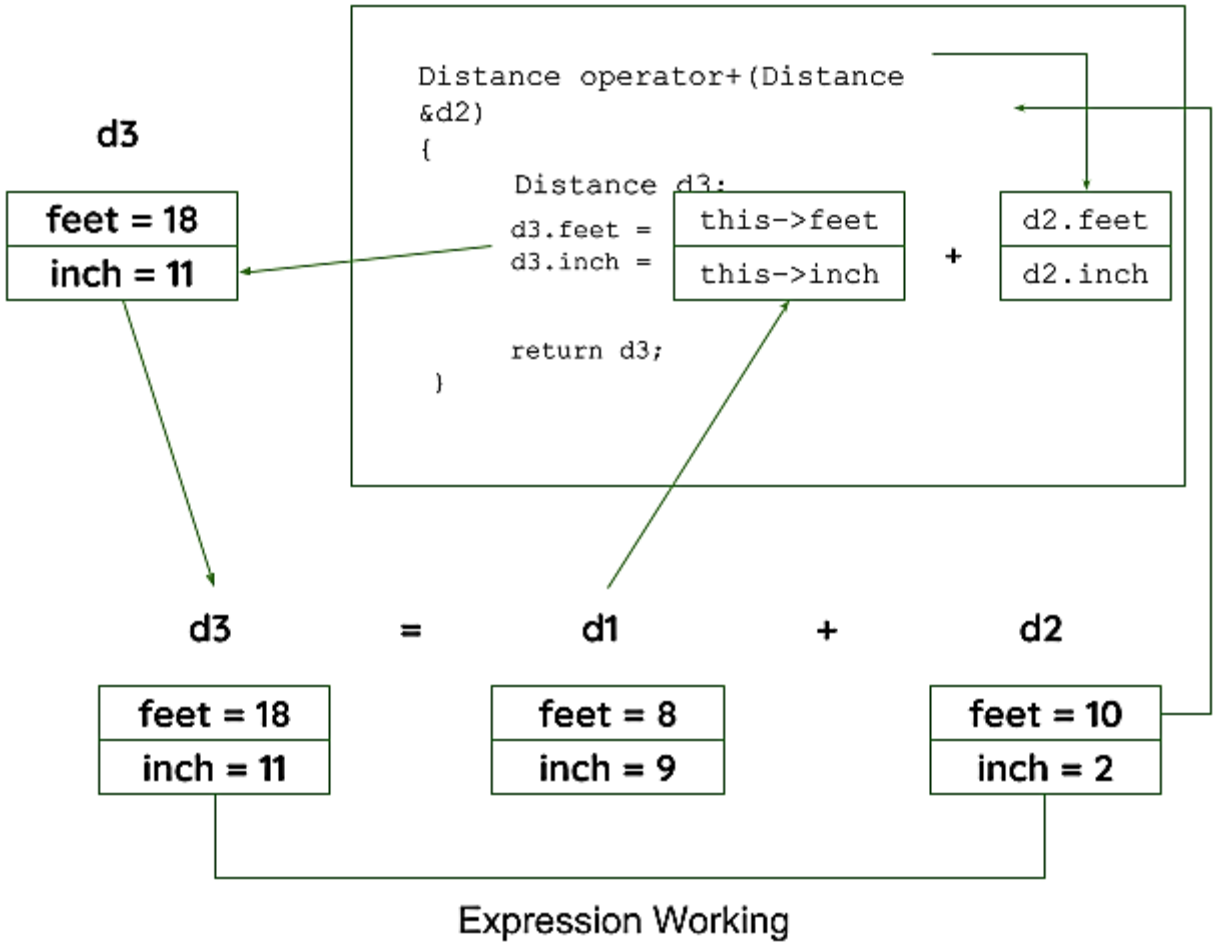
OUTPUT:
9

Overloading Binary Operator: Example

```
#include<iostream>
using namespace std;
class Complex {
private:
        int real, imag;
public:
        Complex(int r = 0, int i =0) {real = r; imag = i;}
        Complex operator + (Complex const &obj) {
                Complex res;
                res.real = real + obj.real;
                res.imag = imag + obj.imag;
                return res;
        }
        void print() { cout << real << " + i" << imag << endl; }
};
int main()
{
        Complex c1(10, 5), c2(2, 4);
        Complex c3 = c1 + c2; // An example call to "operator+"
        c3.print();
}
```

OUTPUT:

12+i9


Distance operator+(Distance
&d2)
{
Distance d3;
d3.feet =
d3.inch =
this->feet
this->inch
+
d2.feet
d2.inch
return d3;
}
d3
feet = 18
inch = 11
d3
=
d1
+
d2
feet = 18
inch = 11
feet = 8
inch = 9
feet = 10
inch = 2
Expression Working

Overloading Binary Operator: Example

Overloading (+) operator to perform addition of two distance object

```
#include <iostream>
using namespace std;
class Distance {
public:
        int feet, inch;
        Distance()
        {
                feet = 0;
                inch = 0;
        }
        Distance(int f, int i)
        {
                feet = f;
                inch = i;
        }
```

```
Distance operator+(Distance& d2) // Call by reference
        {       Distance d3;
                d3.feet = feet + d2.feet;
                d3.inch = inch + d2.inch;
                return d3;
        }
};
int main()
{       Distance d1(8, 9);
        Distance d2(10, 2);
        Distance d3;
        d3 = d1 + d2;
        cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
        return 0;
}
```

Output: `Total Feet & Inches: 18'11`

Overloading Binary Operator using a Friend function:

- In this approach, the operator overloading function must precede with friend keyword, and declare a function class scope.
- Keeping in mind, friend operator function takes two parameters in a binary operator, varies one parameter in a unary operator.
- All the working and implementation would same as binary operator function except this function will be implemented outside of the class scope.

```
#include <iostream>
using namespace std;
class Distance {
public:
        int feet, inch;
        Distance()
        {
                feet = 0;
                inch = 0;
        }
        Distance(int f, int i)
        {
                feet = f;
                inch = i;
        }
        // Declaring friend function using friend keyword
        friend Distance operator+(Distance&, Distance&);
};
```

```
Distance operator+(Distance& d1, Distance& d2) // Call by reference
{       Distance d3;
        d3.feet = d1.feet + d2.feet;
        d3.inch = d1.inch + d2.inch;
        return d3;
}
 int main()
{
        Distance d1(8, 9);
        Distance d2(10, 2);
        Distance d3;
        d3 = d1 + d2;
        cout << "\nTotal Feet & Inches: " << d3.feet << "'" << d3.inch;
        return 0;
}
```

```
Total Feet & Inches: 18'11
```

Disadvantages of an operator overloading:

- In operator overloading, any C++ existing operations can be overloaded, but with some exceptions.
- **Which operators Cannot be overloaded?**

1. Conditional [?:], size of, scope(::), Member selector(.), member pointer selector(.*) and the casting operators.
2. We can only overload the operators that exist and cannot create new operators or rename existing operators.
3. At least one of the operands in overloaded operators must be user-defined, which means we cannot overload the minus operator to work with one integer and one double. However, you could overload the minus operator to work with an integer and a mystring.
4. It is not possible to change the number of operands of an operator supports.
5. All operators keep their default precedence and associations (what they use for), which cannot be changed.
6. Only built-in operators can be overloaded.

Question-1

Assume that class Distance takes two member object i.e. feet and inches, create a function by which Distance object should decrement the value of feet and inches by 1 (having single operand of Distance Type) using friend function.

Overloading Unary Operator using a Friend function:

```
#include <iostream>
using namespace std;
class Distance {
public:
        int feet, inch;
        Distance()
        {
                feet = 0;
                inch = 0;
        }
        Distance(int f, int i)
        {
                feet = f;
                inch = i;
        }
```

```
friend Distance operator--(Distance d)
        {       Distance d2;
                d2.feet=d.feet--;
                d2.inch=d.inch--;
                return d2;
        }
};
int main()
{       Distance d1(8, 9);
        Distance d2;
        d2=--d1;
        cout << "\nFeet & Inches(Decrement): " << d2.feet << "'" << d2.inch;
        return 0;
}
```

Question-2

- Create a class called **time** that has separate **int** member data for **hours, minutes, and seconds**. One constructor should initialize this data to 0, and another should initialize it to fixed values. Another member function should display it, in 11:59:59 format. The final member function should add two objects of type time passed as arguments.

A main() program should create two initialized time objects and one that isn't initialized. Then it should add the two initialized values together, leaving the result in the third time variable. Finally. it should display the value of this third variable. Make appropriate member functions const.

Question-3

- Two strings are anagram of each other, if we can rearrange characters of one string to form another string. All the characters of one string must be present in another string and should appear the same number of times in another string. The strings can contain any ASCII characters.
- **Example**: `rescued and secured, resign and singer, stone and tones, pears and spare, ELEVEN PLUS TWO and TWELVE PLUS ONE.`
- **Write a C++ program using functions, two check if two strings are anagram of each other.**

- **Note:** Your program should work well for strings of any length.

Question-4

- There are N teams participating in a knockout tournament (where if a team loses a game, then it is out of the tournament). Each team is assigned an ID. In this tournament, the ID of the first team is 1, second team is 2, and so on till upto N. The strength of each team is decided by the strength that it contains in the following units:

- Batting
- Bowling
- Fielding
- All-rounder's performance
- Captain's performance

- Now, a team can win against another team if it can win at least 3 units. If a team's strength in a specific unit is higher than the other team, then the former team wins that specific unit. In case, the strength of both teams are the same in a specific unit, then the team that contains the lower team ID wins that unit.
- **The fixture of the tournament is decided as follows:**
- In each round, among the remaining teams, the team that is assigned the lowest team ID plays with the team that contains the next lowest team ID. For example, the team that contains the 3rd lowest ID plays with the team that is assigned the 4th lowest ID and so on.

Write a C++ program using functions that determines the winning team of the tournament and print its respective team ID.

Sample Input:

```
Number of teams participating in the tournament: 4
Individual team statistic/performance:
Team 1: 1 2 3 4 5
Team 2: 2 4 5 6 7
Team 3: 1 2 3 4 5
Team 4: 1 4 2 2 3
```

Output:

```
The winner is Team 2
```

Sample Input:

```
Number of teams participating in the tournament: 7
Individual team statistic/performance:
Team 1: 1 2 3 4 5
Team 2: 2 4 1 6 3
Team 3: 1 2 3 4 5
Team 4: 1 4 2 2 3
Team 5: 3 2 1 4 3
Team 6: 3 4 5 2 6
Team 7: 5 4 2 1 3
```