Lecture 18

- Global operators
- Friends
- Class operators

Global Operators

- How to define **global operators** such as input/output operators << >> ?
- Example: Input/Output declaration syntax:
  - ostream &operator<< (ostream &os, const X &x) { ... }
  - istream &operator>> (istream &is, X &x) { ... }
  - Reference to streams returned. Therefore **cout << x << y;** and **cin >> x >> y;** possible

Example

```
class Complex {     // Complex number class
   ...
private:
   float re, im;    // real and imaginary parts
};

// write complex number to output stream
ostream &operator<< (ostream &os, const Complex &x) {
   os << x.re << ' ' << x.im;
   return os;
}

// read complex number from input stream
istream &operator>> (istream &is, Complex &x) {
   is >> x.re >> x.im;
   return is;
}
// doesn't work:  re,im are private!
```


Solution: getters/setters or friends

```
class Complex {
public: ...
 friend ostream &operator<<(ostream &os, const Complex &x);
 friend istream &operator>>(istream &is, Complex &x);
private:
 float re, im;
};

ostream &operator<< (ostream &os, const Complex &x) {
   os << x.re << ' ' << x.im; return os;
}
// Alternative: << calls x.print(ostream &os);

istream &operator>> (istream &is, Complex &x) {
   is >> x.re >> x.im; return is;
}

// application
Complex a;
cin >> a; cout << a;
```

Friends

- Syntax (in class definition):
  - **friend** <function-declaration> **;**
  - **friend** <class-name> **;**
- Functions or entire classes now have access to all data/function members, even to those that are private!
- Avoid – usually indicates a broken design

Another Inheritance Application: (new) Generic output/input

- Would like to design functions that work on all derived objects
- Solution: pass a reference or pointer to a base class object to your function. Virtual functions in the derived class object can the be accessed.
- Example: C++ standard library output streams
  - **ostream** : output stream (base class)
  - **ofstream** : output file stream (public ostream)
    - Output is directed to a file
  - **ostringstream** – output string stream (public ostream)
    - Output is accumulated in string


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

struct Foo {
   int i, j;
   void write(ostream &os=cout) { os << i << " " << j; }
};

Foo x;

x.write();       // write to cout
x.write(cout);   // write to cout

ofstream of("file");
x.write(of);     // write to file

ostringstream oss;
x.write(oss);    // write to string stream

cout << oss.str() << endl;  // write string stream
                            // contents to cout
```


Class operators can be overloaded:

- Unary:
  `+ - * ! & ~ ++ --` (prefix/suffix)
- Binary:
  `+ - * / % ^ & | << >>`
  `= += -= *= /= %= ^= &= |= <<= >>= ==`
  `!= < > <= >=`
  `[] ()`
  `-> ->*`
  `new delete`
  `&& || ,`
- **DON'T OVERLOAD: prefix-& && || ,**

Int-Vector Revisited

```
class V {
public:
   V(int n_=1) { ... }
   ~V() { ... }

   int &operator[](int i) { check(i); return p[i]; }

   const int &operator[](int i) const {
      check(i); return p[i];
   }
   ...
private:
   void check(int i) const { assert(i >= 0 && i < n); }
   int *p, n; ...
};
```

```
#include "V.H"

V v(100);
v[3] = 0; cout << v[0];
```


Why Two Definitions?

```
class Foo {

public:

   V a;
   ...

   int bar() const {
      return a[0];  // only works if const definition
                    // is provided for V[]
                    // otherwise, the compiler complains
                    // that bar() may change a
   }
};
```


Member Function Syntax

- Unary prefix operator ++x (or --x):
  - `X& operator++() { ... }`
- Unary postfix operator x++ (or x--):
  - `X operator++(int) { ... }`
- Binary infix operator x @ x : (@ = + - * ...)
  - `Y operator@(const X &x) { ... }`
- []: `Y operator[](T i) {...}`
- (): `Y operator()(<params>) {...}`
- ->: `Y* operator->() {...}`
  has to return pointer because e.g. a->foo accesses member

Calling Class Operators

- Class operator applications are transformed into regular member function calls: E.g.
  - v[i] -> v.operator[](i)
  - a + b -> a.operator+(b)
  - ++x -> x.operator++()
  - x-- -> x.operator--(0)
- So, class operators are member functions
- They can be virtual!

```
// Complex Number class

class Complex {
public:

   Complex(float r=0, float i=0) : re(r), im(i) {}
   // default destructor, assignment, copy OK

   Complex operator+(const Complex &x) const;
   Complex operator+(float x) const;   // special case
   ...
   Complex &operator+=(const Complex &x);
   Complex &operator+=(float x);       // special case
   ...
   Complex &operator++();      // prefix ++
   Complex  operator++(int);   // postfix ++ !!!
   Complex  operator-() const; // unary operator
   ...
   float real() const { return re; }// gives environment
   float imag() const { return im; }// access to data

private:
   float re, im;  // real & imaginary part
};
```


Complex Class Implementation

```
#include "Complex.H"

Complex Complex::operator+(const Complex &x) const {
   // computes new coordinates, copy-constructs a new
   // object and returns it to the environment
   return Complex(re + x.real(), im + x.imag());
}

// faster implementation in special case
Complex Complex::operator+(float x) const {
   return Complex(re + x, im);
}

Complex &Complex::operator+=(const Complex &x) {
   re += x.real();
   im += x.imag();
   return *this;
}

Complex Complex::operator-() const {
   return Complex(-re, -im);
}
```