

C++ friend Concept

Motivation

A friend Function

Another friend Function

Using friend Functions

Stream Operator Overloading

Stream Operators as friends

Using friend Operators

friend Classes

A friend of a class is another class or a function that is given special access privileges enabling it to access private and protected members.

The friend relationship is established by placing a statement in the declaration of the class that is granting the access privileges.

The friend relationship is neither symmetric nor transitive.

The use of the friend relationship is somewhat controversial since it violates the fundamental principle of information hiding.

Consider the `Location` class and the need to print `Location` objects.

```
class Location {
private:
    int Row;
    int Col;
public:
    Location();
    Location(int initRow = 0, int initCol = 0);
    bool operator==(const Location& Other);
    int  atRow() const;
    int  atCol() const;
    Location Up();
    Location Down();
    Location Left();
    Location Right();
};
```

This is a bit  
clumsy...

```
void DisplayLocation(Location P, ostream& Out) {

    Out << '(' << setw(2) << P.atRow() << ', '
        << setw(2) << P.atCol() << ')';

}
```

On the one hand, making the display function a member of `Location` would constrain the formatting of printed `Location` objects.

```
void DisplayLocation(Location P, ostream& Out) {  
    Out << '(' << setw(2) << P.atRow() << ',' <<  
        << setw(2) << P.atCol() << ')';  
}
```

On the other hand, this only works if `Location` provides public accessor functions for both the data members... sometimes that is unacceptable since it would allow ANY client of the class to view its private data.

Making the display function a friend of the `Location` class would allow a simpler implementation...

Enabling friend privileges:

```
class Location {
private:
    int Row;
    int Col;
public:
    Location();
    Location(int initRow = 0, int initCol = 0);
    bool operator==(const Location& Other);
    int  atRow() const;
    int  atCol() const;
    Location Up();
    Location Down();
    Location Left();
    Location Right();
    friend void DisplayLocation(Location P, ostream& Out);
};
```

No special syntax is used in the function implementation; the `friend` declaration is also not subject to the class access control specifiers.

The implementation of the friend function can now be simplified:

```
void DisplayLocation(Location P, ostream& Out) {  
    Out << '(' << setw(2) << P.Row << ', '  
        << setw(2) << P.Col << ')';  
}
```

The friend function can now directly access the private data members of the passed `Location` object.

A better case can be made if we want to read in a `Location` object from an input stream; `Location` provides no mutators for its data members, although the constructor could be used...

Here's a function to read in a Location:

```
void ReadLocation(Location& P, istream& In) {  
  
    In.ignore(255, '(');  
    In >> P.Row;  
    In.ignore(255, ',');  
    In >> P.Col;  
    In.ignore(255, ')');  
}
```

We also add a corresponding friend declaration to the Location class declaration.

Here's a short program that uses the friend functions:

```
void main() {  
  
    Location A(7, 2);  
    DisplayLocation(A, cout);  
    cout << endl;  
  
    Location B;  
    cout << "Enter location in format: (r, c)" << endl;  
    ReadLocation(B, cin);  
    DisplayLocation(B, cout);  
    cout << endl;  
}
```

The preceding functions are just alternatives to using the usual insertion and extraction operators for stream I/O.

It is possible to overload the usual operators and adopt a more natural syntax:

Takes an ostream as its first parameter.

Takes a Location as its second parameter.

```
ostream& operator <<(ostream& Out, const Location& Point) {  
  
    Out << '(' << setw(2) << Point.Row << ','  
        << setw(2) << Point.Col << ')';  
  
    return Out;  
}
```

Use it as:

```
Location A(-4, 3);  
cout << A << endl;
```

Returns the passed ostream object so that insertions can be "chained" as usual.

We can similarly overload the extraction operator for reading a `Location`:

```
istream& operator >>(istream& In, Location& Point) {  
  
    In.ignore(255, '(');  
    In >> Point.Row;  
    In.ignore(255, ',');  
    In >> Point.Col;  
    In.ignore(255, ')');  
  
    return In;  
}
```

Technical note: in order for an overloaded operator to be a class member, the **FIRST** parameter **MUST** be an object of that type.

This is not possible here because the first parameter must be the stream object.

So, stream operator overloads cannot be class members.

We can make the overloaded operators friends of `Location`:

```
class Location;
ostream& operator <<(ostream& Out, const Location& Point);
istream& operator >>(istream& In, Location& Point);

class Location {
private:
    // . . .
public:
    // . . .

    friend ostream& operator <<(ostream& Out,
                                const Location& Point);
    friend istream& operator >>(istream& In, Location& Point);
};
```

Note the forward declaration of `Location` and the operator declarations that precede the class declaration.

Typically the operator definitions are placed in `Location.cpp`.

Here's a short program that uses the friend operators:

```
#include "Location.h"

void main() {

    Location A(7, 2);
    cout << A << endl;

    Location B;
    cout << "Enter location in format: (r, c)" << endl;
    cin >> B;
    cout << B << endl;
}
```

In some cases, it may be desirable to allow all member functions of one class to have friend-level access to the members of another class.

```
class Location {
private:
    // . . .
public:
    // . . .

    friend class Maze;
};
```

It is tempting to overuse the friend relationship, as a lazy substitute for providing and using a suitable set of accessors and mutators.