# 505 22240 / ESOE 2012 Data Structures: Lecture 3

## Classes, Exceptions and Templates

### § Two-Dimensional Array and Pointer

・A two-dimensional array is implemented as an “array of arrays”.

・Two-dimensional array: an array of references (pointers) to one-dimensional arrays.

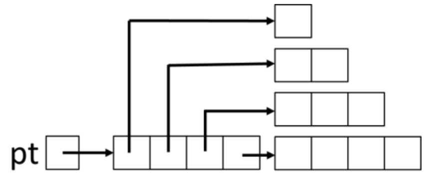
・Pascal’s Triangle:

```
            1        ← row Ø
          1   1
        1   2   1
      1   3   3   1
    1   4   6   4   1
  1   5   10  10  5   1   ← row 5
```

⇨ row i represents coefficients of $(x+1)^i$

e.g. $(x+1)^4 = x^4 + 4x^3 + 6x^2 + 4x + 1$

```
int** pascalTriangle(int n) {
    int** pt = new int*[n];
    for (int i = 0; i < n; i++) {
        pt[i] = new int[i+1];
        pt[i][0] = 1;        // left 1
        for (int j = 1; j < i; j++) {
            // middle values
            pt[i][j] = pt[i-1][j-1] + pt[i-1][j];
        }
        pt[i][i] = 1;        // right 1
    }
    return pt;
}
```




### § Classes

#### ◎Class Structure

・A class consists of members:

①Data members (member variables): variables or constants.

②Member functions (methods): define behavior of the class.

・Example: **Counter**

```
class Counter {
public:       // access control
    Counter( );                  // initialization
    int getCount( );             // get the current count
    void increaseBy(int x);      // add x to the count
private:      // access control
    int count;                   // the counter’s value
};
```

・Definitions of member functions

```
Counter::Counter( )                  // constructor
    {count = 0;}
int Counter::getCount( )             // get current count
    {return count;}
void Counter::increaseBy(int x)      // add x to the count
    {count += x;}
```

· Usage

```
Counter ctr;                    // an instance of Counter
cout << ctr.getCount( ) << endl;
// prints the initial value 0
ctr.increaseBy(3);              // increase by 3
cout << ctr.getCount( ) << endl;    // prints 3
ctr.increaseBy(5);                  // increase by 5
cout << ctr.getCount( ) << endl;    // prints 8
```

· Note: if no access specifier is given, the default is <u>private</u> for classes and <u>public</u> for structures.

## ◎The "public" AND "private" Keywords

· *public*: anyone can access.

· *private*: method or field is <u>invisible</u> & <u>inaccessible</u> to other classes.

→ Instance variables are normally declared <u>private</u> and methods are normally declared <u>public</u>.

· Why use "private"?

① To prevent data from being corrupted by other classes.

② You can improve the implementation without causing other classes that depend on it to fail.

· e.g.

```
class Date {
private:
    int day;
    int month;
    //…
    void setMonth(int m) {
        month = m;
    }
public:
    Date(int month, int day) {
        [Implementation with error-checking code here.]
    }
};
```

· execution:

```
Date d(10, 12);
d.day = 26;             // Failed
d.setMonth(4);          // Failed again
```

## ◎Member Functions

· Two major categories:

①Accessor functions: only read class data, with "`const`".

②Update functions: can alter class data.

· Example: **Passenger**

```
class Passenger {
public:
  Passenger( );            // constructor
  //In-class function
  bool isFrequentFlyer( ) const {return isFreqFlyer;}
  void makeFrequentFlyer(const string& newFreqFlyerNo);
private:
  string name;
```

```
  MealType mealPref;
  bool isFreqFlyer;
  string freqFlyerNo;
};

void Passenger::makeFrequentFlyer(const string&
newFreqFlyerNo) {
  isFreqFlyer = true;
  freqFlyerNo = newFreqFlyerNo;
}
```

◎Constructors

```
Passenger( );            // default constructor
Passenger(const string& nm, MealType mp, const string&
ffn="NONE");             // "NONE" is default argument
Passenger(const Passenger& pass);    // copy constructor
```

· Definitions of constructors

```
Passenger::Passenger( ) {        // default constructor
    name = "--NO NAME-- ";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}
// constructor given member values
Passenger::Passenger(const string& nm, MealType mp, const
string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn is given
    freqFlyerNo = ffn;
}
// copy constructor
Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}
```

· Usage

```
Passenger P1;            // default constructor
Passenger P2("John Smith", VEGETARIAN, "293145");
// 2nd constructor
Passenger P3("Peter Jackson", REGULAR);
// not a frequent flyer
Passenger P4(P3);            // copied from P3
Passenger P5 = P2;           // copied from P2
Passenger* PP1 = new Passenger;     // default constructor
Passenger* PP2 = new Passenger("John Blow", NO_PREF);
// 2nd constructor
Passenger pa[20];            // default constructor
```

## ◎Initializer List: to deal with initialization of member variables that are classes

(without an assignment operator, =) `: member_name(initial_value), …`

・Rewrite the 2nd Passenger constructor:

```
Passenger::Passenger(const string& nm, MealType mp, const
string& ffn) : name(nm), mealPref(mp), isFreqFlyer(ffn !=
"NONE") { freqFlyerNo = ffn; }
```

## ◎Destructors

・The destructor for a class T is denoted as ~T: no arguments and no return type.

・Example

```
class Vect {
public:
    Vect( );              // default constructor
    Vect(int n);          // constructor, given size
    ~Vect( );             // destructor
private:
    int* data;            // an array
    int size;             // number of array entries
};

Vect::Vect() {                // default constructor
    size = 10;
    data = new int[10];
}

Vect::Vect(int n) {           // constructor with given size
    size = n;
    data = new int[n];        // allocate array
}

Vecr::~Vect( ) {              // destructor
    delete [ ] data;          // free the allocated array
}
```

## ◎Memory Allocation

・Using Vect class:

```
Vect a(100);     // a is a vector of size 100
Vect b = a;      // initialize b from a (DANGER!)
Vect c;          // c is a vector (default size 10)
c = a;           // assign a to c (DANGER!)
```

・Shallow copy: a shallow copy of an object (collection, or class) copies all of the member field values, i.e., a copy of the class structure, not the elements. With a shallow copy, two collections share the individual elements.

`Vect b = a` sets `b.data = a.data` (pointer copy)

`c = a` lost the pointer to c's original 10-element array. → memory leak

・a, b, and c all have members that point to the same array.

・Copy constructor: for a class T → `T(const T& t)`

・Deep copy:

```
// copy constructor from a
Vect::Vect(const Vect& a) {
    size = a.size;            // copy size
    data = new int[size];     // allocate new array
```

```
    for (int i = 0; i < size; i++) {
        data[i] = a.data[i];        // copy the contents
    }
}
// assignment operator from a
Vect& Vect::operator=(const Vect& a) {
    if (this != &a) {           // avoid self-assignment
        delete [ ] data;        // delete old array
        size = a.size;          // set new size
        data = new int[size];   // allocate new array
        for (int i = 0; i < size; i++) {
            data[i] = a.data[i];        // copy the contents
        }
    }
    return *this;
}
```

・For any instance of a class object, "`this`" is defined to be the address of this instance.

★ Every class that allocates its own objects using `new` should:

① Define a destructor to free allocated objects.

② Define a copy constructor, which allocates its own new member storage and copies the contents of member variables.

③ Define an assignment operator, which deallocates old storage, allocates new storage, and copies all member variables.

◎The "friend" keyword

・to access *protected* and *private* member data of other classes.

★ Friend function:

```
class SomeClass {
private:
    int secret;
public:
    friend ostream& operator<<(ostream& out, const SomeClass&
x);     // give << operator access to secret
};
ostream& operator<<(ostream& out, const SomeClass& x)
    { cout << x.secret; }
```

・Multiple classes:

```
class Humidity;
class Temperature {
private:
    int m_nTemp;
public:
    Temperature(int nTemp) { m_nTemp = nTemp; }
    friend void PrintWeather(Temperature& cTemperature,
Humidity& cHumidity);
};
class Humidity {
private:
    int m_nHumidity;
public:
```

```
    Humidity(int nHumidity) { m_nHumidity = nHumidity; }
    friend void PrintWeather(Temperature& cTemperature,
Humidity& cHumidity);
};

void PrintWeather(Temperature& cTemperature, Humidity&
cHumidity) {
    std::cout << "The temperature is " << cTemperature.m_nTemp
<< " and the humidity is " << cHumidity.m_nHumidity <<
std::endl;
}
```

★ Friend class

```
class Vector {          // a 3-element vector
public: //…
private:
    double coord[3];
    friend class Matrix;    // give Matrix access to coord
};
class Matrix {          // a 3×3 matrix
public:
    Vector multiple(const Vector& v);
    // multiple by vector v
private:
    double a[3][3];
};
```

```
Vector Matrix::multiply(const Vector& v) {
    Vector w;
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            w.coord[i] += a[i][j] * v.coord[j];
            // access to coord of v allowed
    return w;
}
```

◎Nesting Classes

```
class Book {
public:
    class Bookmark {
    //… (Bookmark definition here)
    };
    //… (Remainder of Book definition)
};
```

·Use **Book::Bookmark** to refer to this nested class.

◎Interface of a Class

① Prototypes for public methods,

② plus descriptions of their behaviors.

◎Abstract Data Type (ADT)

· A class with a well-defined interface, but implementation details are <u>hidden</u> from other classes.

## ◎Invariant

・A fact about a data structure that is always true.

・e.g, "A Date object always represents a valid date."

★ Not all classes are ADTs! Some classes just store data (no invariants).

## ◎STL Vector Class

・A vector can be resized dynamically.

・Each instance of an STL vector can only hold objects of <u>one</u> type.

・Example:

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

vector<int> scores(100);      // 100 integer scores
vector<char> buffer(500);     // buffer of 500 characters
vector<Passenger> passenList(20);    // list of 20 Passengers

int i = 12;
cout << scores[i];                // index (range unchecked)
buffer.at(i) = buffer.at(2*i);    // index (range checked)

vector<int> newScores = scores;
// copy scores to newScores
scores.resize(scores.size() + 10);
// add room for 10 more elements
```

## § Exceptions

・When a run-time error occurs in C++: it "throws an exception" → (Exception object).

・Prevent the error by "catching" the Exception.

## ◎Purpose: surviving errors

・By catching exceptions, you can recover from an unexpected error.

・e.g.: try to open a file that doesn't exist. You can catch exception, print error message, and continue.

```
try {
    fin.open("~esoe/ds/exam.pdf", ios::in);
    getline(fin, str, '\n');
    //…
}
catch (FileNotFoundException& e1) {
    cout << "Error msg … ";
}
catch (IOException& e2) {
    fin.close( );
}
```

★What does this code do?

(a) Executes the code inside "try".

(b) If "try" code executes normally, skip "catch" clauses.

(c) If "try" code throws an exception, do not finish the "try" code. Jumps to first "catch" clause. "Matches" exception object thrown is the same class/subclass of exception type in "catch" clauses.

・When the "catch" clause finishes executing, jumps to the next line of code after all catch clauses.

・Only the first matching "catch" is executed.

・Each "catch" clause is called an exception handler.

・Use "catch (...)" to catch all exceptions. → last handler.

## ◎Exception constructors

・Exception types often form hierarchies.

・e.g.: one generic exception, `MathException`, representing all types of mathematical errors.

```
class MathException {
public:
    MathException(const string& err): errMsg(err) { }
    // constructor
    string getError( ) {return errMsg;}
    // access error message
private:
    string errMsg;              // error message
};
```

・Using inheritance to define new exception types:

```
class ZeroDivide : public MathException {
public:
    ZeroDivide(const string& err): MathException(err) { }
};          // divide by zero
class NegativeRoot : public MathException {
public:
    NegativeRoot(const string& err): MathException(err) { }
};          // negative square root
```

## ◎Exception specification

・When we declare a function, we should also specify the exceptions it might throw.

```
void calculator( ) throw (ZeroDivide, NegativeRoot) {
    //...
    try {
        //...
        if (divisor == 0)
            throw ZeroDivide("Divide by zero in Module X");
    }
    catch (ZeroDivide& zde) {
    // handle division by zero.
    }
    catch (MathException& me) {
    // handle any math exception other than division by zero.
    }
    //...
}
```

・If a function does not provide a "throw" specification, it may throw any exception.

```
void fcn1( );               // can throw any exception
void fcn2( ) throw( );      // can throw no exceptions
```

## ◎A generic exception class

・Serves as the "mother of all exceptions".

```
class RuntimeException {        // Base class
private:
    string errorMsg;
```

```
public:
    RuntimeException(const string& err) {errorMsg = err;}
    string getMessage( ) const {return errorMsg;}
};
```

## § Templates

- Allow functions and classes to operate with generic types, to work on multiple data types without being written for each one.

### ◎Function Templates

・e.g. minimum of two integers:

```
int integerMin(int a, int b) {return (a < b ? a : b);}
```

・A generic function for an arbitrary type T:

```
template <typename T>
T genericMin(T a, T b) {
    return (a < b ? a : b);
}
```

・The compiler looks at the argument types and determines which form of the function to instantiate.

```
cout << genericMin(3, 4) << ' '
     // = genericMin<int>(3, 4)
     << genericMin(1.1, 3.1) << ' '
     // = genericMin<double>(1.1, 3.1)
     << genericMin('t', 'g') << endl;
     // = genericMin<char>('t', 'g')
```

### ◎Class Templates

・A simple class template:

```
template <typename T>
class BasicClass {
public:
    BasicClass(const T& t): myObj(t) { }     // constructor
    T Get( ) const {return myObj;}
    void Set(const T& t) {myObj = t;}
private:
    T myObj;
};
```

・To instantiate a concrete instance of the class BasicClass, provide the class name followed by the actual type parameter enclosed in angled brackets (< … >).

```
BasicClass<float> f;
```

・Use typedef to make your code more readable:

```
typedef BasicClass<float> Float;
Float f(5.5f);
cout << f.Get( ) << endl;
f.Set(12.3f);
cout << f.Get( ) << endl;

typedef BasicClass<string> String;
String s("Steve");
cout << s.Get( ) << endl;
s.Set("Apple");
cout << s.Get( ) << endl;
```