Object Oriented Programming

LAB 11

FRIEND FUNCTIONS, Friend Classes, Operator Overloading

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

FRIEND FUNCTION

A friend function of a class is defined outside that class's scope, yet has the right to access the non-public (and public) members of the class. Standalone functions, entire classes or member functions of other classes may be declared to be friends of another class.

EXAMPLE

```
#include <iostream>
using namespace std;

class Box
{
        double width;
        public:
        friend void printWidth( Box box );
        void setWidth( double wid );
};

// Member function definition
void Box::setWidth( double wid )
{
        width = wid;
}

// Note: printWidth() is not a member function of any class.
void printWidth( Box box )
{
        /* Because printWidth() is a friend of Box, it can directly access any member of this class */
        cout << "Width of box : " << box.width <<endl;
}

// Main function for the program
int main( )
{
        Box box;
        box.setWidth(10.0); // set box width with member function
        printWidth( box );     // Use friend function to print the wdith.
        return 0;
}
```

FRIEND Classes

A **friend class** can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes.

```
#include <iostream>
using namespace std;
class GFG {
private:
   int private_variable;
protected:
   int protected_variable;
 public:
   GFG()
   {
      private_variable = 10;
      protected_variable = 99;
   }
 friend class F;
};
class F {
public:
   void display(GFG& t)
   {
      cout << "The value of Private Variable = " << t.private_variable << endl;
      cout << "The value of Protected Variable = " << t.protected_variable;
   }
};
int main()
{
   GFG g;
   F fri;
   fri.display(g);
   return 0;
}
```

```
The value of Private Variable = 10
The value of Protected Variable = 99
--------------------------------
Process exited after 0.07882 seconds with return value 0
Press any key to continue . . .
```

Operator Overloading

An operator is said to be overloaded if it is defined for multiple types. In other words, overloading an operator means making the operator significant for a new type.

BUILT IN OVERLOADS

Most operators are already overloaded for fundamental types.

Example:

1) In the case of the expression:

   a / b

   the operand type determines the machine code created by the compiler for the division operator. If both operands are integral types, an integral division is performed; in all other cases floating-point division occurs. Thus, different actions are performed depending on the operand types involved.

2) <<, which is used both as the stream insertion operator and as the bitwise left-shift operator.

OVERLOADS FOR USER DEFIND TYPES

Operators can be used with user-defined types as well. Although C++ does not allow new operators to be created, it does allow most existing operators to be overloaded so that, when they're used with objects, they have meaning appropriate to those objects.

Example:

The effect of + operator can be stipulated for the objects of a particular class.

OPERATOR FUNCTION SYNTAX

To overload an operator, an appropriate *operator function is required.*

```
returntype operator op (arg_list)
{
   function body // task defined
}
```

- **returntype** is the type of value returned by the specified operation.
- **op** is the operator being overloaded. (+,- etc)
- **op** is preceded by the keyword **operator**.

LIST OF OPERATORS THAT CAN BE OVERLOADED IN C++

```
new     delete     new[]      delete[]
+       -      *       /      %       ^       &       |       ~
!       =      <       >      +=      -=      *=      /=      %=
^=      &=     |=      <<     >>      >>=     <<=     ==      !=
<=      >=     &&      ||     ++      --      ,       ->*     ->
()      []
```

LIST OF OPERATORS THAT CAN'T BE OVERLOADED

- **?:** (conditional)
- **.** (member selection)
- **.*** (member selection with pointer-to-member)
- **::** (scope resolution)
- **sizeof** (object size information)
- **typeid** (object type information)

OPERATOR OVERLOADING AS MEMBER FUNCTIONS

If the operator function of a binary operator is defined as a method inside the class, the left operand must always be an object of the class. The operator function is called for this object. The second, right operand is passed as an argument to the method. The method thus has a single parameter.

EXAMPLE

```
//Rupee.h
#include <sstream>      // The class stringstream
#include <iomanip>
#include <iostream>
using namespace std;

class Rupee
{
        private:
                long data;
        public:
                Rupee( int rupee = 0)
                        {
                                data = rupee;
                        }
                    Rupee operator-() const                 // Negation (unary minus)
                            {
                                    Rupee
                                    temp;
                                    temp.data
                                    = -data;
                                    return
                                    temp;
                            }

                    Rupee operator+( const Rupee& obj) const        // addition.
                            {
                                    Rupee temp;
                                    temp.data = data +
                                    obj.data; return
                                    temp;
                            }

                    Rupee operator-( const Rupee& obj) const        // Subtraction.
                            {
                                    Rupee temp;
                                    temp.data = data -
                                    obj.data; return
                                    temp;
                            }
                    Rupee& operator+=( const Rupee& obj)            // Add Rupees.
                            {
                                    data +=
                                    obj.data;
                                    return
                                    *this;
                            }

                    Rupee& operator-=( const Rupee& obj)            // Subtract Rupees.
                            {
                                    data -=
                                    obj.data;
                                    return
                                    *this;
                                }
```

```
        friend ostream &operator<<( ostream &os, const Rupee &e );
};

        ostream& operator<<(ostream& os, const Rupee& e) //Overloading << operator
                {
                        os << e.data;
                        return os;
                }

//TestRupee.cpp
#include "Rupee.h"
#include <iostream>
using namespace std;

int main()
{
        Rupee wholesale(20), retail;
        retail = wholesale;             // Standard assignment
        cout << "Wholesale price: "<<wholesale;
        cout << "\nRetail price: "<<retail;

        Rupee discount(2);
        retail -= discount;
        cout << "\nRetail price including discount: "<<retail;
        wholesale = 34.10;
        cout << "\nNew wholesale price: "<<wholesale;
        retail = wholesale + 10;
        cout << "\nNew retail price: "<<retail;
        Rupee profit( retail - wholesale);
        cout << "\nThe profit: "<<profit;.
        profit = -profit;
        cout << "\nThe profit after unary minus: "<<profit;
        return 0;
        }
```

```
Wholesale price: 20
Retail price: 20
Retail price including discount: 18
New wholesale price: 34
New retail price: 44
The profit: 10
The profit after unary minus: -10
--------------------------------
Process exited after 0.03745 seconds with return value 0
Press any key to continue . . .
```

OPERATOR OVERLOADING AS NON-FUNCTIONS

GENERAL SYNTAX:

```
TYPE1 operator OP(TYPE2 lhs, TYPE3 rhs)
{
}
```

EXAMPLE:

```
//Rupee.h
//Globally Defined
Rupee operator+( const Rupee& e1, const Rupee& e2) // addition.
                {
                        Rupee temp(e1);
                        temp += e2;
                        return temp;
                }
```

ISSUE:

A global function cannot access the private members of the class i.e. data. The function operator+() shown above therefore uses the += operator, whose operator function is defined as a public method. A global operator function can be declared as a "**friend**" of the class to allow it access to the private members of that class.

EXAMPLE:

```
//Inside class Rupee
friend Rupee operator+( const Rupee& e1, const Rupee& e2);


//Globally Defined
Rupee operator+( const Rupee& e1, const Rupee& e2) // addition.
                {
                        Rupee temp; temp.data = e1.data + e2.data;
                        return temp;
                }
```