ECE 462
Object-Oriented Programming using C++ and Java

Lecture 6

Yung-Hsiang Lu
yunglu@purdue.edu

Lab 3: Creating Multi-File C++ using Managed Project in eclipse

You can also use eclipse on ECN-Linux computer. The procedure is the same.

Prevent Instantiation
(no object of this class)

- Java: abstract class

```
abstract class User {
}
User uobj ...                    // error
```

- C++: pure virtual function

```
class Shape {
    public:
        virtual double area() = 0;
};
Shape sobj ...                   // error
```

Prevent Inheritance
(no derived class)

- Java: add final in front of class

```
class User {
final class StudentUser extends User {
class UndergradStudentUser extends StudentUser { }  // error
```

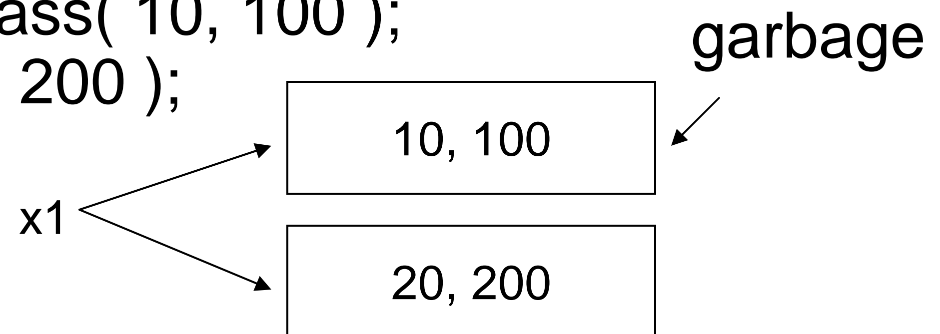
- C++: make constructor private

```
class X {
        int n;
        X( int nn ) { n = nn; }             // constructor is private
};
class Y : public X {                        // error
```

Memory Management
(more in chapter 15)

- In Java, unused memory will be automatically reclaimed (called garbage collection).

```
AClass x1 = new AClass( 10, 100 );
x1 = new AClass( 20, 200 );
```

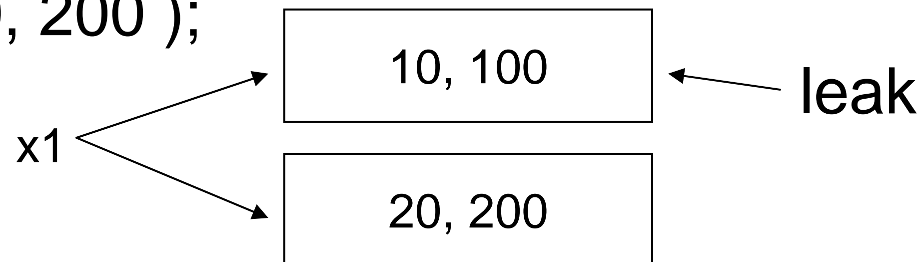


- It is unnecessary to worry about the objects that cannot be reached. Java reclaims the memory.
- You have to remove all references to the object.
- Too much garbage, however, degrades performance.

C++ Object Destruction

- In C++, unreachable memory is lost, "memory leak".

```
AClass * x1 = new AClass( 10, 100 ); // x1 is a pointer
delete x1;
x1 = new AClass( 20, 200 );
```



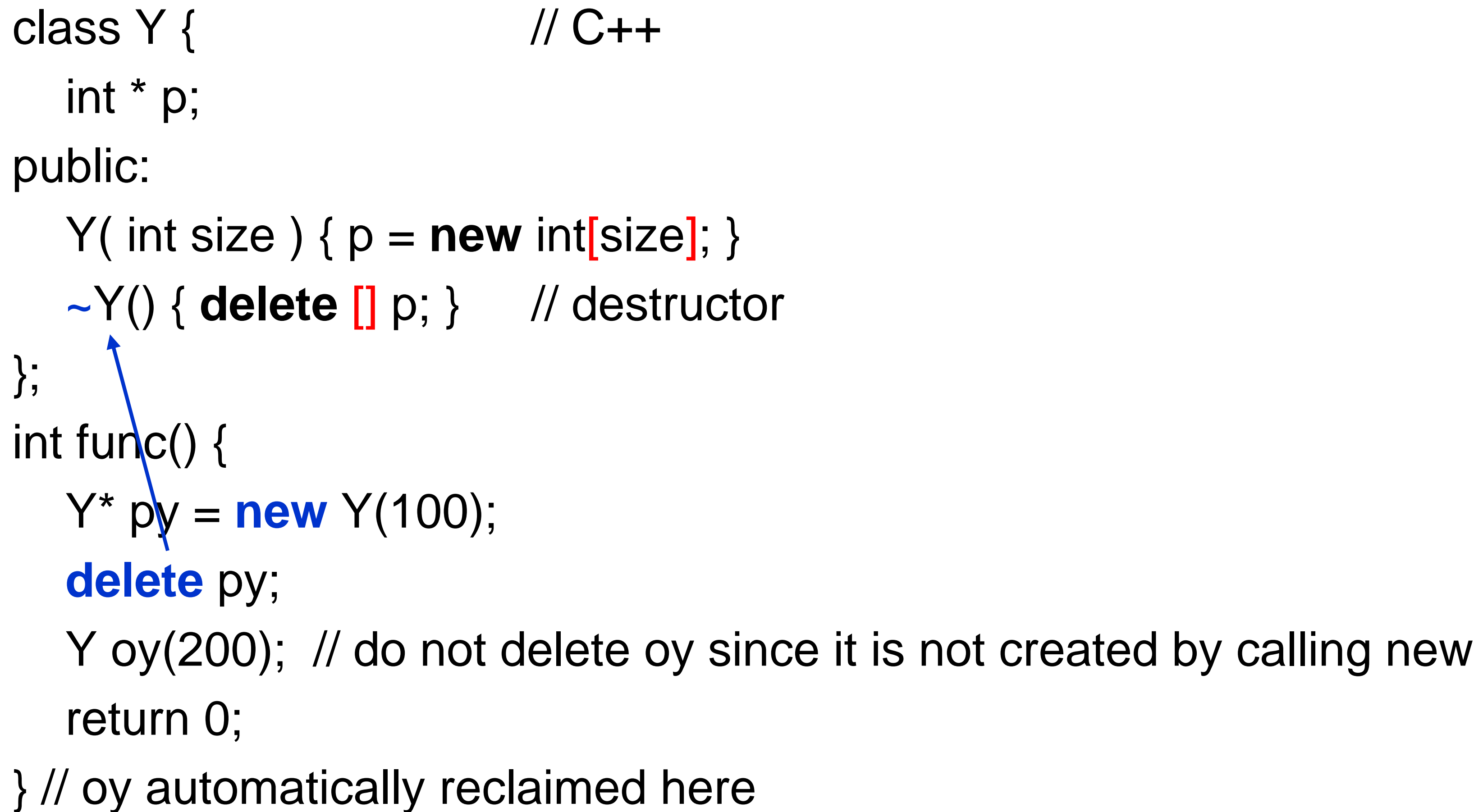
- To prevent memory leak
  delete x1;
- Do not use malloc / free in C++.

C++ Memory Management

- two ways to create objects

```
{
    AClass * x1 = new AClass( 10, 100 ); // x1 is a pointer
    AClass x2( 10, 100 ); // x2 is an object
} // x2's memory automatically reclaimed, x1's is not (memory leak)
```

- x2 automatically destroyed when out of scope
- new – delete pair
  - call delete only if new is called earlier
  - in the same level (not necessarily the same function)
  - do **not mix** new – malloc, delete – free, malloc – delete
- use “**valgrind --tool=memcheck --leak-check=yes executable**” to check memory leak



```
class Y {                        // C++
    int * p;
public:
    Y( int size ) { p = new int[size]; }
    ~Y() { delete [] p; }        // destructor
};
int func() {
    Y* py = new Y(100);
    delete py;
    Y oy(200);  // do not delete oy since it is not created by calling new
    return 0;
} // oy automatically reclaimed here
```

Destructors should always be virtual.

Constructor: "naturally" virtual Destructor: should declare virtual

```
#include <iostream>
using namespace std;
class X
{
public:
  X() { cout << "X() "; }
  ~ X() { cout << "~X() "; }
};
class Y: public X
{
public:
  Y() { cout << "Y() "; }
  ~ Y() { cout << "~Y() "; }
};
```

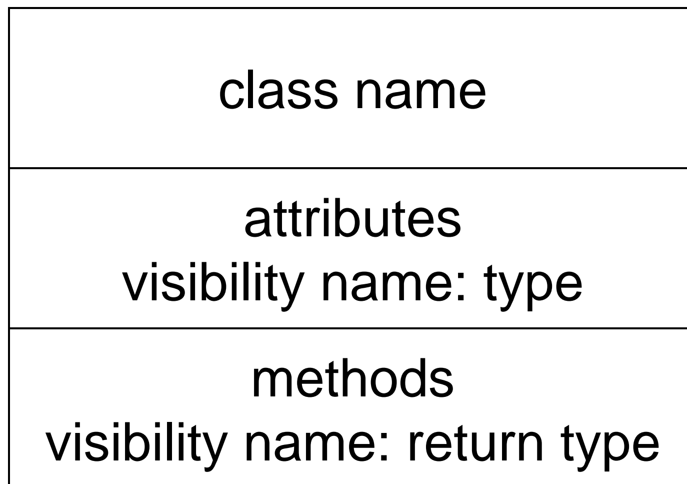
```
class A
{
public:
  A() { cout << "A() "; }
  virtual ~A() { cout << "~A() "; }
};
class B: public A
{
public:
  B() { cout << "B() "; }
  virtual ~B() { cout << "~B() "; }
};
```

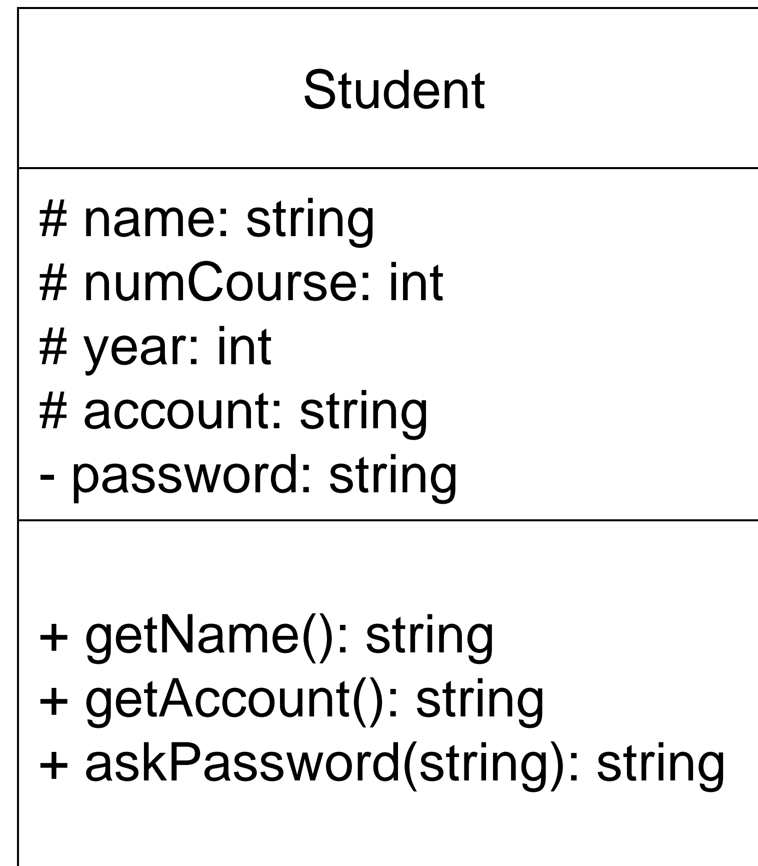
```
int main(void)
{
  X * xptr[2];
  xptr[0] = new X();   // X()
  xptr[1] = new Y();   // X() Y()
  delete xptr[0];      // ~X()
  delete xptr[1];      // ~X()

  A * aptr[2];
  aptr[0] = new A();   // A()
  aptr[1] = new B();   // A() B()
  delete aptr[0];      // ~A()
  delete aptr[1];      // ~B() ~A()

  Y * yptr;
  yptr = new Y();      // X() Y()
  delete yptr;         // ~Y() ~X()
  cout << endl;

  B * bptr;
  bptr = new B();      // A() B()
  delete bptr;         // ~B() ~A()
  return 0;
}
```

Why Are We Doing This?

- This is only one way to increase maxCount $\Rightarrow$ called by the constructor, protect its value from accidental modification
- This is only one way to increase or decrease cxCount.
- The static variables are shared by all objects in the class.
- Constructor and destructor usually "symmetric".
  - memory allocation and object creation
  - counter increment and decrement
- This is an example of **encapsulation**.

UML Class (Unified Modeling Language)





+ public
# protected
- private

ECE 462
Object-Oriented Programming using C++ and Java

Lecture 7

Yung-Hsiang Lu
yunglu@purdue.edu

UML: Unified Modeling Language

- international standard to represent designs
- express both static and dynamic behavior
  - static: class diagrams, association, composition ...
  - dynamic: sequence diagram, statechart ...
- language independent, one of the most important ways to communicate about software designs
- tools available to automatically generate code and document from design, or from code to generate design

Lab 4 Unified Modeling Language using Visual Paradigm

- Fig 14.1 use case diagram
- Fig 14.2 and 14.3 class diagrams
- Fig 14.6 composition
- Fig 14.11 sequence diagram
- Fig 14.16 statechart diagram

C++ Template

- Create a class that contains one (or more) attribute whose type is not-predefined.
- The type is determined when an object is created by specifying the type.
- This type is determined at compile-time.
- Conventionally we use “T” to represent the undefined type. You can use other symbols.

C++ Template

```
//TemplateX.cc
#include <string>
#include <iostream>
using namespace std;
template <class T> class X {
// T is only a place holder
// replace T with any identifier
    T datum;
public:
    X( T dat ) : datum( dat ) {}
    T getDatum(){ return datum; }
};
```

```
int main()
{
    int x = 100;
    X<int> xobj_1( x );
    X<double> xobj_2( 1.234 );
    double d = xobj_2.getDatum();
    cout << d << endl;
    string str = "hello";
    X<string> xobj_3( str );
    string ret1 = xobj_3.getDatum();
    cout << ret1 << endl;
    return 0;
}
```

C++ Template (2 types)

```
#include <iostream>
#include <string>
using namespace std;
template <class T1, class T2> class Container2
{
public:
  Container2(T1 t1in, T2 t2in):   c2_t1(t1in), c2_t2(t2in) { /* nothing */ }
  T1 getT1(void) { return c2_t1; }
  T2 getT2(void) { return c2_t2; }
private:
  T1 c2_t1;
  T2 c2_t2;
};
```

```
int main(void)
{
  Container2<int, char> obj1(15, 'h');
  cout << obj1.getT1() << " " << obj1.getT2() << endl;

  Container2<string, float> obj2("ece462", 6.57841);
  cout << obj2.getT1() << " " << obj2.getT2() << endl;
  return 0;
}

/* output
15 h
ece462 6.57841
*/
```

C++ Vector

- contiguous memory
  - efficient access (**array-like** index)
  - efficient insert / delete at the end
  - **inefficient** insert / delete at the front
  - automatic expand / shrink allocated memory (allocate more than necessary to reduce allocation / release / copy overhead)
  - occasional copying of the whole vector
- iterator: pointer to traverse a vector or other STL (standard template library) container

```
vector<int> v;
cout << "\nvector size is: " << v.size() << endl;
vector<int>::iterator p = v.begin();
while ( p != v.end() )
    cout << *p++ << "  ";
```

```
// VectorBasic.cc
#include <iostream>
#include <vector>
using namespace std;
void print( vector<int> );
int main()
{
    vector<int> vec;
    vec.push_back( 34 );
    vec.push_back( 23 );
     // size is now 2
    print( vec );          // 34 23
    vector<int>::iterator p;
    p = vec.begin();
    *p = 68;
    *(p + 1) = 69;     // 2nd element
     //  *(p + 2) = 70;       // WRONG
     // only two elements now
    print( vec );          // 68 69
    vec.pop_back();
    // size is now 1
    print( vec );          // 68
    vec.push_back(101);
    vec.push_back(103);
    // size is now 3
    int i = 0;
    while ( i < vec.size() )
        cout << vec[i++] << "  ";
    cout << endl;
    // 68 101 103
```

```
    vec[0] = 1000;
    vec[1] = 1001;
    vec[2] = 1002;
    print( vec );
    // 1000 1001 1002
    return 0;
}
void print( vector<int> v ) {
    cout << "\nvector size is: " << v.size() << endl;
    vector<int>::iterator p = v.begin();
    while ( p != v.end() )
        cout << *p++ << "  ";
    cout << endl << endl;
}
```

C++ Vector Insert and Erase

```
vector<int> vec(5);               // initially 5 elements, all zeros
vec.erase( vec.begin() );         // erase the first element
vec.insert( vec.end(),  7 );      // insert at the end
vec.insert( vec.begin() + 1, 3 );
        // add 3 as the second element
vec.erase( find( vec.begin(), vec.end(), 3 ) );
        // find the first element equal to 3 and remove it
```

```
//VectorFrontBackResize.cc
#include <iostream>
#include <vector>
using namespace std;
int sum( vector<int> vec ) {
   int result = 0;
   vector<int>::iterator p = vec.begin();
   while ( p != vec.end() ) result += *p++;
   return result;
}
int main()
{
   vector<int> v1(100);           // allocate 100 elements, initialized to 0
   cout << v1.size() << endl;     // 100
   cout << sum( v1 ) << endl;   // 0
```

```
v1.push_back( 23 );
cout << v1.size() << endl;           // 101
cout << sum( v1 ) << endl;           // 23
cout << v1.front() << endl;          // 0
cout << v1.back() << endl;           // 23

vector<int> v2(150, 2);              // 150 elements, initialized to 2
cout << sum( v2 ) << endl;           // 300
v2.clear();
cout << v2.empty() << endl;          // true
```