===================================================================

Friend Class and Function in C++

==>

Data hiding is a fundamental concept of object-oriented programming. It restricts the access of private members from outside of the class.

Similarly, protected members can only be accessed by derived classes and are inaccessible from outside. For example,

However, there is a feature in C++ called friend functions that break this rule and allow us to access member functions from outside the class.

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.

We can declare a friend class in C++ by using the friend keyword.

class Demo //Base class

{

//Demo1 is a friend class of Demo

friend class Demo1;

}

class Demo1 // friend class

{

//statements;

}

Syntax:

friend class class\_name; // declared in the base class

========================================================

// C++ Program to demonstrate the functioning of a friend class

#include <iostream>

using namespace std;

class Demo {

private:

int private\_variable;

protected:

int protected\_variable;

public:

Demo()

{

private\_variable = 10;

protected\_variable = 99;

}

// friend class declaration

friend class F;

};

// Here, class F is declared as a friend inside class Demo. Therefore,

// F is a friend of class Demo. Class F can access the private/PROTECTED members of class Demo.

class F {

public:

void display(Demo& t)

{

cout << "The value of Private Variable = "<< t.private\_variable << endl;

cout << "The value of Protected Variable = "<< t.protected\_variable;

}

};

int main()

{

Demo g;

F fri;

fri.display(g);

return 0;

}

Output

The value of Private Variable = 10

The value of Protected Variable = 99

Note: We can declare friend class or function anywhere in the base class body whether its private, protected or public block. It works all the same.

=====================================================================================

Friend Function

==>Like a friend class, a friend function can be granted special access to private and protected members of a class in C++. They are the non-member functions that can access and manipulate the private and protected members of the class in which they are declared as friends.

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

class Demo

{

//private members and functions

}

Friend function has the privileges to access all private and protected members of the class.

A friend function can be:

A global function:

friend return\_type function\_name (arguments); // for a global function

A member function of another class

friend return\_type class\_name::function\_name (arguments); // for a member function of another class

===================================================================

1. Global Function as Friend Function

We can declare any global function as a friend function. The following example demonstrates how to declare a global function as a friend function in C++:

Example:

// C++ program to create a global function as a friend

// function of some class

#include <iostream>

using namespace std;

class base {

private:

int private\_variable;

protected:

int protected\_variable;

public:

base()

{

private\_variable = 10;

protected\_variable = 99;

}

// friend function declaration

friend void friendFunction(baseclass&);

};

// friend function definition

void friendFunction(base& obj)

{

cout << "Private Variable: " << obj.private\_variable

<< endl;

cout << "Protected Variable: " << obj.protected\_variable;

}

// driver code

int main()

{

base object1;

friendFunction(object1);

return 0;

}

Output

Private Variable: 10

Protected Variable: 99

In the above example, we have used a global function as a friend function. In the next example, we will use a member function of another class as a friend function.

==============================================================================================

2 Member Function of Another Class as Friend Function

We can also declare a member function of another class as a friend function in C++. The following example demonstrates how to use a member function of another class as a friend function in C++:

Example:

C++

// C++ program to create a member function of another class

// as a friend function

#include <iostream>

using namespace std;

class base; // forward definition needed.... another class in which function is declared

class anotherClass {

public:

void memberFunction(base&);

};

// base class for which friend is declared

class base {

private:

int private\_variable;

protected:

int protected\_variable;

public:

base()

{

private\_variable = 10;

protected\_variable = 99;

}

// friend function declaration

friend void anotherClass::memberFunction(base&);

};

// friend function definition

void anotherClass::memberFunction(base& obj)

{

cout << "Private Variable: " << obj.private\_variable

<< endl;

cout << "Protected Variable: " << obj.protected\_variable;

}

// driver code

int main()

{

base object1;

anotherClass object2;

object2.memberFunction(object1);

return 0;

}

Output

Private Variable: 10

Protected Variable: 99

Note: The order in which we define the friend function of another class is important and should be taken care of. We always have to define both the classes before the function definition. Thats why we have used out of class member function definition.

======================================================================

Features of Friend Functions

==>A friend function is a special function in C++ that in spite of not being a member function of a class has the privilege to access the private and protected data of a class.

==>A friend function is a non-member function or ordinary function of a class, which is declared as a friend using the keyword “friend” inside the class. By declaring a function as a friend, all the access permissions are given to the function.

==>The keyword “friend” is placed only in the function declaration of the friend function and not in the function definition or call.

==>A friend function is called like an ordinary function. It cannot be called using the object name and dot operator. However, it may accept the object as an argument whose value it wants to access.

==>A friend function can be declared in any section of the class i.e. public or private or protected.

=======================================================

Advantages of Friend Functions

==>A friend function is able to access members without the need of inheriting the class.

==>The friend function acts as a bridge between two classes by accessing their private data.

==>It can be declared either in the public or private or protected part of the class.

Disadvantages of Friend Functions

==>Friend functions have access to private members of a class from outside the class which violates the law of data hiding.

==>Friend functions cannot do any run-time polymorphism

Important Points About Friend Functions and Classes

==>Friendship is not mutual. If class A is a friend of B, then B doesn’t become a friend of A automatically.

i.e Friendship is not inherited.

======================================================================================

Example :

#include <iostream>

using namespace std;

class Box

{

private:

int length;

public:

Box()

{

length=5;

}

friend int printLength(Box); //friend function

};

int printLength(Box b)

{

b.length += 10;

return b.length;

}

int main()

{

Box b;

cout<<"Length of box: "<< printLength(b)<<endl;

return 0;

}

Output:

Length of box: 10

=============================================================

Let's see a simple example when the function is friendly to two classes.

#include <iostream>

using namespace std;

class B; // forward declarartion.

class A

{

int x;

public:

void setdata(int i)

{

x=i;

}

friend void min(A,B); // friend function.

};

class B

{

int y;

public:

void setdata(int i)

{

y=i;

}

friend void min(A,B); // friend function

};

void min(A a,B b)

{

if(a.x<=b.y)

std::cout << a.x << std::endl;

else

std::cout << b.y << std::endl;

}

int main()

{

A a;

B b;

a.setdata(10);

b.setdata(20);

min(a,b);

return 0;

}

Output:

10

In the above example, min() function is friendly to two classes, i.e., the min() function can access the private members of both the classes A and B.

=====================================================================

Example 2: Add Members of Two Different Classes

// Add members of two different classes using friend functions

#include <iostream>

using namespace std;

// forward declaration

class ClassB;

class ClassA {

public:

// constructor to initialize numA to 12

ClassA() : numA(12) {}

private:

int numA;

// friend function declaration

friend int add(ClassA, ClassB);

};

class ClassB {

public:

// constructor to initialize numB to 1

ClassB() : numB(1) {}

private:

int numB;

// friend function declaration

friend int add(ClassA, ClassB);

};

// access members of both classes

int add(ClassA objectA, ClassB objectB) {

return (objectA.numA + objectB.numB);

}

int main() {

ClassA objectA;

ClassB objectB;

cout << "Sum: " << add(objectA, objectB);

return 0;

}

Run Code

Output

Sum: 13

In this program, ClassA and ClassB have declared add() as a friend function. Thus, this function can access private data of both classes.

One thing to notice here is the friend function inside ClassA is using the ClassB. However, we haven't defined ClassB at this point.

// inside classA

friend int add(ClassA, ClassB);

For this to work, we need a forward declaration of ClassB in our program.

// forward declaration

class ClassB;

==================================================================================

C++ Program to swap two members using Friend Function

Approach:

Create a class Swap, declare three variables in it, i.e., a, b, and temp and create a constructor for inputs. Declare a friend function in it. Define the friend function outside the class scope by taking arguments as call by reference to pass the copy of Swap Object. Perform the swap operation with Swap variables.

// C++ Program to swap two numbers using friend function

#include <iostream>

using namespace std;

class Swap {

// Declare the variables of Swap Class

int temp, a, b;

public:

// Define the parameterized constructor, for inputs

Swap(int a, int b)

{

this->a = a;

this->b = b;

}

// Declare the friend function to swap, take arguments as call by reference

friend void swap1(Swap&);

};

// Define the swap function outside class scope

void swap1(Swap& s1)

{

// Call by reference is used to passed object copy to

// the function

cout << "\nBefore Swapping: " << s1.a << " " << s1.b;

// Swap operations with Swap Class variables

s1.temp = s1.a;

s1.a = s1.b;

s1.b = s1.temp;

cout << "\nAfter Swapping: " << s1.a << " " << s1.b;

}

// Driver Code

int main()

{

// Declare and Initialize the Swap object

Swap s(4, 6);

swap1(s);

return 0;

}

Output:

Before Swapping: 4 6

After Swapping: 6 4

========================================================

Approach 2:

Approach:

Create a class Swap, declare one variable in it, i.e., num and create a constructor for inputs. Declare a friend function in it. Define the friend function outside the class scope by taking arguments as call by reference to pass the copy of Swap Object. Perform the swap operation.

//C++ Program to swap data members of two objects of a class using friend function.

#include <iostream>

using namespace std;

class Swap {

// Declare the variable of Swap Class

int num;

public:

// Define the parameterized constructor, for input.

Swap(int num)

{

this->num = num;

}

// Declare the friend function to swap, take arguments

// as call by reference

friend void swap(Swap&, Swap&);

};

// Define the swap function outside class scope

void swap(Swap& s1, Swap& s2)

{

// declare a temporary variable.

int temp;

// Call by reference is used to passed object copy to

// the function

cout << "\nBefore Swapping: " << s1.num << " " << s2.num;

// Swap operations with objects of class Swap

temp = s1.num;

s1.num = s2.num;

s2.num = temp;

cout << "\nAfter Swapping: " << s1.num << " " << s2.num;

}

// Driver Code

int main()

{

// Declare and Initialize the objects of Swap class

Swap s1(6), s2(10);

swap(s1,s2);

return 0;

}

===================================================================

difference between Static Function and Friend Function:

1)In this, it cannot access any variable of its class except for static variables.

1)In this, it can access private and public members of the class.

2)It is denoted by placing a static keyword before the function name.

2)It is denoted by placing a friend keyword before the function name.

3)It can have access to members of one class.

3)It can have access to members of several classes.

=========================================================================================

Namespace

==>Namespace provide the space where we can define or declare identifier i.e. variable, method, classes. Namespace are necessary if you want more than one function with the same name. You can declare two different namespaces for these functions and call them by referencing their corresponding namespace.

==>A namespace is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries.

==>The code generally has multiple libraries, and the namespace helps in avoiding the ambiguity that may occur when two identifiers have the same name.

For example, suppose you have two functions named calculate(), and both are performing different tasks. One calculate() function is doing the multiplication, and another is doing the addition. So in this case, to avoid ambiguity, you will declare both the functions in two different namespaces. These namespaces will differentiate both the functions and also provide information regarding both the functions.

==>The best example of namespace scope is the C++ standard library (std) where all the classes, methods and templates are declared. Hence while writing a C++ program we usually include the directive

using namespace std;

==============================================

Defining a Namespace:

A namespace definition begins with the keyword namespace followed by the namespace name as follows:

namespace namespace\_name

{

// code declarations i.e. variable (int a;)

method (void add();)

classes ( class student{};)

}

==>It is to be noted that, there is no semicolon (;) after the closing brace.

==>Namespace only define them in a global scope.

==>It is

only present in C++ and not in C.

==>To access a class inside a namespace, you can use namespacename::classname.

==>Namespace declarations don’t have access specifiers (Public or Private).

=================================================================

C++ Namespace Example

namespace Data

{

class Manager

{

public:

void task() {}

};

}

You can access the class inside the above namespace in two ways:

1)by using :: operator

Data::Manager mgr;// mgr is an object

Example:

#include <iostream>

using namespace std;

// first name space

namespace first\_space

{

void func()

{

cout << "Inside first\_space" << endl;

}

}

// second name space

namespace second\_space

{

void func()

{

cout << "Inside second\_space" << endl;

}

}

int main ()

{

// Calls function from first name space.

first\_space :: func();

// Calls function from second name space.

second\_space :: func();

return 0;

}

Output

Inside first\_space

Inside second\_space

=======================================================================

2)By using "using" directive

==>You use the "using" directive to import the entire namespace into a program or another namespace. This directive eliminates the need to use the "namespace-name" every time. So it is better to use the "using" if you need several functions or classes from the namespace. Otherwise, if you only need to use it once or twice, "namespace-name::" would be a better choice.

==>The using directive is positioned at the top of the program and it allows us to access all the namespace members.

using namespace Data;

Manager mgr;

mgr.task();

Example:

#include <iostream>

using namespace std;

// first name space

namespace first\_space

{

void func()

{

cout << "Inside first\_space" << endl;

}

}

// second name space

namespace second\_space

{

void func()

{

cout << "Inside second\_space" << endl;

}

}

using namespace first\_space;

int main ()

{

// This calls function from first name space.

func();

return 0;

}

Output

Inside first\_space

==========================================================

// without using std namespace

std::cout

std::cin

std::endl

std::exception

std::bad\_cast

By including the using namespace std; code in our program, we can omit the std:: part for the identifiers defined in the std namespace:

// using std namespace

cout

cin

endl

exception

bad\_cast

===========================================================

Nested Namespaces:

Namespaces can be nested where you can define one namespace inside another name space as follows:

SYNTAX:

namespace namespace\_name1

{

// code declarations

namespace namespace\_name2

{

// code declarations

}

}

You can access members of nested namespace by using resolution operators as follows:

// to access members of namespace\_name2

using namespace namespace\_name1::namespace\_name2;

// to access members of namespace\_name1

using namespace namespace\_name1;

==>In the above statements if you are using namespace\_name1, then it will make elements of namespace\_name2 available in the scope.

Example:

#include <iostream>

using namespace std;

// first name space

namespace first\_space

{

void func()

{

cout << "Inside first\_space" << endl;

}

// second name space

namespace second\_space

{

void func()

{

cout << "Inside second\_space" << endl;

}

}

}

using namespace first\_space::second\_space;

int main ()

{

// This calls function from second name space.

func();

return 0;

}

Output

Inside second\_space

=========================================================

FAQ:

FAQ:

#include <iostream>

using namespace std;

// Variable created inside namespace

namespace first {

int val = 500;

}

// Global variable

int val = 100;

int main()

{

// Local variable

int val = 200;

// These variables can be accessed from outside the namespace using the scope

// operator ::

cout << first::val << '\n';

return 0;

}

Output

500

====================================================

FAQ:

FAQ:

// A C++ program to demonstrate use of class

// in a namespace

#include<iostream>

using namespace std;

namespace ns

{

// A Class in a namespace

class geek

{

public:

void display()

{

cout<<"ns::geek::display()"<<endl;;

}

};

}

int main()

{

// Creating Object of geek Class

ns::geek obj;

obj.display();

return 0;

}

Output

ns::geek::display()

================================================================

FAQ:

FAQ:

A class can also be declared inside namespace and defined outside namespace using the following syntax:

CPP

// A C++ program to demonstrate use of class

// in a namespace

#include <iostream>

using namespace std;

namespace ns {

// Only declaring class here

class geek;

} // namespace ns

// Defining class outside

class ns::geek {

public:

void display() { cout << "ns::geek::display()\n"; }

};

int main()

{

// Creating Object of geek Class

ns::geek obj;

obj.display();

return 0;

}

Output

ns::geek::display()

We can define methods as well outside the namespace. The following is an example code:

C++

// A C++ code to demonstrate that we can define

// methods outside namespace.

#include <iostream>

using namespace std;

// Creating a namespace

namespace ns {

void display();

class geek {

public:

void display();

};

} // namespace ns

// Defining methods of namespace

void ns::geek::display()

{

cout << "ns::geek::display()\n";

}

void ns::display() { cout << "ns::display()\n"; }

// Driver code

int main()

{

ns::geek obj;

ns::display();

obj.display();

return 0;

}

Output:

ns::display()

ns::geek::display():

==========================================================

Standard Namespace

The std is a short form of standard, the std namespace contains the built-in classes and declared functions.

You can find all the standard types and functions in the C++ "std" namespace.

#include<iostream>

int main()

{

std::cout<<"enter no to add";

int num1,num2;

std::cin>>num1;

std::cin>>num2;

std::cout<<"addition is"<<num1+num2;

}

Here std is used in front of cin and cout along with scope resolution operator, which indicates that the object cin and cout are defined inside the namespace whose name is std.The std is the standard library, and both cin and cout are defined inside this scope.

====================================================================

Advantage

The primary advantage of namespaces is that they resolve any naming conflict. For example, sometimes, you may need more than one function with the same name. And namespaces provide a way to declare such functions without making the compiler ambiguous.

====================================================================

FAQ::

FAQ::

FAQ)DIFFERENCE BETWEEN CLASS AND NAMESPACE

========================================================================

You can have unnamed namespaces but you can’t have a unnamed class.For example:

namespace { // fine

// some code....

}

class { // illegal

}You can have unnamed namespaces but you can’t have a unnamed class.For example:

namespace { // fine

// some code....

}

class { // illegal

}

================================================

=======================================================================

operator overloading

COMPLEX C1(1,2);

COMPLEX C2(3,4)

COMPLEX C3=C1+C2 ; //C1.SHOW(C2);

// C1.SHOW(C2);

INT A=10;

INT B=20;

INT C=A+B;

Suppose there is a need to add two complex numbers, a function Add() can be written in the class and used as follows:

c3=c1.Add(c2);// c1,c2,c3 are the objects of complex class

==>However, it would be more natural if it is written as c1+c2.But the + operator in c++ works only for numeric built-in datatypes and c1 and c2 are user defined datatypes.As compiler doesNot know the behaviour of any operator with UDTs ,if an operator is used with them,compiler gives an error.

==>To overcome such problems,c++ provides the feature of operator overloading. The '+' operator can be overloaded for a class and how to add objects can be specified.This bringd UDTs closer to built-in types.

==>operator overloading giv

es additional meaning to the existing operator like +,-,\*,/ etc. Giving additional meaning doesnot mean a radical change,but an extension in the default behaviour of the operator. operator overloading helps the compiler to perform this type of operation when it is applied on UDTs. With the help of operator oerloading one can use the existing operators on UDTs.

==>By using operator overloading we can add 2 userdefined datatypes also.we are giving extra task to an existing operator.

it gives additional meaning to the existing operator.this features makes UDTs more natural and close to built in datatypes.

==>the main idea behind “Operator overloading” is to use C++ operators with class variables or class objects. Redefining the meaning of operators really does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

==>operator keyword is used to implement operator overloading.overloaded operators are implemented as function.An overloaded operator is called an operator function. An operator function is declared with the keyword operator preceding the operator.

ex:to overload the addition operator,a function called operator+ can be defined and implemented.

syntax:

returntype operator symbol(arg-list)

{

}

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.

Rules and Regulations

========================================================================================

1)New operators cannot be defined or invented ..extending the meaning of the operators doesnot mean giving new meaning to the operators.

2)one cannot change the precedence or associativity of the operators.

3)one cannot change the arity of the operator.

4)we cannot overload membership operator(.),scope resolution operator(::),ternary operator,sizeof operator.

5) For operator overloading to work, at least one of the operands must be a user-defined class object.

1+2

1+C1;

C1+5

================================================

#include<iostream>

using namespace std;

class complex

{

int real,img;

public:

complex();

complex(int,int);

void display();

complex operator+(complex&);

complex operator-(complex&);

complex operator-();

};

complex::complex()

{

real=img=0;

}

complex::complex(int real,int img)

{

this->real=real;

this->img=img;

}

void complex::display()

{

if(img>0)

cout<<"complex number is "<<real<<"+"<<img<<"i"<<endl;

else

cout<<"complex number is "<<real<<img<<"i"<<endl;

}

complex complex::operator+(complex & c)

{

complex temp;

temp.real=this->real + c.real;

temp.img=this->img + c.img;

return temp;

}

complex complex::operator-(complex & c)

{

complex temp;

temp.real=this->real - c.real;

temp.img=this->img -c.img;

return temp;

}

complex complex::operator-()

{

complex temp;

temp.real=-this->real;

temp.img=-this->img;

return temp;

}

int main()

{

complex c1(1,2);

complex c2(3,4);

complex c3=c1-c2;//c3=c1.operator-(c2)

c3.display();

complex c4(5,6);

complex c5=-c4;//c5=c4.operator-()

c5.display();

}