

FRIEND FUNCTIONS

Friend function



FRIEND FUNCTIONS

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.

Friend function can be

- Standalone functions
- Entire classes or
- Member functions of other classes may be declared to be friends of another class.

FRIEND FUNCTIONS

The functions which are not member functions of the class yet they can access all private members of the class are called friend functions.

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

The compiler knows a given function is a friend function by the use of the keyword friend.

ARE FRIEND FUNCTIONS AGAINST THE CONCEPT OF OBJECT ORIENTED PROGRAMMING?

One of the important concepts of OOP is data hiding, i.e., a nonmember function cannot access an object's private or protected data.

But, sometimes this restriction may force programmer to write long and complex codes. It can be said that friend functions are against the principle of object oriented programming because they violate the principle of encapsulation which clearly says that each object methods and functions should be encapsulated in it. But there we are making our private member accessible to other outside functions.

GLOBAL FRIEND FUNCTION

```
#include <iostream>
using namespace std;
class Me{
    int wallet;
    int marks;
public:
    friend void Bro ( Me me );
    void setWallet( int amount )
    { wallet=amount;
};
void Bro( Me me ) {
    cout << "Friend Accessing my wallet :( " << me.wallet <<endl;}
```

```
int main() {  
    Me me;  
    me.setWallet(100);  
    Bro( me ); //I hate you bro  
    return 0;  
}
```

FRIEND FUNCTION

Prototypes of friend functions appear in the class definition. But friend functions are NOT member functions.

Friend functions can be placed anywhere in the class without any effect Access specifiers don't affect friend functions or Classes.

FRIEND FUNCTION OF ANOTHER CLASS

Member functions of other classes may be declared to be friends of another class

FRIEND FUNCTION OF ANOTHER CLASS

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

class MyFriend;
class Me {
    int wallet;
    int marks;
public:
    void Bro(MyFriend x);
    void setWallet(int amount) {
        wallet = amount;
    }
};
```

FRIEND FUNCTION OF ANOTHER CLASS

```
class MyFriend {  
    int toys;  
    int bank;  
public:  
    friend void Me::Bro(MyFriend x);  
    void setBank(int amount) {  
        bank = amount;  
    }  
};
```

```
void Me::Bro(MyFriend x) {  
    cout << "Hey Bro, I can check your bank account: " << x.bank << endl;  
}
```

FRIEND FUNCTION OF ANOTHER CLASS

```
int main() {  
    Me me;  
    me.setWallet(100);  
    MyFriend myfriend;  
    myfriend.setBank(1000);  
    me.Bro(myfriend);  
    return 0;  
}
```

MULTIPLE FRIENDS

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

class MyFriend;
class Me {
    int wallet;
    int marks;
public:
    friend void SuperBro(Me me, MyFriend x);
    void setWallet(int amount) {
        wallet = amount;
    }
};
```

MULTIPLE FRIENDS

```
class MyFriend {  
    int toys;  
    int bank;  
public:  
    friend void SuperBro(Me me, MyFriend x);  
    void setBank(int amount) {  
        bank = amount;  
    }  
};
```

MULTIPLE FRIENDS

```
void SuperBro(Me me, MyFriend x) {  
    cout << "Hey, I am SuperBro"<<endl;  
    cout<<"I can check your wallet and also bank account: "  
    << me.wallet<<" "<< x.bank << endl;  
}
```

```
int main() {  
    Me me;    me.setWallet(100);  
    MyFriend myfriend;    myfriend.setBank(1000);  
    SuperBro(me,myfriend);  
    return 0;  
}
```

FRIEND CLASSES

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

To declare all member functions of class ClassTwo as friends of class ClassOne, place a declaration of the form

```
friend class ClassTwo;
```

in the definition of class ClassOne.

FRIEND CLASSES

```
class B;  
class A  
{  
    // class B is a friend class of class A  
    friend class B;  
    ... ..  
}  
class B  
{  
    ... ..  
}
```


FRIEND CLASSES

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.

But, member functions of class A cannot access the data of class B.

ONE SIDED FRIENDSHIP

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

class MyFriend;
class Me {
    int wallet;
    int marks;
public:
    friend class MyFriend;
    void setWallet(int amount) {
        wallet = amount;}
    void display()
    {
        cout<<"Amount: "<<wallet;
    }
};
```

ONE SIDED FRIENDSHIP

```
class MyFriend {
    int toys;
    int bank;
public:
    void OneSidedFreindship(Me me)
    {
        cout<<"I am Mean friend"<<endl;
        cout<<"I can access all data of Me class but doesn't share mine :p"<<endl;
        cout<<me.wallet<<endl;
    }
    void LetsHaveATreat(Me &me)
    {
        cout<<"hehehe"<<endl;
        cout<<"I am ordering a pizza using my friend wallet :p"<<endl;
        me.wallet=0;
    }
    void setBank(int amount) {
        bank = amount;}
};
```

ONE SIDED FRIENDSHIP

```
int main() {  
    Me me;  
    me.setWallet(100);  
    MyFriend myfriend;  
    myfriend.setBank(1000);  
    myfriend.OneSidedFreindship(me);  
    myfriend.LetsHaveATreat(me);  
    me.display();  
    return 0;  
}
```

FRIEND CLASSES

Friendship is granted, not taken—i.e.,

For class B to be a friend of class A, class A must explicitly declare that class B is its friend.

Friendship is not mutual unless explicitly specified.

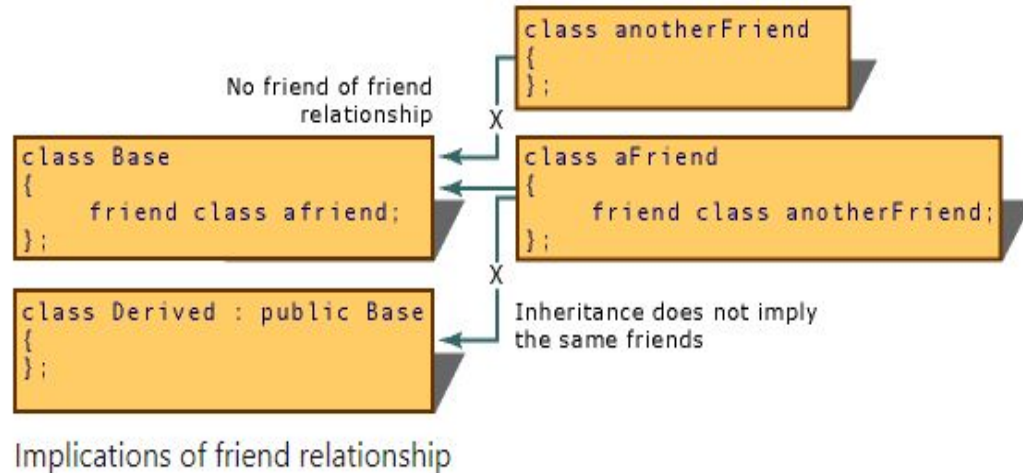
Friendship is not transitive.

If class A is a friend of class B, and class B is a friend of class C, you cannot infer that class A is a friend of class C

FRIEND CLASSES

Friendship is not inherited

If a base class has a friend function, then the function doesn't become a friend of the derived class(es).



MUTUAL FRIENDSHIP

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

class MyFriend;
class Me {
    int wallet;
    int marks;
public:
    friend class MyFriend;
    void setWallet(int amount) {
        wallet = amount;}
    void display()
    {
        cout<<"Amount: "<<wallet;
    }
    void MutualFreindship(MyFriend myfriend);

    void LetsHaveATreat(MyFriend &myfriend);
};
```

MUTUAL FRIENDSHIP

```
class MyFriend {
    int toys;
    int bank;
public:
    friend class Me;
    void MutualFreindship(Me me)
    {
        cout<<"We are mutual friend"<<endl;
        cout<<me.wallet<<endl;
    }
    void LetsHaveATreat(Me &me)
    {
        cout<<"hehehe"<<endl;
        cout<<"I am ordering a pizza using my friend wallet :p"<<endl;
        me.wallet=0;
    }
    void setBank(int amount) {
        bank = amount;
    }
};
```


MUTUAL FRIENDSHIP

```
void Me::MutualFreindship(MyFriend myfriend)
{
    cout<<"We are mutual friend"<<endl;
    cout<<myfriend.bank<<endl;
}
void Me::LetsHaveATreat(MyFriend &myfriend)
{
    cout<<"hehehe"<<endl;
    cout<<"I am ordering a pizza using my friend bank account :p"<<endl;
    myfriend.bank=0;
}
int main() {
    Me me;
    me.setWallet(100);
    MyFriend myfriend;
    myfriend.setBank(1000);
    myfriend.MutualFreindship(me);
    me.MutualFreindship(myfriend);
    return 0;
}
```

WHY THEY ARE NEEDED?

They are needed in situations where we have written code for some function in one class and it need to be used by other classes as well for example, suppose we wrote the code to compute a complex mathematical formula in one class but later it was required by other classes as well, in that case we will make that function friend of all other classes.

OVERLOADED FRIEND FUNCTIONS

- It's possible to specify overloaded functions as friends of a class.
- Each function intended to be a friend must be explicitly declared in the class definition as a friend of the class.

OVERLOADED FRIEND FUNCTIONS

```
class Distance {  
    int meter;  
    friend int addFive(Distance);  
    friend int addFive(Distance , int);  
public:  
    Distance() : meter(0) {}  
int addFive(Distance d) {  
    d.meter += 5;  
    return d.meter;}  
int addFive(Distance d , int i) {  
    d.meter += i;  
    return d.meter;}  
}
```

OVERLOADED FRIEND FUNCTIONS

```
int main() {  
    Distance D;  
    cout << "Distance: " << addFive(D)  
    <<endl;;  
    cout << "Distance overloaded: " <<  
    addFive(D,6);  
    return 0;}
```

OPERATOR FUNCTIONS AS CLASS MEMBERS VS. GLOBAL FUNCTIONS

Operator functions can be member functions(already discussed)

Or global functions (non-member function)

global functions are often made friends for performance reasons.

PERFORMANCE REASONS

It's possible to overload an operator as a global, non-friend function, but such a function requiring access to a class's private or protected data would need to use set or get functions provided in that class's public interface. The overhead of calling these functions could cause poor performance, so these functions can be inlined to improve performance.

SOME CRITERIA/RULES TO DEFINE THE OPERATOR FUNCTION

Operator overloading function can be applied on a member function if the left operand is an object of that class, but if the Left operand is different, then the Operator overloading function must be defined as a non-member function.

Member functions use `this` pointer implicitly to obtain one of their class object arguments (the left operand for binary operators).

Arguments for both operands of a binary operator must be explicitly listed in a global function call

SOME CRITERIA/RULES TO DEFINE THE OPERATOR FUNCTION:

In case of a non-static function, the binary operator should have only one argument and unary should not have an argument.

In the case of a friend function, the binary operator should have only two argument and unary should have only one argument.

SYNTAX FOR BINARY OPERATOR OVERLOADING USING FRIEND FUNCTION

```
friend return-type operator operator-symbol  
(Variable 1, Variable 2)  
{  
    //Statements;  
}
```

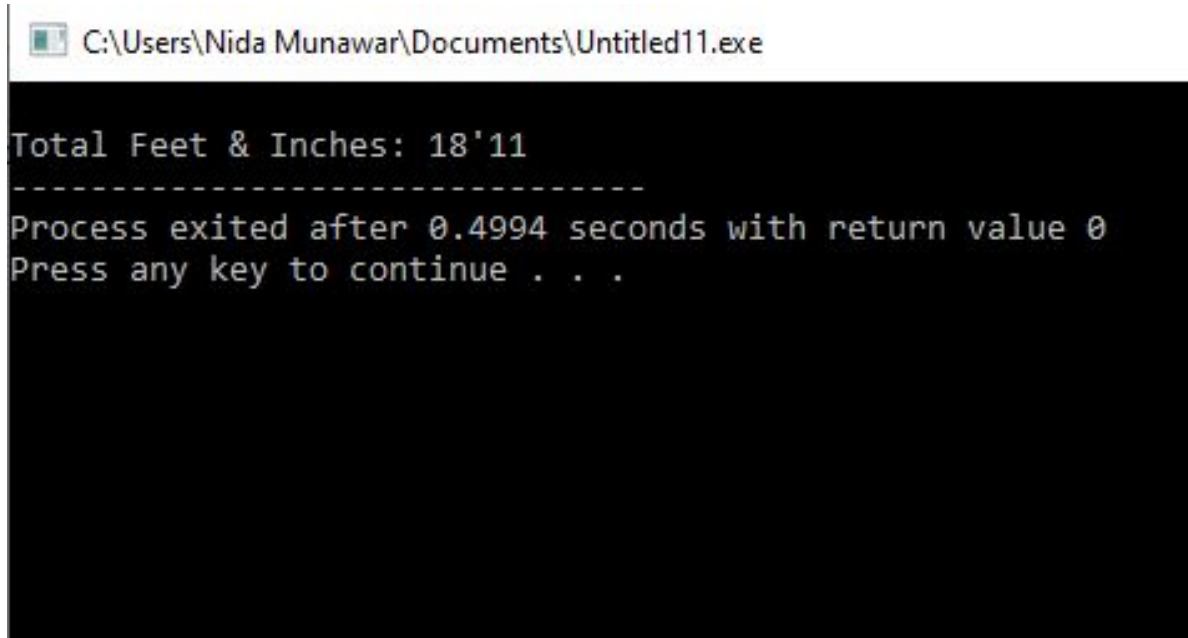
OVERLOADING BINARY OPERATOR USING A FRIEND FUNCTION

```
class Distance {  
public:  
    int feet, inch;  
    Distance() {  
        this->feet = 0;  
        this->inch = 0; }  
    Distance(int f, int i) {  
        this->feet = f;  
        this->inch = i; }  
    friend Distance operator+(Distance, Distance);  
};
```

OVERLOADING BINARY OPERATOR USING A FRIEND FUNCTION

```
Distance operator+(Distance d1, Distance d2) {  
    Distance d3;  
    d3.feet = d1.feet + d2.feet;  
    d3.inch = d1.inch + d2.inch;  
    return d3; }  
  
int main() {  
    Distance d1(8, 9);  
    Distance d2(10, 2);  
    Distance d3;  
    d3 = d1 + d2;  
    cout << "\nTotal Feet & Inches: " << d3.feet << " " <<  
d3.inch;  
    return 0; }
```

OVERLOADING BINARY OPERATOR USING A FRIEND FUNCTION



```
C:\Users\Nida Munawar\Documents\Untitled11.exe  
  
Total Feet & Inches: 18'11  
-----  
Process exited after 0.4994 seconds with return value 0  
Press any key to continue . . .
```

OVERLOADING BINARY OPERATOR USING A FRIEND FUNCTION

Another way of calling binary operator overloading with friend function is to call like a non member function as follows,

```
d3 =operator+ ( d1,d2 );
```

UNARY OPERATOR OVERLOADING

```
class UnaryFriend{
    int a=10;
    int b=20;
    public:
    void getvalues(){
        cout<<"Values of A and B\n";
        cout<<a<<"\n"<<b<<"\n"<<endl;}
    void friend operator-(UnaryFriend &x); };
void operator-(UnaryFriend &x){
    x.a = -x.a;    //Object name must be used as it is a friend function
    x.b = -x.b;}
```

UNARY OPERATOR OVERLOADING

```
int main(){  
    UnaryFriend x1;  
    cout<<"Before Overloading\n";  
    x1.getvalues();  
    cout<<"After Overloading \n";  
    -x1;// operator-(x1);  
    x1.getvalues();  
    return 0;}
```


<< OPERATOR

```
friend ostream & operator << (ostream &out, const Complex &c);
```

```
ostream & operator << (ostream &out, const Complex &c)
```

```
{
```

```
    // write code
```

```
}
```

>> OPERATOR

```
friend istream & operator >> (istream &in, Complex &c);
```

```
istream & operator >> (istream &in, Complex &c)
```

```
{
```

```
    // write code
```

```
}
```

OPERATOR OVERLOADING

```
#include <iostream>
using namespace std;
class Complex
{
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)
    {   real = r;   imag = i; }
    friend ostream & operator << (ostream &out, const Complex &c);
    friend istream & operator >> (istream &in, Complex &c);
};
```

OPERATOR OVERLOADING

```
ostream & operator << (ostream &out, const Complex &c)
{
    out << c.real;
    out << "+i" << c.imag << endl;
    return out;
}

istream & operator >> (istream &in, Complex &c)
{
    cout << "Enter Real Part ";
    in >> c.real;
    cout << "Enter Imaginary Part ";
    in >> c.imag;
    return in;
}
```

OPERATOR OVERLOADING

```
int main()
{
    Complex c1;
    cin >> c1;
    cout << "The complex object is ";
    cout << c1;
    return 0;
}
```

REFERENCES

<https://www.geeksforgeeks.org/overloading-stream-insertion-operators-c/>