**Experiment No. 07 and 08**

**Friend Function, Friend Classes and Composition**

**OBJECTIVE**:

Things that will be covered in today’s lab:

* Friend Function and Friend Classes
* Composition

**THEORY**:

**Friend Function and Friend Classes:**

C++ programs are built in a two stage process. First, each source file is *compiled* on its own. The compiler generates intermediate files for each compiled source file. These intermediate files are often called *object files* -- but they are not to be confused with objects in your code. Once all the files have been individually compiled, it then *links* all the object files together, which generates the final binary (the program).

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not apply to *"friends"*.*Friends* are functions or classes declared with the *friend* keyword.

**Friend functions:** A non-member function can access the private and protected members of a class if it is declared a *friend* of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword friend. The general form is

Friend data\_type function\_name( ); // (inside the class)

A friend function is preceded by the keyword *friend*. Properties of *friend* functions are given below:

* *Friend* function is not in the scope of the class to which it has been declared as a *friend*. Hence it cannot be called using the object of that class.
* Usually it has objects as arguments.
* It can be declared either in the public or the private part of a class. It cannot access member names directly. It has to use an object name and dot membership operator with each member name. (e.g., A . x )

Let’s take an example:

|  |
| --- |
| // Add members of two different classes using friend functions  #include <iostream>  using namespace std;  class ClassA {    public:  // constructor to initialize numA to 12  ClassA() : numA(0) {}    private:  int numA;    // friend function declaration  friend int add(ClassA &, ClassA &);  };  // access members of both classes  int add(ClassA &objectA, ClassA &objectB) {  cout<<"Enter 1st number"<<endl;  cin>>objectA.numA;  cout<<"Enter 2nd number"<<endl;  cin>>objectB.numA;  return (objectA.numA + objectB.numA);  }  int main() {  ClassA objectA, objectB;  int sum;  sum=add(objectA, objectB);  cout << "Sum: " << sum;  return 0;  } |

**Friend classes:** Similar to friend functions, a friend class is a class whose members have access to the private or protected members of another class. A class can be made a friend of another class using keyword *friend*.

class A{

friend class B; // class B is a friend class

...

}

class B{

...

}

When a class is made a friend class, all the member functions of that class becomes *friend* functions. In this program, all member functions of class B will be *friend* functions of class A. Thus, any member function of class B can access the private and protected data of class A.

If B is declareda*friend* class of A then, all member functions of class B can access private data and protected data of class A but, member functions of class A cannot private and protected data of class B. Remember, friendship relation in C++ is granted not taken.

**Example:** What should be the output of this program?

|  |
| --- |
| // C++ program to demonstrate the working of friend class  #include <iostream>  using namespace std;  // forward declaration  class ClassB;  class ClassA {  private:  int numA;  // friend class declaration  friend class ClassB;  public:  // constructor to initialize numA to 12  ClassA() : numA(12) {}    };  class ClassB {  private:  int numB;  public:  // constructor to initialize numB to 1  ClassB() : numB(1) {}  // member function to add numA  // from ClassA and numB from ClassB  int add(ClassA &objectB);    };  int ClassB::add(ClassA &objectB) {  cout<<"Enter 1st number"<<endl;  cin>>numB;  cout<<"Enter 2nd number"<<endl;  cin>>objectB.numA;// we can access private variable as ClassA has allowed it  return numB + objectB.numA;  }  int main() {  ClassB obB; ClassA obA;  int sum;  sum=obB.add(obA);  cout << "Sum: " <<sum;  return 0;  }} |

**Composition (aggregation)** is another way to relate two classes. In composition (aggregation), one or more members of a class are objects of another class type. Composition is a “has a” relation; for example “every person has a date of birth”.

class DOB

{

// member variables/ functions

};

class person

{

DOB date; // Declare object of date of birth

// member variable/ functions

};

**Access nested members:**

* You can access the members of “*person”* class using dot operator.
* As *“date”*is the object of *“DOB”* which is a member of *“person”*class, so Members of *“DOB”* can be accessed using “*person”* class object.

// Access member variables

Student.person\_members

Student.date.DOB\_members

Now, let us discuss how the constructors of the objects of date are invoked. Recall that a class constructor is automatically executed when a class object enters its scope. Suppose that we have the following statement:

person student;

When the object “student” enters its scope, the objects “date”, which is the member of the student, also enters their scope. As a result, one of their constructors is executed. We, therefore, need to know how to pass arguments to the constructors of the member objects “date”, which occurs when we give the definitions of the constructors of the class. Recall that constructors do not have a type and so cannot be called like other functions. The arguments to the constructor of a member object are specified in the heading part of the definition of the constructor of the class.

Furthermore, member objects of a class are constructed (that is, initialized) in the order they are declared (not in the order they are listed in the constructor’s member initialization list) and before the containing class objects are constructed. Thus, in our case, the object “date” is initialized first and then “student”.

.

The following statements illustrate how to pass arguments to the constructors of the member objects DOB:

person::person(string f\_n, string l\_n, int m, int d, int y)

:date(m, d, y)

{

// Definition of Constructor person

}

**Example:** What should be the output of this program?

|  |
| --- |
| // Online C++ compiler to run C++ program online  #include <iostream>  #include<string>  using namespace std;  class Name  {  private:  string name;  public:  Name (string cname){  name = cname; }  void printname(){ cout<<name;}  };  class Birthday  {  public:  Birthday(int cmonth, int cday, int cyear){  month = cmonth;  day = cday;  year = cyear;  }  void printDate(){ cout<<month<<"/"<<day <<"/"<<year;}  private:  int month;  int day;  int year;  };  class People  {  public:  People(Name cN, Birthday cdateOfBirth)  :N(cN),dateOfBirth(cdateOfBirth){}  void printInfo(){  N.printname();  dateOfBirth.printDate();  }  private:  Name N;  Birthday dateOfBirth;  };  int main()  {  Birthday birthObject(7,9,97);  Name N("Ali");  People infoObject(N, birthObject);  infoObject.printInfo();  } |

**For all of the exercises below save your code on the learning management system (LMS) and give the screen shot of the output you get on the console in the space provided after every exercise.**

**Exercise 1:** (**15 points)**

Create two classes **DM** and **DB** that store the value of distances. DM stores distance in *meters* and *centimeters* and DB converts them to *feet* and *inches*.

Write a program that can read values for the class objects and converts them. Use a *friend* function to carry out the conversion operation by getting values from the user. The object that stores the results should be a DB object, depending on the units in which the results are required. The display should be in the format of feet and inches.

**Exercise 2:** (**15 points)**

Every Circle has a center and a radius. Create a class **CircleType** that can store the center, the radius, and the color of the circle. Since the center of a circle is a point in the x-y plane, create a class **PointType** to store the x and y coordinate. Use class *PointType* to define the class *CircleType*.

Provide *constructors* that enable objects of these classes to be initialized when they are declared. The constructors should contain default values in case no initializes are provided.

The definition of class *CircleType* and class *PointType* is as under: (you may define additional functions if you require any).

class PointType

{

int x;

int y;

public:

//default constructor

//constructor so that objects are initialized

//print the x and y- coordinates

//determine the quadrant in which the point lies

//getter and setter functions

};

class CircleType

{

double radius;

char \* color;

PointType center;

public:

//default constructor

//constructor so that objects are initialized

//print the radius, center, color

//calculate area

//calculate circumference

//getter and setter functions

//destructor

};

Your program should run for the following main.

int main()

{

CircleType C(21,2,3.5,"blue");

cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n"<<endl;

C.print();

cout<<"\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n"<<endl;

cout<<" Area of circle is "<<C.calc\_area();

cout<<"\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n"<<endl;

PointType P(-20,3);

int p=P.checkquad();

P.print();

switch (p)

{

case 0:

cout<<"Point lies at center"<<endl;

break;

case 1:

cout<<"Point lies in I quadrant"<<endl;

break;

case 2:

cout<<"Point lies in II quadrant"<<endl;

break;

case 3:

cout<<"Point lies in III quadrant"<<endl;

break;

case 4:

cout<<"Point lies in IV quadrant"<<endl;

break;

default:

cout<<"INVALID";

break;

}

int x;

inty;

double r;

char col[9];

CircleType circ(2,5,4.89,"purple");

cout<<"\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

circ.print();

cout<<"\n Enter radius \n";

cin>>r;

cout<<"\n Enter the coordinates where the center lies \n";

cin>>x>>y;

cout<<"\n Enter color \n";

cin>>col;

circ.setparam(x,y,r,col);

cout<<"\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

circ.print();

cout<<"\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

return 0;

}