Input and Output Operators

The library uses `>>` and `<<` for input and output, repectively. The IO library itself defines versions of these operators to read and write the built-in types. Classes that support IO ordinarily define versions of these operators for objects of the class type.

Overloading the Output Operator `<<`

Ordinarily, the first parameter of an output operator is a reference to a `nonconst ostream` object. The `ostream` is `nonconst` because writing to the stream changes its state. The parameter is a reference because we cannot copy an `ostream` object.

> Thus, the output operator must be defined as nonmember function. As for the overloaded operator function, the first parameter implicitly is `this` object.

The second parameter ordinarily should be a reference to `const` of the class type we want to print. The parameter is a reference to avoid copying the argument. It can be `const` because (ordinarily) printing an object does not change the object.

To be consistent with other output operators, `operator<<` normally returns its `ostream` parameter.

An example:

```
class Foo{
    private:
        int val;
        string str;
    public:

};


// IO operator overloading must be defined as nonmember function
ostream & operator<<(ostream &os,const Foo &f){
    os<<f.val<<std::endl;
    os<<f.str<<std::endl;
    return os;
}

// equivalent call to the output operator
std::cout<<f;
operator<<(std::cout,f);
```

Output Opeartors Usually Do Minimal Formatting

Generally, output operators should print the contents of the object, with minimal formatting. They should not print a newline.

IO Operators Must Be Nonmember Functions

Input and output operator that conform to the conventions of the `iostream` libarary must be ordinary nonmember functions. These operators cannot be members of our own class. If they were, the the left-hand operand would have to be an object of our class type:

```
Foo f;
f<<cout;// if operator << is a member of Foo
```

If these operators are members of any class, they would have to be memebrs of `istream` or `ostream`. However, those classes are part of the standard library, and we cannot add members to a class in the library.

Thus, if we want to define the IO operators for our types, we must define them as nonmember functions. Of course, IO operators usually need to read or write the `nonpublic` data members. As a consequence, IO operators usually must be declared as friends.

Overloading the Input Operator `>>`

Ordinarily, the first parameter of an input operator is a reference to the stream from which it is to read, and the second parameter is a reference to the (`nonconst`) object into which to read. The operator usually returns a reference to its given stream. The second parameter must be `nonconst` because the purpose of an input operator is to read data into this object.

An example:

```
class Foo{
    //
    private:
    int val;
    public:
        friend istream & operator>>(istream &,Foo &);
};
istream & operator>>(istream &is,Foo &f){
    is>>f.val;

    if(is)  //check that the inputs succeeded
    else // failure
        f=Foo();// input failed:give the object the default state
    return is;
}

// equivalent calls to the input operator
std::cin>>f;
operator>>(std::cin,f);
```

> Note: Input operators must deal with the possibility that the input might fail; output operators generally don't bother.

Error during Input

The kinds of errors that might happen in an input operator include the following:

- a read operation might fail because the stream contains data of an incorrect type. If data with compatiable type is input, that read and any subsequent use of the stream will fail.
- any of the reads could hit end-of-file or some other error on the input stream.

If there was an error, we do not worry about which input failed. Instead, we reset the entire object to the empty. It is important to leave the object in a valid state.

> Input operators should decide what, if anything, to do about error recovery.

Indicating Error

Some input operators need to do additional data verification. For example, our input operator might check that the data we read is in an appropriate format. In such cases, the input operator might need to set the stream's condition state to indicate failure, even though technically speaking the actual IO was successful.

- `eofbit` implys that the file was exhausted
- `badbit` indicate that the stream was corrupted