LAB 5

CONSTRUCTORS AND OTHER TOOLS
&
OPERATOR OVERLOADING, FRIENDS, REFERENCE

Outline

- Constructors

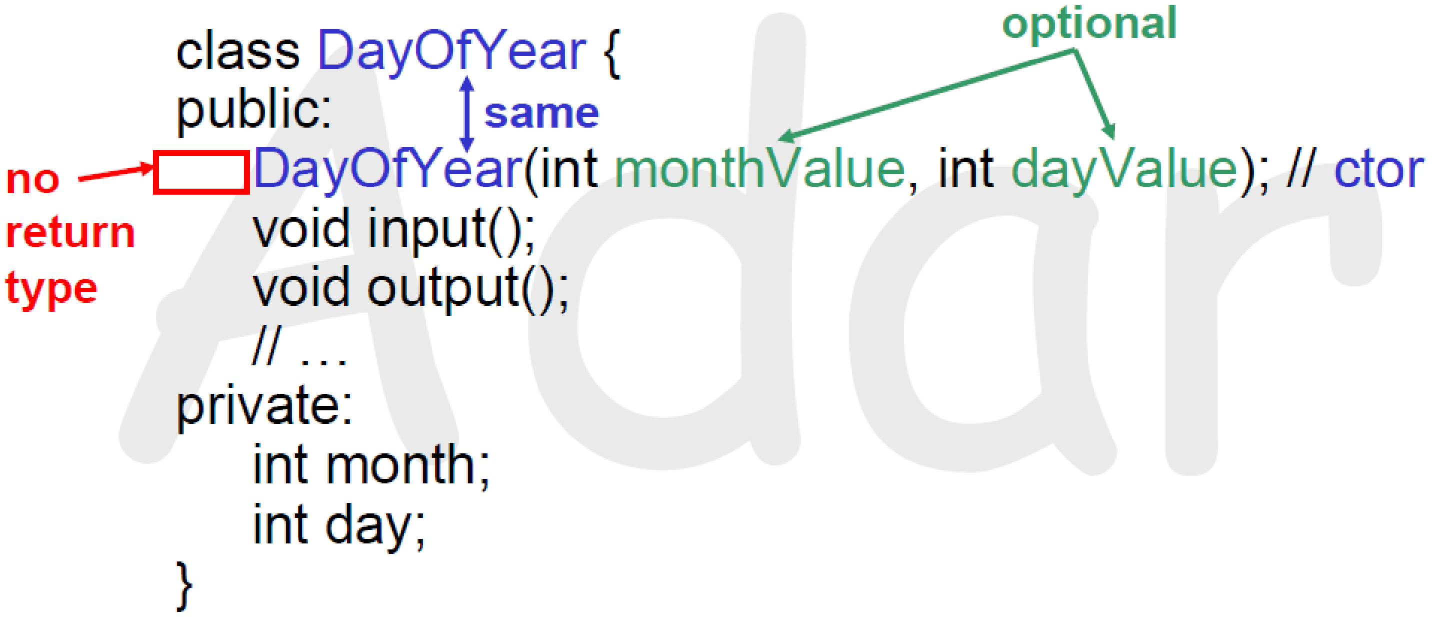
- Operator Overloading

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



```
class DayOfYear {
public:
    DayOfYear(int monthValue, int dayValue); // ctor
    void input();
    void output();
    // …
private:
    int month;
    int day;
}
```

optional

same

no return type

Calling Constructors (1/2)

-
  ```
  void f(){
          DayOfYear date1(7, 4), date2(3, 6);
          // implicit ctor call for date1 with (7, 4)
          // implicit ctor call for date2 with (3, 6)
  }
  ```
- As soon as data1/date2 is born (created)

  - ctor is automatically called for each
  - Values in parentheses passed as arguments to the ctor
  - Data members (month & day) are then initialized by the ctor

Calling Constructors (2/2)

-
  ```
  void f() {
          DayOfYear date1;  // Error
          date1.DayOfYear(7, 4);  // Error
          DayOfYear date2(3, 6);  // ok
          // …
  }
  ```
- Error

  – If there is any existing ctor, compiler will NOT generate a default ctor for you
- Error

  – Object is NOT allowed to call ctors directly

Constructor Definition

- ctor definition is like all other member functions

  – except it has no return type

-
  ```
  DayOfYear::DayOfYear(int monthValue, int dayValue) {
          month = monthValue;
          day = dayValue;
  }
  ```

- Note that same name around ::

  – Clearly identifies a ctor

- Note that no return type

  – Just as in class definition

Complete Constructor Definition

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

- Prefer this style for ctor definition

Overloaded Constructors

- Ctor can be overloaded just like other functions

  - that is, a class can have multiple ctors

```
class DayOfYear{
public:
        DayOfYear(int monthValue, int dayValue);  // ctor#1
        DayOfYear(int monthValue)  // ctor#2
        DayOfYear( )  // ctor#3
        (default constructor)
        // …
}
void f() { DateOfYear d1(7, 4), d2(9), d3; … }
```

Explicit Constructor Calls

- Explicit Constructor Calls
- A temporary object will be created by explicitly calling a ctor

  – that object has no name

  – it is destroyed at the end of expression

```
void f() {
        DayOfYear holiday(7, 4);
        holiday = DayOfYear(5, 5);
        // 1. Explicitly call a ctor
        // 2. Create a temp object w/o name and initialized by that ctor call
        // 3. The temp object is assigned to holiday
        // 4. After finishing assignment, the temp object is destroyed
}
```

Constant Member Functions (1/2)

- If a member function does not make any modifications on data members - ALWAYS make it a constant member function

```
class DayOfYear {
public:
        DayOfYear(int, int);
        DayOfYear(int);
        DayOfYear( );
        void output( ) const;
        int getMonthNumber( ) const;
        int getDay( ) const;
 private:
         int month;
         int month;
         int month;
         int day;
        void testDate( ) const;
 };
```

```
class Holiday {
public:
            Holiday( );
            Holiday(int, int, bool);
            void output( ) const;
private:
            DayOfYear date;
            bool parkingEnforcement;
};
```

Constant Member Functions (2/2)

```
void Holiday::output( ) const
{
    date.output( ); cout << endl;
    if (parkingEnforcement)
        cout << "Parking laws will be enforced.\n";
    else
        cout << "Parking laws will not be enforced.\n";
}
```

const modifier must be presented in both function declaration & definition

```
void f() { // if Holiday::output() is NOT a const member function
  Holiday dragon_boat(6, 6, true); const Holiday new_year(1, 1, true);
  dragon_boat.output();  // ok
  new_year.output();       // compilation error!
}
```

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+
}
```

Returning Constant Value

- const complex operator+(const complex& lhs , const complex& rhs)
- complex operator+(const complex& lhs , const complex& rhs)

```
void f() {
        complex a(1,1), b(2,2), c(3,3);
        (a + b) = c; // no error if using red one; error if using blue one
        if((a+b) = c) // Oops, programmer actually wants => if((a+b) ==c)
        do_things // again, no error if using red one; error if using blue one
}
```

- Hence, blue one is preferred

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 (1/3)

- Nonmember functions

  - access private members through accessors and mutators

  - inefficient (overhead of calls to accessors and mutators)
- Friend functions can directly access private members

  - same access privilege as member functions

  - no calls to accessors and mutators =>more efficient
- You can make specific nonmember functions friends for better efficiency!

Friend Functions (2/3)

```
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&);
};
const complex operator-(const complex&, const complex&);
```

Friend Functions (3/3)

```
// 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

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
```

Overload <<

```
ostream& operator<<(ostream& os, const complex& rhs) {
    os << rhs.real() << ‘+’ << rhs.image() << ‘i’ ;
    return os;
}

void f() {
    complex a(2,3), b(4,5);
    cout << a << endl << b << endl; // more elegant!
}
```

- It is common to make operator<< a friend

Return Value of Operator <<

- If you make operator<< return void …

```
void operator<<(ostream& os, const complex& rhs) {
        os << rhs.real() << ‘+’ << rhs.image() << ‘i’ ;
}
void f() {
        complex a(2,3), b(4,5);
        cout << a << endl << b << endl; // compilation error!
}           void
```

Overload >>

- You can use “cin >>” for user-defined types

  - First, make istream& operator>>(istream&, complex&) a friend

  - then,

```
istream& operator>>(istream& is, complex& rhs) {
        is >> rhs.re >> rhs.im ;
        return is;
}
void f() {
        complex a, b;
        cin >> a >> b; // more elegant!
}       istream
```

Exercise (1/4)

- 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 (2/4)

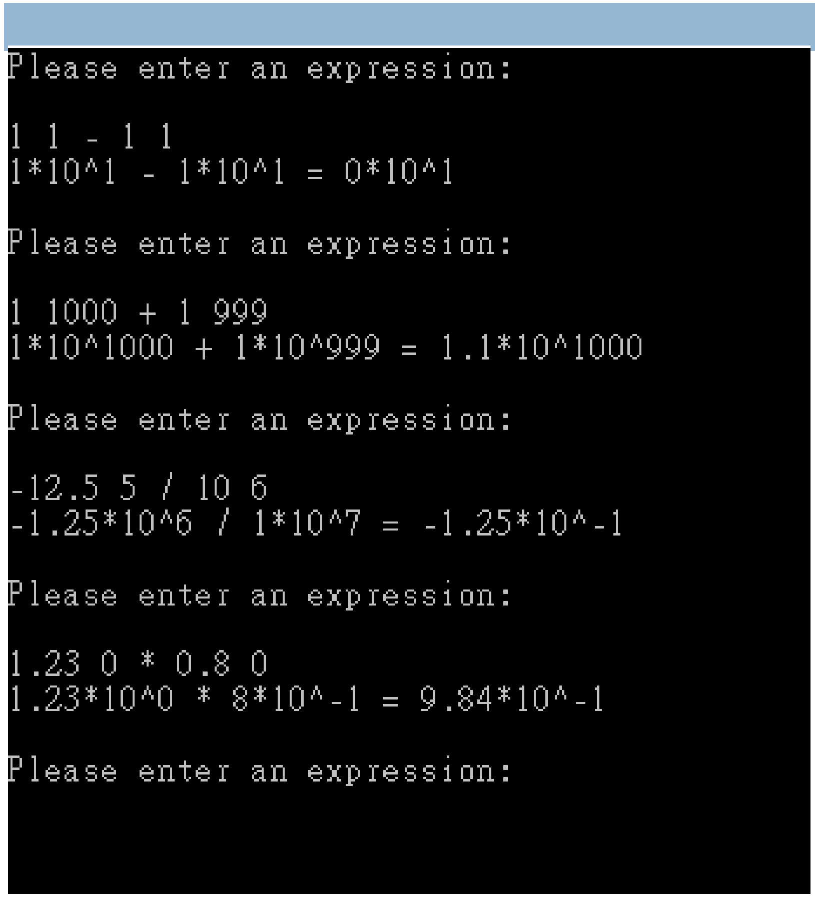
- Please don’t touch the provided main function
  - Just finish operator overloading function declarations/definitions
- Your class should be able to handle operations of large numbers
  - Ex: 1.23*10^1000 / 2.7*10^800 = 4.55556*10^199
- TA will not input an expression with too large difference between two operands
  - Ex: 1*10^1000 + 1*10^10

Exercise (3/4)

Hint: You can use following predefined functions in <cmath> and <cstdlib>:

double pow(double),
double fabs(double),
double log10(double),   ex: log10(120.0) == 2.07918
double ceil(double),   ex: ceil(10.5) == 11.0
double floor(double),   ex: ceil(10.5) == 10.0

Exercise (4/4)



```
Please enter an expression:

1 1 - 1 1
1*10^1 - 1*10^1 = 0*10^1

Please enter an expression:

1 1000 + 1 999
1*10^1000 + 1*10^999 = 1.1*10^1000

Please enter an expression:

-12.5 5 / 10 6
-1.25*10^6 / 1*10^7 = -1.25*10^-1

Please enter an expression:

1.23 0 * 0.8 0
1.23*10^0 * 8*10^-1 = 9.84*10^-1

Please enter an expression:
```