Simple program showing the working of the friend function:

```
#include <iostream>

using namespace std;

class Distance {

    private:

        int meter;

        // friend function

        friend int addFive(Distance);

    public:

        Distance() : meter(0) {}

};

// friend function definition

int addFive(Distance d) {

    //accessing private members from the friend function

    d.meter += 5;

    return d.meter;

}

int main() {

    Distance D;

    cout << "Distance: " << addFive(D);

    return 0;

}
```

```
// 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(base obj);
};

// 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;
}
```

```
};

#include <iostream>

using namespace std;

// forward declaration

class ClassB;

class ClassA {

        Public:

        //This is a constructor of a C++ class named "ClassA". The constructor initializes a
//member variable named "numA" with the value 12.
        //ClassA() : numA(12) {}

        // 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 = 8;
        }

        private:

        int numB;

        // friend function declaration

        friend int add(ClassA, ClassB);

};
```

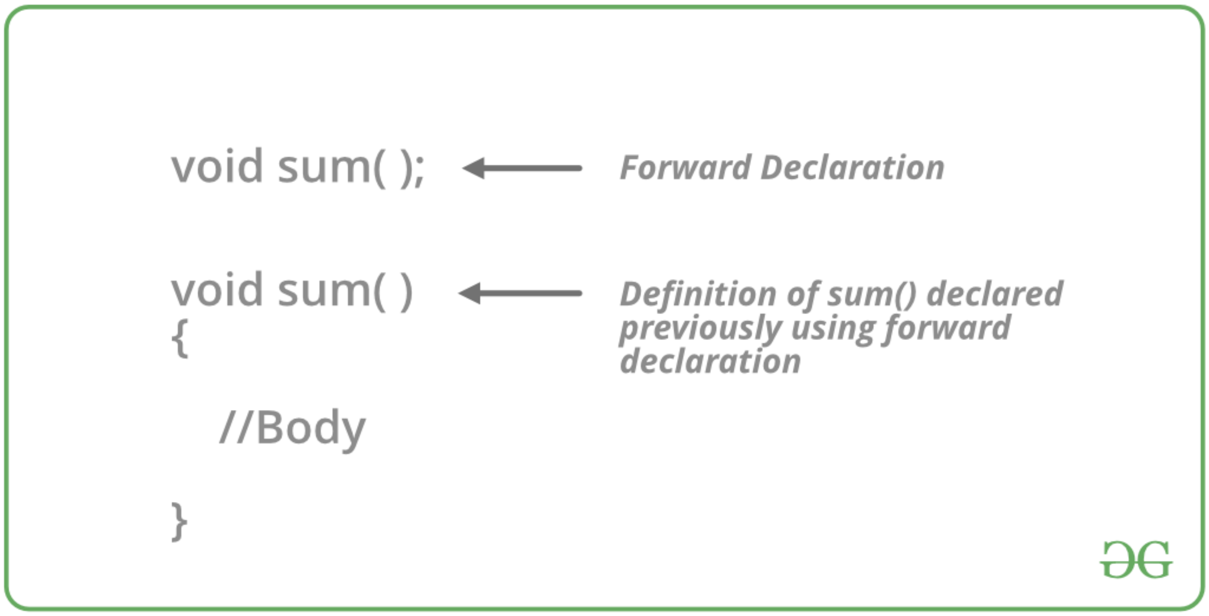
```
// access members of both classes

int add(ClassA objectA, ClassB objectB) {

        return (objectA.numA + objectB.numB);

}

int main() {

        ClassA objectA;

        ClassB objectB;

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

        return 0;

}
```

***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 objectA;

        return objectA.numA + numB;

        }

};

int main() {

        ClassB objectB;

        cout << "Sum: " << objectB.add();

        return 0;

}
```

## What are Forward declarations in C++

**Forward Declaration** refers to the beforehand declaration of the syntax or signature of an identifier, variable, function, class, etc. prior to its usage (done later in the program).


void sum( );
Forward Declaration
void sum( )
{
//Body
}
Definition of sum() declared previously using forward declaration

**Example:**

```
// Forward Declaration of the sum()
void sum(int, int);

// Usage of the sum
void sum(int a, int b)
{
    // Body
}
```

In C++, Forward declarations are usually used for Classes. In this, the class is pre-defined before its use so that it can be called and used by other classes that are defined before this.

**Example:**

```
// Forward Declaration class A
class A;

// Definition of class A
class A{
    // Body
};
```

## Need for Forward Declarations:

Let us understand the need for forward declaration **with an example.**

**Example 1:**

```
.
// C++ program to show
// the need for Forward Declaration

#include <iostream>
    using namespace std;

class B {

public:
    int x;

    void getdata(int n)
    {
        x = n;
    }
    friend int sum(A, B);
};

class A {
public:
    int y;

    void getdata(int m)
    {
        y = m;
    }
    friend int sum(A, B);
};

int sum(A m, B n)
{
    int result;
    result = m.y + n.x;
    return result;
}

int main()
```

```
 {
     B b;
     A a;
     a.getdata(5);
     b.getdata(4);
     cout << "The sum is : " << sum(a, b);
     return 0;
 }
```

**Output:**

```
Compile Errors :
prog.cpp:14:18: error: 'A' has not been declared
   friend int sum(A, B);
                  ^
```

**Explanation:** Here the compiler throws this error because, in class B, the object of class A is being used, which has no declaration till that line. Hence the compiler couldn’t find class A. So what if class A is written before class B? Let’s find out in the next example.

**Example 2:**

```
.
// C++ program to show
// the need for Forward Declaration

#include <iostream>
    using namespace std;

class A {
public:
    int y;

    void getdata(int m)
    {
        y = m;
    }
    friend int sum(A, B);
};

class B {

public:
    int x;

    void getdata(int n)
    {
        x = n;
    }
    friend int sum(A, B);
};

int sum(A m, B n)
{
    int result;
    result = m.y + n.x;
    return result;
}

int main()
{
    B b;
    A a;
    a.getdata(5);
    b.getdata(4);
    cout << "The sum is : " << sum(a, b);
    return 0;
 }
```

**Output:**

```
Compile Errors :
prog.cpp:16:23: error: 'B' has not been declared
    friend int sum(A, B);
                     ^
```

**Explanation:** Here the compiler throws this error because, in class A, the object of class B is being used, which has no declaration till that line. Hence the compiler couldn't find class B.

**Now it is clear that any of the above codes wouldn't work, no matter in which order the classes are written. Hence this problem needs a new solution- Forward Declaration.**

Let us add the forward declaration to the above example and check the output again.

**Example 3:**

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

// Forward declaration
class A;
class B;

class B {
    int x;

public:
    void getdata(int n)
    {
        x = n;
    }
    friend int sum(A, B);
};

class A {
    int y;

public:
    void getdata(int m)
    {
        y = m;
    }
    friend int sum(A, B);
};
int sum(A m, B n)
{
    int result;
    result = m.y + n.x;
    return result;
}

int main()
{
    B b;
    A a;
    a.getdata(5);
    b.getdata(4);
    cout << "The sum is : " << sum(a, b);
    return 0;
 }
```

**Output:**

```
The sum is : 9
```

The program runs without any errors now. A **forward declaration** tells the compiler about the existence of an entity before actually defining the entity. Forward declarations can also be used with other entity in C++, such as functions, variables and user-defined types.

## What is a Static Function?

A function that is declared static using the '**static**' keyword becomes a static function in C++.

**Syntax of the Static Function:**

```
static <return_type> <function_name>(<arguments>){
   //code
}
```

When a function inside a class is declared as static, it can be accessed outside the class using the class name and scope resolution operator (**::**), without creating any object.

A static member method has access only to the static members of the class, we can not call any non-static functions inside it.

All objects in the class share the same copy of the static function. It is useful in manipulating global static variables, which are again shared by all objects.

Since the static function is not bounded by the object of the class, it does not has access to the **this** keyword.

## C++ Example: Static Function

Let see an example using static function for the better understanding.

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

class Nokia{
private:
   /*
      *declaring variable as staric, so that-
      *it will be accessible to the static functions
   */
   static string phone_name;
public:
   //static function I
```

```
    static void name(){
        cout << "Phone: "<< phone_name;
    }

    //static function II
    static void set_name(string name){
        phone_name = name;
    }
};

//Initializing private static member
string Nokia::phone_name = "";

int main()
{
    //no object jas been created
    //accessing static function directly with class name
    Nokia::set_name("Nokia 2600");
    Nokia::name();

    return 0;
}
```

**Output**:

```
Phone: Nokia 2600
```

In the above program, we are setting a string static variable via a static function without creating any object, because the static variable (phone_name) cannot be set through a non-static method.

**Example : Let's consider an example to access the static member function //using the object and class in the C++ programming language.**

```
#include <iostream>
using namespace std;
class Member
{

private:
// declaration of the static data members
static int A;
static int B;
static int C;

// declare public access specifier
public:
// define the static member function
static void disp ()
{
cout << " The value of the A is: " << A << endl;
cout << " The value of the B is: " << B << endl;
cout << " The value of the C is: " << C << endl;
}
};
// initialization of the static data members
int Member :: A = 20;
int Member :: B = 30;
int Member :: C = 40;

int main ()
{
// create object of the class Member
Member mb;
// access the static member function using the class object name
cout << " Print the static member through object name: " << endl;
mb. disp();
// access the static member function using the class name
cout << " Print the static member through the class name: " << endl;
Member::disp();
return 0;
}
```

Assignment and Copy Initialization

The Copy initialization can be done using the concept of copy constructor. As we know that the constructors are used to initialize the objects. We can create our copy constructor to make a copy of some other object, or in other words, initialize current object with the value of another object. On the other hand, the direct initialization can be done using assignment operation.

The main difference between these two types of initialization is that the copy initialization creates a separate memory block for the new object. But the direct initialization does not make new memory space. It uses reference variable to point to the previous memory block.

## Copy Constructor Or Copy Initialization (Syntax)

```
classname (const classname &obj) {
   // body of constructor
}
```

## Direct Initialization or Assignment Operator (Syntax)

```
classname Ob1, Ob2;
Ob2 = Ob1;
```

Let us see the detailed differences between Copy initialization and Direct initialization.

| **Copy initialization** | **Direct Initialization** |
|---|---|
| The Copy initialization is basically an overloaded constructor | Direct initialization can be done using assignment operator. |
| This initializes the new object with an already existing object | This assigns the value of one object to another object both of which are already exists. |
| Copy initialization is used when a new object is created with some existing object | This is used when we want to assign existing object to new object. |
| Both the objects uses separate memory locations. | One memory location is used but different reference variables are pointing to the same location. |
| If no copy constructor is defined in the class, the compiler provides one. | If the assignment operator is not overloaded then bitwise copy will be made |

```
#include<iostream>
using namespace std;
class Example {
  int x, y; //data members
  public:
  Example(int a, int b) {
        x = a;
        y = b;
  }
  /* Copy constructor */
  Example(Example & ex) {
        x = ex.x;
        y = ex.y;
  }
  void display() {
        cout << x << " " << y << endl;
  }
};
/* main function */
int main() {
  Example obj1(20, 30); // Normal constructor
  Example obj2 = obj1; // Copy constructor
  cout << "Normal constructor : ";
  obj1.display();
  cout << "Copy constructor : ";
  obj2.display();
  return 0;
}
```

**OUTPUT:**

```
Normal constructor : 20 30
Copy constructor : 20 30
```

**‘this’ Pointer in C++**

“this” is a keyword that is an implicit pointer. “this” pointer points to the object which calls the member function.
In C++ programming, **this** is a keyword that refers to the current instance of the class.

this pointer in C++ stores the address of the class instance, which is called from the member function that enables functions to access the correct object data members. We don't need to define the this pointer in C++ as a function argument in the class, and the compiler implicitly does it for us. this pointer can be accessed from every class function, including the constructor.

**NOTE:**
Friend functions do not have a **this** pointer, because friends are not members of a class. Only member functions have a **this** pointer.

```
#include<iostream>
using namespace std;
class A{
        int a;
        public:
        void setData(int a){
        a = a;
        }

        void getData(){
        cout<<"The value of a is "<<a<<endl;
        }
};
int main(){
        A a;
        a.setData(4);
        a.getData();
        return 0;
}
```

```
#include<iostream>
using namespace std;
class A{
        int a;
        public:
        void setData(int a){
        this->a = a;
        }
        void getData(){
        cout<<"The value of a is "<<a<<endl;
        }
};
int main(){
        A a;
        a.setData(4);
        a.getData();
        return 0;
}
```

This C++ program differentiates between the concrete and abstract class. An abstract class is meant to be used as a base class where some or all functions are declared purely virtual and hence can not be instantiated. A concrete class is an ordinary class which has no purely virtual functions and hence can be instantiated.

Here is the source code of the C++ program which differentiates between the concrete and abstract class. The C++ program is successfully compiled and run on a Linux system. The program output is also shown below.

```
/*
 * C++ Program to differentiate between concrete class and abstract class
 */
#include <iostream>
#include <string>
using namespace std;

class Abstract {
  private:
      string info;
  public:
      virtual void printContent() = 0;
};

class Concrete {
  private:
      string info;
  public:
      Concrete(string s) : info(s) { }
      void printContent() {
          cout << "Concrete Object Information\n" << info << endl;
      }
};

int main()
{
  /*
   * Abstract a;
   * Error : Abstract Instance Creation Failed
   */
```

```
    string s;

    s = "Object Creation Date : 23:26 PM 15 Dec 2013";
    Concrete c(s);
    c. printContent();
}
```

OUTPUT:

```
Concrete Object Information

Object Creation Date : 23:26 PM 15 Dec 2013
```

**Abstract class** can not be used to create an object. Whereas, **concrete class** can be used to create an object.
An *abstract* class is one that has one or more pure virtual functions. Whereas a *concrete* class has no pure virtual functions.
A concrete class is a class that can be used to create an object. An abstract class cannot be used to create an object (you must extend an abstract class and make a concrete class to be able to then create an object).