Operator Overloading

- Operators used on objects of different class types have different semantics (i.e. meaning)
- C++ allows most of the built-in operators to be overloaded (redefined) for user-defined classes
- The assignment (=), address of (&), and comma (,) operators have predefined meaning for operands of class type; all other operators must be defined explicitly for class type
- An operator may be overloaded for a class only if its conventional “meaning” is preserved in applying it to the class type objects

Operator Functions

- An operator is overloaded using an operator function type returntype operatorX(parameter list); where X is the operator symbol
- The use of an operator is a shorthand of an explicit call to the operator function complex

```cpp
Complex a(1.0,1.0); // use operator +
Complex b(0.5,0.5);
a.operator+(b) // explicit call to operator+
```

- Operator overloading cannot change
  - the precedence of the operators
  - the associativity of the operators
  - the “arity” of the operators

Complex number class revisited

Complex number class needs to provide functions for performing common arithmetic operations such as addition, subtraction etc., input/output, companions such as less-than etc.
Operator overloading provides a natural and convenient way to implement these operations

Example of complex number use

```cpp
Complex z1(1,2), z2(2,3);
cout << "z1==z2?: " << (z1==z2) << endl;
```

Friends

- In some cases it is convenient to allow certain nonmember functions and classes to access the private members of a class
- The friend mechanism allows a class to grant access to its nonpublic members
- A friend declaration begins with the keyword friend

```cpp
class C {
    friend return_type f(parameters);
    friend class C1;
    //...
};
```

- The function f and the class C1 given access to the private members of C
- Since friends are not member of the class granting friendship, they are not affected by the class access restrictions (public, private, etc.)

Operator Functions Implementation

Operators implemented as class members vs. nonmembers

- When an overloaded operator is class member, the class member is only invoked when the operator is used with a left operand that is an object of class type. If the operator must be used with a left operand of another type, then the overloaded operator must be a nonmember. e.g., stream extraction (<<) and insertion (>>) must be nonmembers.
- The assignment (=), subscript ([]), call (()), and member access arrow (->) operators are required by the language to be defined as class member operators.

Nonmember Input and Output operators for Complex class

The following declaration must appear outside the Date class definition

```cpp
istream operator>>(istream&, Complex&);
ostream operator<<(ostream&, const Complex&);
```

Operator Functions as Friends
class Complex
{
public:
    // Constructor
    Complex(double re=0.0, double im=0.0);
    // basic arithmetic functions
    friend Complex operator+(const Complex&, const Complex&);
    friend Complex operator-(const Complex&, const Complex&);
    friend Complex operator*(const Complex&, const Complex&);
    friend Complex operator/(const Complex&, const Complex&);
    // unary minus - change sign
    // this is implemented as a member function
    Complex operator-() const;

private:
    double re_;  // real part
    double im_;  // imaginary part
};

52 Operator Functions as Friends (2)
// boolean functions
friend bool operator==(const Complex&, const Complex&);
friend bool operator!=(const Complex&, const Complex&);

// I/O functions
friend istream& operator>>(istream& is, Complex& a);
friend ostream& operator<<(ostream& os, const Complex& a);

53 Implementation of Arithmetic Operators
// binary operators
Complex operator+(const Complex& a, const Complex& b)
{
    return Complex(a.re_ + b.re_, a.im_ + b.im_);
}

Complex operator-(const Complex& a, const Complex& b)
{
    return Complex(a.re_- b.re_, a.im_- b.im_);
}

54 Implementation of >> and <<
istream& operator>>(istream& is, Complex&a)
{
    char c1, c2, c3; // for ( , )
    double r, i;
    is >> c1 >> r >> c2 >> i >> c3;
    if (is)
        a = Complex(r,i);
    return is;
}

ostream& operator<<(ostream& os, const Complex& a)
{
    return (os << "(" << a.re_ << " , " << a.im_ << ")" );
}

55 Implementation of Relational Operators
bool operator==(const Complex& a, const Complex& b)
return (a.re_ == b.re_ && a.im_ == b.im_);
}

bool operator!=(const Complex &a, const Complex &b)
{
    return (a.re_ != b.re_ || a.im_ != b.im_);
}

// Unary minus as member function
Complex Complex::operator-() const
{
    return Complex(-re_, -im_);
}

56  Testing Overloaded Operator Functions
Complex z1, z2;

cout << "A complex number please, in the form (real, imag): ";
cin >> z1;
cout << endl;
cout << "Another one, please: ";
cin >> z2;
cout << endl;
cout << "I read " << z1 << " * " << z2 << endl;
cout << "z1==z2? : " << (z1==z2) << endl;
cout << "z1!=z2? : " << (z1!=z2) << endl;
cout << "z1 + z2 : " << (z1+z2) << endl;

57  Agenda
• The complete code for Complex number class is available from the class webpage
• Operator overloading is described on pages of the Savitch text.
• Wednesday and Friday lectures will be on pointers and dynamic memory
Reading: