

**Lab 07**

**Separate Compilation ,Friend Function and Friend Classes**

### OBJECTIVE:

Things that will be covered in today's lab:

- Separate Compilation
- Friend Function and Friend Classes:

### THEORY:

You might be wondering why you need header files and why you would want to have multiple .cpp files for a program. The reasons for this are simple:

- It speeds up compile time.
- It keeps your code more organized
- It allows you to separate *interface* from *implementation*

In C++, the contents of a module consist of structure type (struct) declarations, class declarations, global variables, and functions. The functions themselves are normally defined in a source file (a ".cpp" file). Except for the main module, each source (.cpp) file has a header file (a ".h" file) associated with it that provides the declarations needed by other modules to make use of this module.

The idea is that other modules can access the functionality in module X simply by *#including* the "X.h" header file and the linker will do the rest. The code in X.cpp needs to be compiled only the first time or if it is changed; the rest of the time, the linker will link X's code into the final executable without needing to recompile it, which enables IDEs to work very efficiently.

**Header files** contain mostly declarations that are used in the rest of the program. The skeleton of a class is usually provided in a header file. Header files are not compiled, but rather provided to other parts of the program through the use of *#include*.

A typical header file looks like the following:

```
// Inside sample.h
#ifndef SAMPLE_H
#define SAMPLE_H
// Contents of the header file.
//...
#endif /* SAMPLE_H */
```

**Source File:** An implementation file includes the specific details, that is the definitions, for what is done by the program

```
#include"sample.h"  
    //Definition of function
```

### 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:

```
class Box{  
    double width;  
public:  
    friend void printWidth( Box box );  
    void setWidth(double w) { width=w; }  
};  
void printWidth( Box box ){  
    cout << box.width <<endl;  
}  
void main( ){  
    Box box;  
    box.setWidth(10.0);  
    printWidth( box );  
}
```

**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 declared *afriend* 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?

```
class myclass {
int a, b;
public:
friend int sum(myclass x);
void set_ab(int i, int j);
};
void myclass::set_ab(int i, int j){
a = i;
b = j;
}
int sum(myclass x){
return x.a + x.b;
}
int main(){
myclass n;
n.set_ab(3, 4);
cout << sum(n);
return 0;
}
```

## Exercise 1:

Define a class **Student** that has the following attributes:

- Name: allocated dynamically by a character pointer.
- Roll no: an integer.
- Marks: dynamic array (double type).
- Percentage: a double

Include a constructor that takes values of the *Name*, *Rollno* and *Marks* from user as input. Also include the following methods:

**CalculatePercentage:** that adds all elements of array *Marks* and calculate percentage and stores the result in the member variable *Percentage*.

**Grade:** that calls *CalculatePercentage* method and displays the grade accordingly

Write a driver program to test your class.

## Exercise 2

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

Write a program that can read values for the class objects and add one object of DM with another object of DB. Use a *friend* function to carry out the addition operation. The object that stores the results maybe a DM object or DB object, depending on the units in which the results are required. The display should be in the format of feet and inches or meters and centimeters depending on the object on display.

## Post Lab:

Write a program with a class **integer** that contains an array of integers. Initialize the integer array in the constructor of the class. Then create *friend* functions to the class

- Find the largest integer in the array.
- Find the smallest integer in the array.
- Find the repeated elements in array.
- Sort the elements of array in ascending order.
- Create a destructor that sets all of the elements in the array to 0.