UEE1303(1070) S'12 Object-Oriented Programming in C++

Lecture 05:

Understanding Friends and Overloading Operators

Learning Objectives

- Basic operator overloading
  - unary and binary operators
  - as member functions
- Friends and automatic type conversion
  - friend functions and friend classes
  - constructors for automatic type conversion
- References and more overloading
  - input insertion << and output extraction >>
  - operators: ++, --, []

Overloading for Integral Datatypes

- Operators are overloaded in C/C++:
  - `+7, 2+5, 3.25+7.3`
- In addition to overloading, compilers often need to perform coercion or casting when the + symbol is used with mixed arithmetic
- To use arithmetic symbols with our own objects ⇨ must overload the symbols
  - polymorphism allows the same operations to be carried out differently
  - overload the + operator with a reasonable meaning
  - Ex: `ptA + ptB //ptA∈CPoint, ptB∈CPoint`

A Starting Example (1/2)

lec5-1.cpp

```
class CComplex {
    double real, imag;
public:
    CComplex() { real=0; imag=0; }
    CComplex(double r, double i) {
        real=r; imag=i; }
    CComplex cadd(CComplex & o2);
    void display() { cout << "(" << real
        << "," << imag << "i)" << endl };
};
CComplex CComplex::cadd(CComplex & o2) {
    CComplex c; c.real=real+o2.real;
    c.imag=imag+02.imag; return c;
}
```

A Starting Example (2/2)

```
int main() {
    CComplex c1(3,4), c2(2,-7), c3;
    c3 = c1.cadd(c2);
    cout << "c1 = "; c1.display();
    cout << "c2 = "; c2.display();
    cout << "c1+c2 = "; c3.display();
    return 0;
}
```

lec5-1.cpp

```
c1 = (3,4i)
c2 = (2,-7i)
c1+c2 = (5,-3i)
```

- Using member function is cumbersome
  - good to have `c3=c1+c2`

More Operator Overloading

- Operators `+`, `-`, `%`, ==, etc
  - ⇨ really just functions!
- Simply called with different syntax: `x+7`
  - `+` is binary operator with `x` and `7` as its operands ⇨ like this notation as humans
- Overload an operator by making it a function
  - `+(x,7)` ⇨ `+`: function name, `x` & `7`: arguments
  - function `+` returns sum of all its arguments
- If an operator is normally defined to be unary only, then you cannot overload it to be binary
  - cannot change associativity or precedence

Operator Overloading Perspective

- Built-in operators
  - e.g., +, -, =, %, ==, /, *
  - already work for C++ built-in types
  - in standard *binary* notation
- Overload these basic operators
  - to work with our own datatypes!
  - to add "Chair types", or "Money types"
    - ⇨ as appropriate for our needs
  - in notation that we are comfortable with
- Always overload with similar *actions*!

Overloading Basics (1/2)

- Overloading operators
  - ***very*** similar to overloading functions
  - operator itself is the *name of function*
- Example:

```
const CMoney operator+(const CMoney &obj);
```

  - overload `+` for operands of type `CMoney`
  - use constant reference parameters for efficiency
  - the returned value is type `CMoney`
    - ⇨ allow addition of `CMoney` objects

Overloading Basics (2/2)

- Overloading operators can be classified into
  - (1) overloading member functions

```
⟨datatype⟩ ⟨class_name⟩::operator⟨operator⟩
(⟨parameter_list⟩) { //functional body; }
```

  - typically, for binary operators
    Ex: assignment/subscript([ ])/function(( ))
  - (2) overloading friend functions

```
⟨datatype⟩ operator⟨operator⟩
(⟨parameter_list⟩) { //functional body; }
```

  - typically, for unary operators
    Ex: input insertion(<<) & output extraction(>>)

Review of Friends (1/2)

- Only *member functions* can access the `private` data of one class
- You may want to allow a *nonmember function* to have access to a `private` data
  - ⇨ a `friend` function is a nonmember function that can access the `non-public` members of a class
  - –should be used only when absolute necessary
  - –avoid them simply to overcome encapsulation

Review of Friends (2/2)

- A nonmember function can be declared in the `public` or `private` section or first in the class
- Not required to use the word `friend` within in the function name of a `friend` function
- Overloaded functions can be `friends`
  - –but each must be explicitly designated as a `friend` function
  - –limit your use of `friend` functions
  - –necessary when you overload input and output operators for a class

Example of `friends` (1/2)

```
class CCustomer {
    friend void showAFriend(CCustomer);
    int cid; double balance;
public:
    CCustomer(int x=0, double y=0) {
        cid=x; balance=y; }
    void showCCustomer() {
        cout << cid << " with $"
             << balance << endl;    }
};
void showAFriend(CCustomer c){
     cout << c.cid << " with $"
          << c.balance << endl;
}
```

lec5-2.cpp

Example of friends (2/2)

```
int main() {
    CCustomer one( 10963, 3437.95);

    //call member function
    one.showCCustomer();

    //call friend function
    showAFriend(one);

    return 0;
}
```

lec5-2.cpp

List of Overloading Operators (1/2)

- Arithmetic
  - +, -, *, /, %
- Bitwise
  - ^, &, |, ~, >>, <<
- Correlational
  - <, <=, >, >=, !=, ==
- Logic
  - !, &&, ||
- Assignment
  - =, +=, -=, *=, /=, %=, <<=, >>=, &=, ^=,|=
- Other
  - ++, --, [], (), ->, new, new [], delete, delete [], ,

List of Overloading Operators (2/2)

- Five operators **cannot** be overloaded
- You also cannot overload operators that *you invent*, ex: cannot redefine o1@o2
- Operators cannot be overloaded for built-in datatypes, ex: cannot redefine 5+3

operator	usual use
.(dot operator)	member
*	pointer to member
::	scope resolution
?:	conditional
sizeof	size of

Overloading Operator +

- Given previous example:
  - overloaded "+" NOT member function
  - definition is *more involved* than a simple add
  - require issues of money-type addition
  - must handle negative/positive values
- Operator overload definitions generally very simple
  - just perform addition particular to the *user-defined* type

Overloading + on CComplex

lec5-3.cpp

```
class CComplex {
    double real, imag;
public:
    CComplex() { real=0; imag=0; }
    CComplex(double r, double i) {
        real=r; imag=i; }
    friend CComplex operator+(
        CComplex& o1, Complex& o2 );
    void display() { //... }
};
CComplex operator+( CComplex& o1,
                    CComplex& o2 ){
    return CComplex(o1.real+o2.real,
                    o1.imag+o2.imag);
}
```


Overloading + on CComplex

lec5-3.cpp

```
class CComplex {
    double real, imag;
public:
    CComplex() { real=0; imag=0; }
    CComplex(double r, double i) {
```

What declare the overloading operator function into a friend function ?

```
    void display() { //... }
};
CComplex operator+( CComplex& o1,
                    CComplex& o2 ){
    return CComplex(o1.real+o2.real,
                    o1.imag+o2.imag);
}
```


Member vs. Friend Functions

- Implement the overloading operator as a member function
  - use `this` to visit the data member
  - the *left* operand must be a object of same class, Ex: **`c1`**`+c2`
- What if `c3=c1+x`?

```
CComplex CComplex::operator+(int& x){
    return CComplex(real+x,imag); }
```

- What if `c3=x+c2`? ⇨ use a *friend* function

```
CComplex operator+(int& x, CComplex& o){
    return CComplex(x+o.real,o.imag); }
```


Overloading Operator ==

- Equality operator ==
  - enable comparison of objects of one class
  - return `bool` type for true/false equality
  - again, it is a non-member function, like overloading `+`
- Overloading `==` on `CComplex`

```
friend bool operator==(const CComplex &,
                       const CComplex &);
```

```
bool operator==( const CComplex& o1,
                 const CComplex& o2 ){
    return ((o1.real==o2.real)&&
            (o1.imag==o2.imag)); }
```

Constructors Return Objects

- Is constructor a `void` function?
  - –we think that way, but the truth is no
  - –actually, it is a special function with special properties and can return a value!
- Recall `operator+` from

```
CComplex CComplex::operator+(int& x){
    return CComplex(real+x,imag);
}
```

  - –return an invocation of class `CComplex`
  - –So constructor actually returns an object named anonymous(nameless) object

Return by non-const Value (1/2)

- Assume a mutator `input()` and non-`const` overloading `+` in class `CComplex` :

```
void CComplex::input(){
    cout << "input real="; cin >> real;
    cout << "input imag="; cin >> imag;
}
CComplex CComplex::operator+(
         CComplex& o2 ){
    return CComplex(real+o2.real,
                    imag+o2.imag);
}
```

lec5-3.cpp

  - –return a `non-const` object
  - –allow modification of anonymous object

Return by non-const Value (2/2)

- Consider non-`const` in declaration:

```
CComplex CComplex::operator+(
         CComplex& o2 ){
    return CComplex(real+o2.real,
                    imag+o2.imag);
}
```

- Given two `CComplex` object `c1` and `c2`
  - –object returned is a `CComplex` object
  - –can do something with the returned object
  - –Like calling a member function
  - –Ex: `(t1+t2).input()` Why?

Return by const Value

- So define the returned object as `const`

```
const CComplex CComplex::operator+(
                CComplex& o2 ){
    return CComplex(real+o2.real,
                    imag+o2.imag);
}
```

- What if ??

```
CComplex t1(3,4), t2(2,-5), t3;
t3 = (t1 + t2); //assignment does??
t3.input(); //legal??
```

- Is `(t1+t2).input()` a legal call??
- `t3` and `(t1+t2)` are different objects

Overloading Unary Operators

- C++ has unary operators:
  - defined as taking one operand
  - e.g. `x=-y; //set x to negated y`
  - other unary operators, ex: `++`, `--`
  - unary operators can also be overloaded
- Overloading operator `-` on `CComplex`

```
const CComplex CComplex::operator-() {
    return CComplex(-real,-imag);
} //what if a friend func?
```

lec5-4.cpp

  - need no argument
  - overload twice: one for binary (minus) and one for unary (negation)

Example of Unary Operators

```
int main() {
    CComplex c1(3,4), c2(2,-7), c3;
    c3 = c1 - c2; //call binary operator-
    cout << "c3 = "; c3.display();
    c3 = -c2;     //call unary operator-
    cout << "c3 = "; c3.display();
    return 0;
}
```

lec5-4.cpp

- Output

```
c3 = (1,11i)
c3 = (-2,7i)
```


Overloading as Member Functions

- Previous examples: standalone functions
  - defined outside a class
  - use `friend`
- Can overload as a member operator
  - implement a member function like others
- When operator is a member function:
  - only ***one*** parameter, not two!
  - calling object serves as the first parameter

Member Operator in Action

- Example

```
CComplex c1(3,4), c2(2,-7), c3;
c3 = c1 + c2; //call binary operator+
```

  - if "+" overloaded as a member operator,
    - ⇨ variable/object cost is calling object
    - ⇨ object `c2` is single argument
  - think of as: `c3 = c1.`**`opertor+`**`(c2);`
  - declaration of "**`opertor+`**" in class definition

```
const CComplex CComplex::operator+(
                CComplex& o2) { //... }
```

  - notice only one argument

Example of +/-/+= on CComplex (1/3)

lec5-5.cpp

```
class CComplex {
    double real, imag;
public:
    CComplex() { real=0; imag=0; }
    CComplex(double r, double i) {
        real=r; imag=i; }
    void display() { cout << "(" << real
        << "," << imag << "i)" << endl };
    CComplex operator+(CComplex& o2);
    CComplex operator+(double r);
    void operator+=(CComplex& o2);
    friend CComplex operator+(
                    double r, CComplex& o1);
    friend CComplex operator-(CComplex& o1);
};
```


Example of +/-/+= on CComplex (2/3)

lec5-5.cpp

```
//overloading as member functions
CComplex CComplex::operator+(CComplex& o2) {
    CComplex t; t.real=real+o2.real;
    t.imag=imag+o2.imag; return t; }
CComplex CComplex::operator+(double r) {
    CComplex t; t.real=real+r;
    t.imag=imag; return t; }
void CComplex::operator+=(CComplex& o2) {
    real+=o2.real; imag+=o2.imag; }
//overloading as friend functions
CComplex operator+(double r, CComplex& o1) {
    CComplex t; t.real=r+o1.real;
    t.imag=o1.imag; return t; }
CComplex operator-(CComplex& o1) {
    return CComplex(-o1.real,-o1.imag); }
```


Example of +/-/+= on CComplex (3/3)

```
int main() {
    CComplex c1(12,-20), c2(-5, 9), c3;
    c3=c1+c2; c3.display();
    c3=c1+10; c3.display();
    c3=-8+c2; c3.display();
    c2+=c1;   c2.display();
    c1=-c3;   c1.display();
    return 0;
}
```

lec5-5.cpp

```
(7,-11i)
(22,-20i)
(-13,9i)
(7,-11i)
(13,-9i)
```


Overloading Operator ++/--

- ++/-- are unary operators
  - prefix operation: ++obj, --obj
  - postfix operation: obj++, obj--
- Declare as ***member functions***

```
⟨CNAME⟩& ⟨CNAME⟩::operator++(); //prefix
⟨CNAME⟩ ⟨CNAME⟩::operator++(int); //postfix
```

- Declare as ***friend functions***

```
//prefix friend function
friend ⟨CNAME⟩& ⟨CNAME⟩::operator++(⟨CNAME⟩&);
//postfix friend function
friend ⟨CNAME⟩ ⟨CNAME⟩::operator++(
                                ⟨CNAME⟩&, int);
```

Example of Overloading ++ (1/2)

lec5-6.cpp

```
class CCount {
    unsigned int cnt;
public:
    CCount(int n=0) { cnt=n; }
    void display() { cout << cnt; }
    //prefix increment as member
    CCount& operator++();
    //postfix increment as friend
    friend CCount operator++(CCount&, int);
};
CCount& CCount::operator++() {
    cnt++; return *this;                }
CCount operator++(CCount& x, int y) {
    CCount tmp=x; x.cnt++; return tmp;  }
```


Example of Overloading ++ (2/2)

lec5-6.cpp

```
//int main()
    CCount d1(10), d2;
    d2=d1++; //call postfix increment
    d1.display();d2.display();cout << endl;
    d2=++d1; //call prefix increment
    d1.display();d2.display();cout << endl;
    ++++d1;
    d1.display();d2.display();cout << endl;
```

```
11 10
12 12
14 12
```


Overloading >> and <<

- Enable input and output of our objects
  - similar to other operator overloads
  - new subtleties
- Format of overloading operator >> and <<

```
istream& operator>>(istream&,〈CNAME〉&);
ostream& operator<<(ostream&,〈CNAME〉&);
```

```
class CComplex {
    friend ostream& operator<<(
              ostream& out, CComplex& c) {
        out << "(" << c.real << ","
            << c.imag << "i)" << endl;
        return out;                     }
};
```


Overloading >> and <<

- Enable input and output of our objects
  - similar to other operator overloads
  - new subtleties
- Format of overloading operator >> and <<

```
CComplex c1(2,5), c2(-3,-2), c3;
c3=c1+c2; cout<<c3;//what if cout<<c3<<c2;
```

```
class CComplex {
    friend ostream& operator<<(
              ostream& out, CComplex& c) {
        out << "(" << c.real << ","
            << c.imag << "i)" << endl;
        return out;                     }
};
```

Example of Overloading >>

- Overloading operator >> on CComplex

```
class CComplex {
    friend istream& operator>>(
             istream& inp, CComplex& c) {
        inp >> c.real >> c.imag;
        return inp;
    }
};
```

```
//in main(){}
CComplex c1, c2, c3;
cin >> c1 >> c2;
c3 = c1 + c2;
cout << c1 << c2 << c3;
```


Overload Array Operator []

- Can overload [] for your class
  - typically, X[i]⇔*(X+i)
  - used to check ***out-of-bound***
  - a binary operator: the left operand is a reference object + the right one is an integer
- Format
  - operator **must** return a reference
  - operator [] **must** be a member function

```
⟨cname⟩& ⟨cname⟩::operator[](int i)
{ //functional body; }
```


Example of Overloading [] (1/2)

- Recall CStr() example in Lecture 03

lec5-7.cpp

```
class CStr
{
private:
    char * line;
public:
    CStr(char* word);
    CStr(const CStr & old);
    ...

    //overloading [] as a member function
    char & operator[](int i);
};
```


Example of Overloading [] (2/2)

```
char& CStr::operator[](int i) {
        if (i>=strlen(line)) {
            cerr << "Error: " << i <<
            " is out of bound!!!" << endl;
        }
        return line[i];
};
```

lec5-7.cpp

```
int main() {
    CStr one("lec5-7"); //call constructor
    cout << one[5] << endl; //what happens?
    cout << one[8] << endl; //what happens?
    return 0;
}
```

Type Casting for Class (1/2)

- C++ provides *explicit* type conversion
  - ⟨datatype⟩(⟨data⟩), ex: `int(82.7)`
  - (⟨datatype⟩)⟨data⟩, ex: `(double)49`
- Conversion constructor casts the data of one type into an object of another class, ex:
  - can have only one parameter ⇨ why?

```
class CComplex {
    CComplex(double r) {
        real = r; imag = 0;                }
};
```

```
CComplex o1(4.2);
CComplex o2 = o1 + CComplex(2.5);
```


Type Casting for Class (2/2)

- What if converting a `CComplex` into a `double`?
  - need a *type-conversion* function
  - format:

```
⟨cname⟩::operator⟨datatype⟩ ()
    { //functional body; }
```

- Example

```
class CComplex {
    operator double() { return real; }
};
```

  - cannot assign the returned datatype
  - cannot have any parameter

```
CComplex o1(4.2); double d2 = 12;
double d3 = d2 + o1; //double operator +
```


Summary (1/2)

- C++ built-in operators can be overloaded
  - to work with objects of your class
- Operators are really just functions
- `friend` functions have direct private member access
- Operators can be overloaded as member functions where
  - the first operand is the calling object

Summary (2/2)

- `friend` functions add efficiency only
  - not required if sufficient accessors and mutators are available
- Reference "names" a variable with an alias
- References and more overloading
  - operators: = , [], ++, --
- Can overload << and >>
  - return type is a reference to stream type