Programming Languages
Lecture12 – OOP (C++ Review 1)

SUNG KYUN KWAN
UNIVERSITY

남 범 석

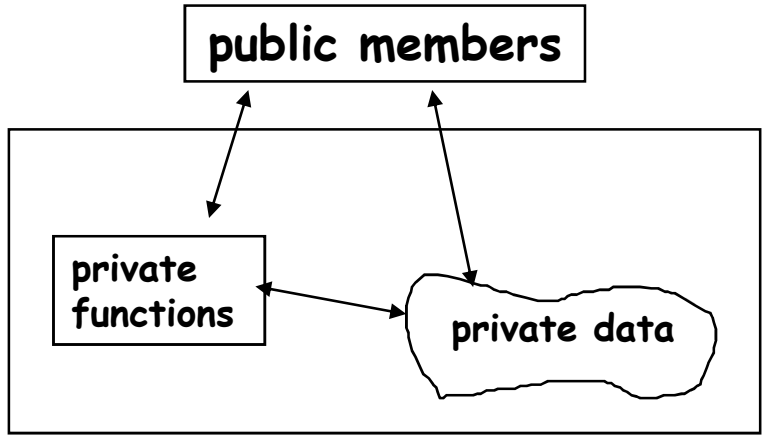
bnam@skku.edu

Object-Oriented Terminology

- A programming language is called OOP if it supports
  - Encapsulation
  - Inheritance
  - Polymorphism

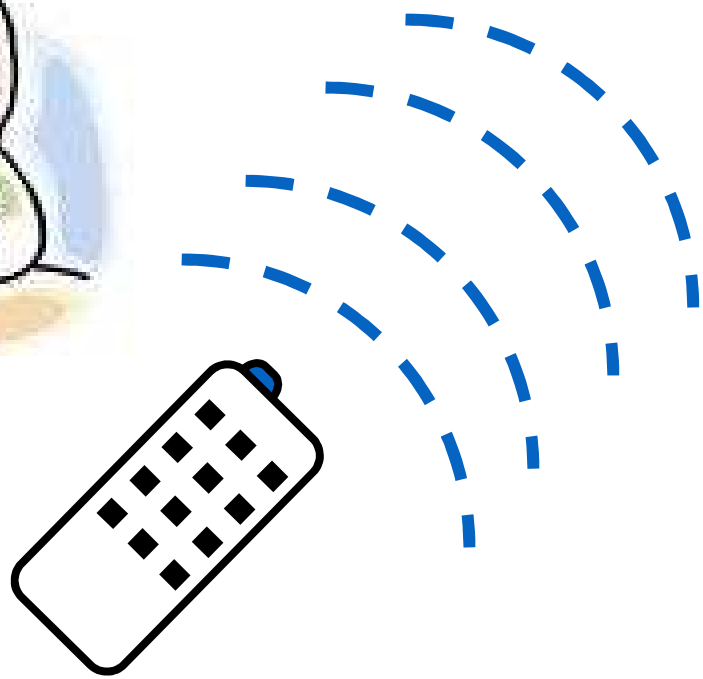
Object-Oriented Programming

- Why abstractions?
  - easier to think about - hide what doesn't matter
  - protection - prevent access to things you shouldn't see
  - plug compatibility
    - replacement of pieces, often without recompilation, definitely without rewriting libraries
    - division of labor in software projects



Data Hiding (Encapsulation)

Customer



TV Product

Manages Complexity

C++ Review

Static Members

- Static member variable:
  - One instance of variable for the entire class
  - Shared by all objects of the class

- Static member function:
  - Can be used to access static member variables
  - Can be called before any class objects are created

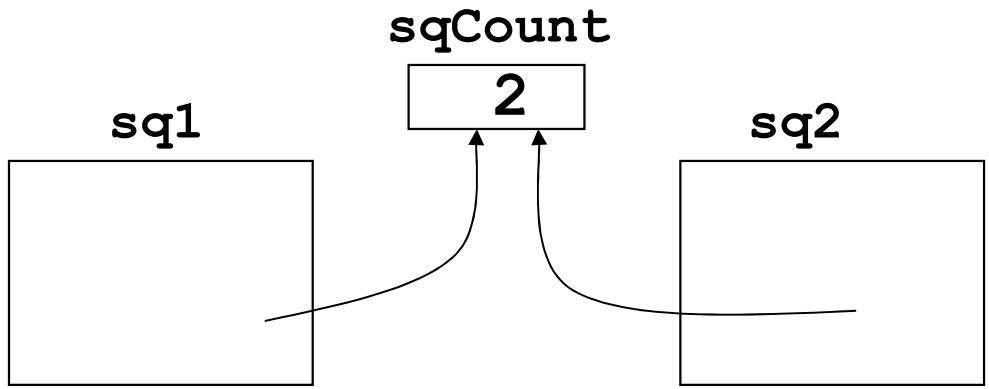
Static Member Variable

```
class Square
{
  private:
    int side;
    static int sqCount;
  public:
    void setSide(int s);
    int getSide();
};

// must be initialized outside of class
int Square::sqCount = 0;

void Square::setSide(int s)
{ this->side = s; }

int Square::getSide()
{ return this->side; }
```

Static Member Variable

- Must be declared in class with keyword static:
- Must be defined/initialized outside of the class:
- Can be accessed or modified by any object of the class:
- Modifications by one object are visible to all objects of the class:

```
int Square::sqCount = 0;
Square sq1, sq2;
```



Static Member Functions

1) Declared with static before return type:

```
class IntVal
{
  public:
    static int getValCount()
    {
       return valCount;
    }
  private:
    int value;
    static int valCount;
};
```

Static Member Functions

2) Can be called independently of class objects, through the class name:

```
cout << IntVal::getValCount();
```

3) **this** pointer cannot be used

There's no associated object with the static member functions.

4) Can be called before any objects of the class have been created

5) Used mostly to manipulate static member variables of the class

11.3 Friends of Classes

- Friend function:
  - a function that is not a member of a class, but has access to private members of the class
  - "Friends of yours know your private things."
- A friend function can be a stand-alone function or a member function of another class
- It is declared with the friend keyword in the function prototype

Friend Function Declarations

1) Friend function may be a stand-alone function:

```
class aClass
{
  private:
   int x;
   friend void fSet(aClass &c, int a);
};

void fSet(aClass &c, int a)
{
   c.x = a;
}
```

Friend Function Declarations

2) Friend function may be a member of another class:

```
class aClass;
class OtherClass
{ public:
     void fSet(aClass &c, int a);
};

class aClass
{ private:
    int x;
    friend void OtherClass::fSet
                          (aClass &c, int a);
};
void OtherClass::fSet(aClass &c, int a){
    c.x = a;
}
```

Friend Class Declaration

- 3) An entire class can be declared a friend of a class:

```
class aClass
{
  private:
    int x;
    friend class frClass;
};

class frClass
{
  public:
    void fSet(aClass &c,int a){c.x = a;}
    int fGet(aClass c){return c.x;}
};
```

Friend Class Declaration

- If **`frClass`** is a friend of **`aClass`**, then all member functions of **`frClass`** have unrestricted access to all members of **`aClass`**, including the private members.

- In general, restrict the property of friendship to only those functions that must have access to the private members of a class.

Memberwise Assignment

- Can use = to assign one object to another, or to initialize an object with an object's data

- Examples (assuming class `V`):

```
V v1, v2;
     ... // statements that assign
     ... // values to members of v1

v2 = v1;     // assignment
V v3 = v2;   // initialization
             // copy constructor is called
```

Copy Constructors

- Special constructor used when a newly created object is initialized to the data of another object of same class
- Default copy constructor is created by a compiler.
- Default copy constructor copies field-to-field
- Default copy constructor works fine in many cases

Copy Constructors

Problems occur when objects contain **pointers** to dynamic storage:

```
class CpClass
{
  private:
    int *p;
  public:
    CpClass(int v=0)
    {
        p = new int;
        *p = v;
    }
    ~CpClass()
    { delete p; }
};
```

Operator Overloading

- Operators such as =, +, and others can be redefined for use with objects of a class

- The name of the function for the overloaded operator is **`operator`** followed by the operator symbol, *e.g.*,
  - **`operator+`** is the overloaded + operator and
  - **`operator=`** is the overloaded = operator

C++ Operator Overloading

- You can overload any of the following operators in C++

```
+       -       *       /       %       ^       &       |       ~
!       =       <       >       +=      -=      *=      /=      %=
^=      &=      |=      <<      >>      <<=     >>=     ==      !=
<=      >=      &&      ||      ++      --      ,       ->*     ->
( )     [ ]     new     delete  new[]   delete[]
```

- Note that () is the function call operator and [] is the subscript operator.
- You can also overload both the unary and binary forms of the following operators:

```
+       -       *       &
```

Operator Overloading

- Operators can be overloaded as
  - instance member functions or as
  - friend functions

- Overloaded operator must have the same number of objects as the standard version.
  - For example, `operator=` must have two objects, since the standard = operator takes two entities.

Overloading Operators as Instance Members

A binary operator overloaded as an instance member needs only one parameter, which represents the operand on the right:

```
class OpClass
{
  private:
    int x;
  public:
    OpClass operator+(const OpClass& right);
};
```

Overloading Operators as Instance Members

- The left operand of the overloaded binary operator is the calling object
- The implicit left parameter is accessed through the **`this`** pointer

```
OpClass OpClass::operator+(const OpClass& r)
{
    OpClass sum;
    sum.x = this->x + r.x;
    return sum;
}
```

Invoking an Overloaded Operator

- Operator can be invoked as a member function:

```
OpClass a, b, s;
s = a.operator+(b);
```

- It can also be invoked in the more conventional manner:

```
OpClass a, b, s;
s = a + b;
```

Overloading Assignment

- Overloading assignment operator solves problems with object assignment when object contains pointer to dynamic memory.

- Assignment operator is most naturally overloaded as an instance member function

- Needs to return a value of the assigned object to allow cascaded assignments such as

```
a = b = c;
```

Overloading Assignment

Assignment overloaded as a member function:

```
class CpClass
{
   private:
      int *p;
   public:
      CpClass(int v=0)
      {
         p = new int;
         *p = v;
      }
      ~CpClass(){delete p;}
      CpClass& operator=(const CpClass&);
};
```

Overloading Assignment

Implementation returns a value:

```
CpClass& CpClass::operator=(const CpClass& r)
{
    *p = *r.p;
    return *this;
};
// without &, copy constructor will be called.
```

Invoking the assignment operator:

```
CpClass a, x(45);
a.operator=(x); // either of these
a = x;          // lines can be used
```

Review: Copy Constructor

- Copy Constructor is called when
  - A class object is initialized with another object.
    - `MyClass obj1(obj2);`
      `MyClass obj1 = obj2;`
  - A class object is passed to a function using ***call-by-value***.
    - `void function(MyClass arg){ … }`
  - A class object is returned from a function using ***call-by-value***.
    - `MyClass function() { … }`
  - In order to avoid calling copy constructor for the above cases, use ***call-by-reference***.
    - `MyClass& function(MyClass& arg) { … }`

Notes on Overloaded Operators

- Can change the entire meaning of an operator
- Most operators can be overloaded
- Cannot change the number of operands of the operator
- Cannot overload the following operators:

```
?:   .   .*   ::   sizeof   #   ##
```

Overloading Operators

- `++, --` operators overloaded differently for prefix vs. postfix notation

- Overloaded relational operators should return a **`bool`** value

- Overloaded stream operators >>, << must return **`istream`**, **`ostream`** objects and take **`istream`**, **`ostream`** objects as parameters

Overloading Operators

- C++ "Hello World"

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

int main()
{
    cout << "Hello World" << endl;
}
```

- std::operator<< *(pronounced as output operator)*
  - `std::operator<<` is a function that inserts a given parameter object into a given stream.
  - function definition of `std::operator<<`
    - `ostream& operator<<(ostream& os, const T& obj)`

Overloading Operators

- How to read user's input?

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

int main()
{
    string name;
    cin >> name;
    cout << "Hello, " << name << endl;
}
```

- `std::cin  >> variable;`
- **std::cin** is an object of class *istream* that represents the standard input stream.
- By default, it reads from user's keyboard.

Overloading operator<<

- operator<< should be declared as a friend in your class.

```
ostream& operator<<(ostream& os,
                                const TimeClass& t)
{
    os << t.hour << ":" << t.min << ":" << t.sec;
    return os;
};
```

```
TimeClass t1(2, 30, 0);

cout << "Our class starts at " << t1 << endl;

// prints out "Our class starts at 2:30:0"
```

Type Conversion Operators

- Conversion Operators are member functions that tell the compiler how to convert an object of the class type to a value of another type

- The conversion information provided by the conversion operators is automatically used by the compiler in assignments, initializations, and parameter passing

Syntax of Conversion Operators

- Conversion operator must be a member function of the class you are converting from
- The name of the operator is the name of the type you are converting to
- The operator does not specify a return type

Conversion Operator Example

- To convert from a class `IntVal` to an integer:

```
class IntVal
{
   int x;
 public:
   IntVal(int a = 0){x = a;}
   operator int(){return x;}
};
```

- Automatic conversion during assignment:

```
IntVal obj(15);

int i;
i = obj;
cout << i; // prints 15
```

Convert Constructors

Convert constructors are constructors with a single parameter of a type other than the class

```
class CCClass
{
      int x;
   public:
      CCClass()                  //default
      CCClass(int a, int b);

      CCClass(int a);            //convert
      CCClass(string s);         //convert
;
```

Example of a Convert Constructor

The C++ `string` class has a convert constructor that converts from C-strings:

```
class string
{
    public:
       string(char *);  //convert
       …
};
```

Uses of Convert Constructors

- Automatically invoked by the compiler to create an object from the value passed as parameter:

```
string s("hello");   //convert C-string
CCClass obj(24);     //convert int
```

- Compiler allows convert constructor to be invoked with assignment-like notation:

```
string s = "hello"; //convert C-string
CCClass obj = 24;   //convert int
```

Uses of Convert Constructors

- Convert constructors allow functions that take the class type as parameter to take parameters of other types:

```
void myFun(string s); // needs string
                      // object
myFun("hello");       // accepts C-string

void myFun(CCClass c);
myFun(34);            // accepts int
```

operator++()

- **Pre-increment** operator

```
Date& Date::operator++()
{
        increment();
        return *this;
}
```

- **Post-increment** operator
  - Note that the dummy integer parameter does not have a parameter name.

```
Date Date::operator++(int)
{
        Date temp = *this;
        increment();
        return temp; // value return
}
```

SimpleArray Class with operator[]

```
class SimpleArray3{
   private:
        int length;
        int* data;
   public:
        SimpleArray3(int len, int* arr){
                length = len;
                data = new int[length];
                for(int i=0;i<length;i++){
                        data[i] = arr[i];
                }
        }

        int& operator[](int idx){
            if(idx < length)
                return data[idx];
            return data[length-1];
        }

        ~SimpleArray3(){ delete[] data; }
};
```

Review: Reference

```
#include <iostream>

using namespace std;

int& get_reference(int* arr, int i)
{
   return arr[i];
}

int main()
{
    int arr[] = {1, 2, 3, 4};

    get_reference(arr, 1) = 100;

    cout << arr[0] << " " << arr[1] << " "
         << arr[2] << " " << arr[3] << endl;
}
```

- Q1: What will be printed?
- Q2: What if & is not in the return type?
- Q3: What if the return type is int* ?