LAB 6

CONSTRUCTORS, OPERATOR OVERLOADING & FRIEND

UEE 1303
Department of Electrical and Computer Engineering, Institute of Electronics
National Chiao Tung University

Outline

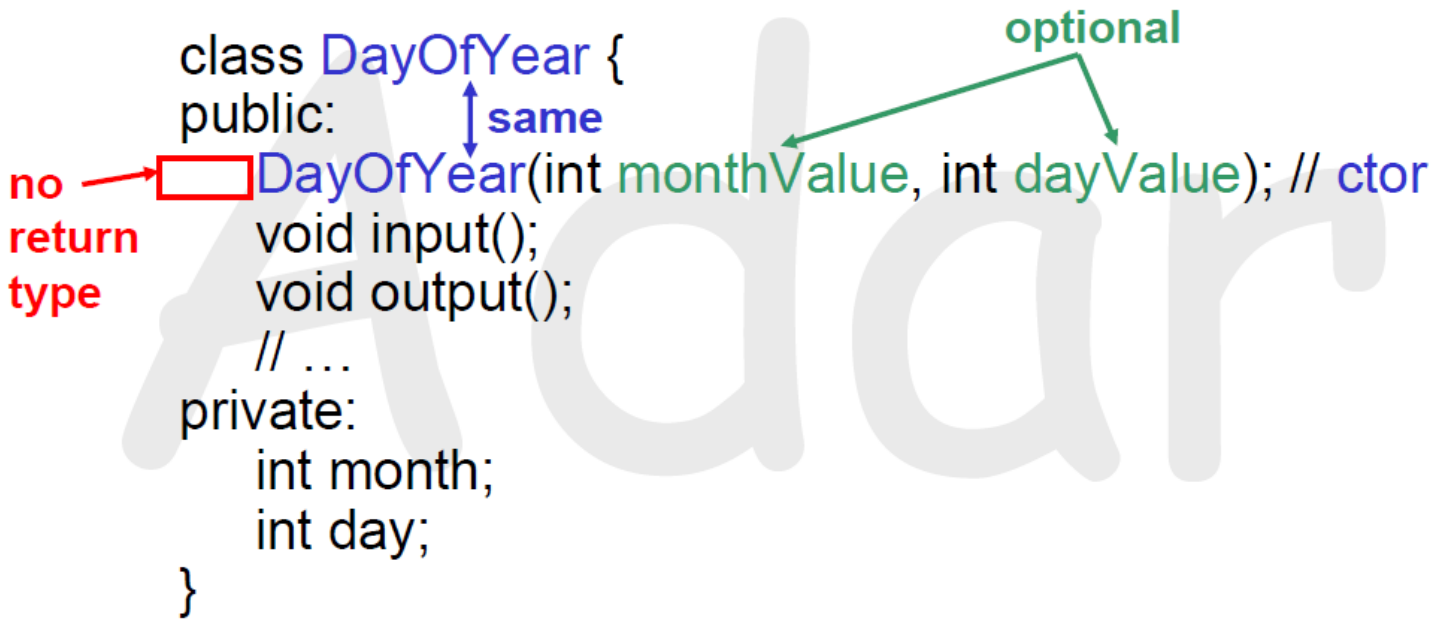
- Constructor
  - Initialization list
  - Copy Constructor
- Operator overloading
- Friend function and friend class

Constructors (ctors)

- ctor is dedicated to
  - initialization of some or all data members
  - other necessary actions during initialization
- ctor is a special kind of member function
  - **automatically called** when an object is **born**
- ctor is declared just like any other member **functions** except
  - ctor name **MUST** be **SAME as class name**
  - ctor has **NO return type**; not even void
- Yes, there are destructors(dtors) too(Chapter 10)

Constructor Declaration

- Class definition with ctor declaration



Initialization List

```
DayOfYear::DayOfYear(int monthValue, int dayValue):
month(monthValue), day(dayValue)
{ // can be empty if nothing to do here }
```

- Why use initialization list?
  - For Initialization of Non-Static const Data Members
  - For Initialization of Reference Members
  - For Initialization of Member Objects that do not have a Default Constructor

Initialization List

```
class A {
    int i;
public:
    A(int x): i(x) {};
};

class B {
    A a;
public:
    B(int x);
};
```

```
// Initializer list must be used
B::B(int x): a(x)
{
        cout << "B's Constructor called";
}

int main()
{
    B obj(10);
    return 0;
}
```

Otherwise, “error: no matching function for call to ‘A::A()’ ” since A does not have a default constructor.

Copy Constructor

- If you do not define the copy constructor
  - Compiler does it
  - **Member-wise** initialization
- When do we need to write a copy constructor?
  - Member data contains pointers

```
class person
{
public:
    person();
    person(int x) { age = new int{x}; };
    int *age;
};

int main(void)
{
    person p1{5};
    cout << *p1.age << endl;      // 5

    person p2{p1};
    cout << *p2.age << endl;      // 5

    *p2.age = 10;
    cout << *p1.age << endl;      // 10!! They point to the same thing!!
    cout << *p2.age << endl;      // 10
```

Why Operator Overloading?

- Define a member function operator+

```
class complex {
        double re, im;
public:
        complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }
        const complex operator+(const complex&) const;
};
const complex complex::operator+(const complex& rhs) const {
        complex result(rhs); // using copy ctor, too
        result.re += re; result.im += im;
        return result;
 }
void f() {
        complex a(1, 1), b(2, 2), c;
        c = a.operator+(b); // ok! explicit call, just ugly!
        c = a + b; // ok! it is just a shorthand for operator+
}
```

Another Way for Operator Overloading

- Overloaded operators are NOT necessarily member functions!

```
class complex {
        double re, im;
public:
        complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }
        double real() const { return re; }
        double image() const { return im; }
};
const complex operator+(const complex& lhs, const complex& rhs) {
        double real, image;
        real = lhs.real() + rhs.real(); image = lhs.image() + rhs.image();
        return complex(real, image);
}
void f() {
        complex a(1, 1), b(2, 2), c;
        c = operator+(a, b); // ok! explicit call, just ugly!
        c = a + b; // ok! it is just a shorthand for operator+
}
```

Member vs. Nonmember Operators

- If mixed-mode arithmetic is allowed e.g., allow adding a complex with a double

```
void f() { // operator+ is a member function here
        complex a(1,1), b;
        b = a + 1.0; // ok! a.operator+( complex(1.0) )
        b = 1.0 + a; // error! 1.0.operator+(a) <= no such function!
}
void f() { // operator+ is a nonmember function here
        complex a(1,1), b;
        b = a + 1.0; // ok! operator+( a, complex(1.0) )
        b = 1.0 + a; // ok! operator+( complex(1.0), a )
}
```

- In general, nonmember version is preferred

Friend Functions

- Nonmember functions access private data via functions

  - inefficient due to function calls

- Friend functions can directly access private and protected members of the class.

  - no calls to accessors and mutators => more efficient

Friend Functions – Usage (1/2)

- A function is declared as a friend of a class using the keyword “friend” inside the class.
- The keyword “friend” doesn’t need to be placed at the function definition.
- The function is called like an ordinary function.

- A function can be a friend of multiple classes.
- A friend declaration can be placed in any section (public, private, or protected).

Friend Functions – Usage (2/2)

- A friend function can be
  - A global function (operator = a kind of function)
  - A member function of another class
- Most common use: nonmember operators
  - Mixed-type arithmetic
  - Efficiency

Friend Functions – Example (1/2)

const complex operator+(const complex&, const complex&);

```
class complex {
    double re, im;
public:
    complex(double r = 0.0, double i = 0.0) : re(r), im(i) { }
    double real() const { return re; }
    double image() const { return im; }
    friend const complex operator+(const complex&, const complex&);
};
```

Friend Functions – Example (2/2)

```
// operator+ is a friend function of class complex
// no need to add friend prefix in function definition
const complex operator+(const complex& lhs, const complex& rhs) {
    complex result(lhs);
    result.re += rhs.re; result.im += rhs.im;
    return result;
} // a friend function has same access privilege as member functions

// operator- is not a friend function of class complex
const complex operator-(const complex& lhs, const complex& rhs)
    double real = lhs.real() + rhs.real();
    double image = lhs.image() + rhs.image();
    return complex(real, image);
} // need accessors to get private data
```

Friend Class

- Make class X be a friend class of class Y

```
class Y {
    friend class X; // s.t. all member functions of X are friend functions of Y.
    …
};
```

  - X can access the private/protected data of Y
  - Y cannot access the private/protected data of X

Recall: Exercise in the Last Lab

- Create a class Science and provide the following functions
- 2 private data members : double and int type
  - a*10^n , where $1 \le |a| < 10$ or $a = 0$, n is an integer
- Finish constructor, operator+,-,*,/,>>,<<
- operator<< and operator>> for output/input science
  - The output format : a*10^n
    - Always in reduced form
  - The input format : a n
    - input can be in non-reduced form , ex: a=12.34, n=1
    - n will always be an integer

Exercise (1/4)

- Create a class Complex.
- 2 private data members:
  - Science re, im;
- Finish the constructor.
- Finish the operators +,- (unitary), - (binary), *
- Finish operator<< and operator>> for output/input science
  - The output format: {re} + {im} i
  - The input format: re.a re.n im.a im.n

Exercise (2/4)

- Create a class ComplexPolynomial, which supports the computation of complex numbers.
- 3 private data members:
  - Complex a, b, c; // represent the coefficients of a quadratic polynomial
- Finish the constructor.
- Finish the member function **eval**.
  - Complex eval(const Complex &x)
  - Return the answer of ax^2+bx+c

Exercise (3/4)

- A template file and a testing data file is given.
- Please don't touch the provided main function
  - Just finish constructor, operator overloading, and eval().

- Your class should be able to handle operations of large numbers
  - Ex: 1.23*10^1000 / 2.7*10^800 = 4.55556*10^199
- TA will not input an expression with too large difference between two operands
  - Ex: 1*10^1000 + 1*10^10

Exercise (4/4)

- Result:

```
[M112zkxu@adar10 OOP]$ ./a.out
Enter coefficient a (4 numbers: re.a re.n im.a im.n)
2.5 2 6.2 2
Enter coefficient b (4 numbers: re.a re.n im.a im.n)
-12.5 1 3.6 2
Enter coefficient c (4 numbers: re.a re.n im.a im.n)
1.7 2 -0.45 3
Enter x (4 numbers: re.a re.n im.a im.n)
3.2 2 1.3 1
The evaluation answer is 2.03548*10^7 + 6.55763*10^7 i
```

- You can use “./a.out < in.dat” to read the input file.