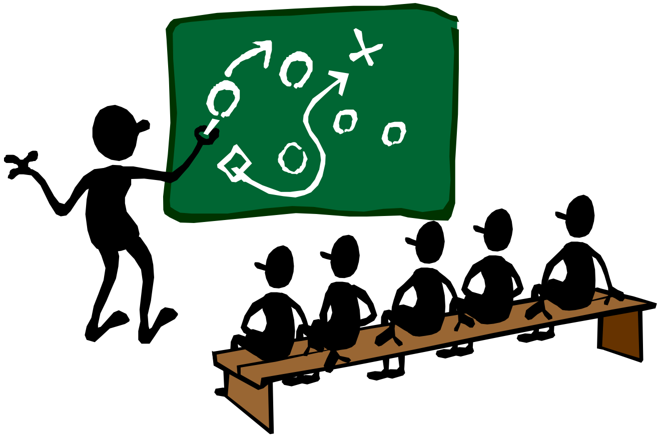
C++ Programming Language
Chapter 14 Inheritance



Juinn-Dar Huang

Associate Professor

jdhuang@mail.nctu.edu.tw

May 2011

Learning Objectives

- Inheritance basics
  - derived class vs. base class
  - constructors of derived class
  - what’s inherited in derived class?
  - rules for access controls
- Protected members
- Revisit ctor/dtor/copy ctor/copy assignment operator in derived class
- Public/protected/private inheritance
- Single inheritance vs. multiple inheritance

Introduction to Inheritance

- Inheritance is one of key concepts of OOP
- In OOP, a class is used to represent a concept
  - polygon, rectangle, ellipse, circle, shape, …
- Concepts don’t exist in isolation; they are related
  - rectangle is a special kind of polygon
  - circle is a special kind of ellipse
  - they are all shapes
- C++ provides a mechanism to express such hierarchical relationships
  - inheritance
  - base class vs. derived class (e.g., polygon vs. rectangle)

Inheritance Basics

- Class D is **inherited** from class B
  - base class B represents a more generalized concept
  - derived class D represents a specialized concept that inherits properties from base class
  - derived class can add new properties and/or refine existing properties for its appropriate use

- Derived class
  - automatically get something from its base class
    - all data members (still with access limitations, discuss later)
    - all member functions except for private ones (with few exceptions, discuss later)
  - can add its own data members and member functions

Terminology

- Base class and derived class are the most commonly used terms

Other terms

- Parent class
  - refer to base class
- Child class
  - refer to derived class
- Ancestor class and descendent class

Inheritance Example (1/3)

- Define a strcut (class) for employee record in a firm

```
struct Employee {
    string first_name, family_name;
    char middle_initial;
    Date hiring_date;
    short department;
    // …
};
```

Inheritance Example (2/3)

- Now we want to define a struct (class) for manager record
  - two methods
- Method 1

```
struct Manager {
    Employee emp;              // manager’s employee record
    Employee* group[100];      // people managed
    short level;
    // …
};
```

- Programmer knows a manager is (also) an employee but compiler does not!

```
Manager m;
Employee *pe = &m;  // “Error,” compiler says,
                    // “hey, a manager is NOT an employee”
```

Inheritance Example (3/3)

- Is there any way to let compiler know that a manager is (also) an employee? ➔ Yes, of course!

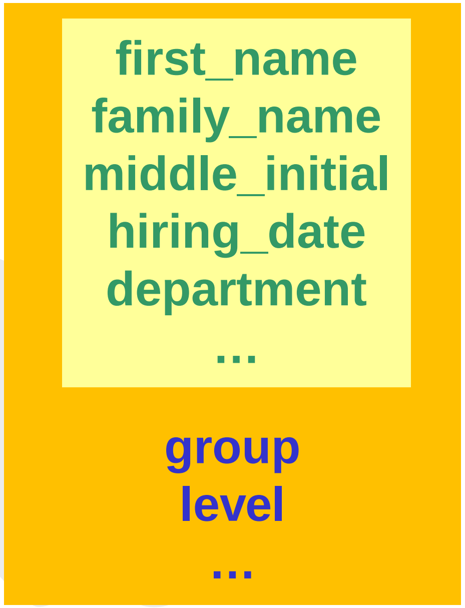
- Method 2

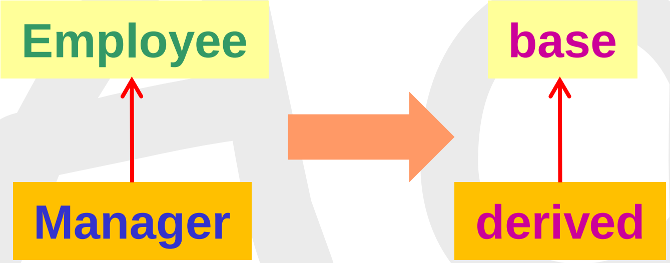
```
struct Manager : public Employee {  // public inheritance
        Employee* group[100];  // people managed
        short level;
        // …
};
```

```
  Manager m;
  Employee *pe = &m;  // "OK," compiler says,
                                   // "I DO know a manager is an employee since
                                   // you've already told me by public inheritance
```

Public Inheritance and Data Members

- Public inheritance models "**is-a**" relationship
  - e.g., a Manager **is an** Employee
  - a derived class inherits ALL data members of its base class


first_name
family_name
middle_initial
hiring_date
department
…
group
level
…


Employee
Manager
base
derived

```
void f(Manager mm, Employee ee) {
    Employee* pe = &mm;      // ok, every manager is an Employee
    pe->first_name = "Adar"; // ok, a manager also has first name
    Manager* pm = &ee;       // error, not every Employee is a Manager
    pm->level = 3;           // disaster!  ee doesn't have a level
```

Access Controls (1/2)

- A member of a publicly derived class can use public and protected members (discuss later) of its base class
  - as if they were declared in the derived class
  - a derived class can **NOT** access **private** members of its base class

```
~~struct~~ class Employee {
    string first_name, family_name;
    char middle_initial;
    // same data members
public:
    void print() const;
    string full_name() const {
        return first_name + ‘ ’ +
        middle_initial + ‘. ’ + family_name ;   }
        // …
};
```

```
class Manager : public Employee {
    Employee* group[100];
    short level;
    // same data members
public:
    void print() const;
    // …
};
```

Access Controls (2/2)

- Remind you again, same access control rules still apply

```
void Manager::print() const {        // First version
    cout << “Name is ” << full_name() << endl;
    // ok! Though Manager does not define its own full_name(),
    // it inherits and uses Employee’s public member function full_name()
    // …
}

void Manager::print() const {        // Second version
    cout << “Name is ” << family_name << endl;
    // error! Though Manager has (inherits) data member family_name,
    // family_name cannot be directly accessed
    // since it is Employee’s private member
    // …
}
```

Think about why?

Constructors in Derived Classes (1/2)

- Ctor of derived class is responsible to call ctors for its **base classes** (and its own **non-static class data members**)

```
class Employee {
    // data members
public:
    Employee(const string& s, int d);  // Employee’s ctor needs arguments
    // …
};

class Manager : public Employee {
    // data members
public:
    Manager(const string& s, int d, int lvl);  // Manager’s ctor
    // …
};
```

Constructors in Derived Classes (2/2)

```
Employee::Employee(const string& s, int d)
  : family_name(s), department(d)  // initialize non-static data members
{  // …   }

Manager::Manager(string& s, int d, int lvl) // correct ctor version
  : Employee(s, d), level(lvl)   // initialize base and non-static  data members
{   // …  }

Manager::Manager(string& s, int d, int lvl)  // incorrect version
  : family_name(s), department(d), level(lvl)  // private members of Employee
{   // …  }
```

- Initialize base and non-static data members using their corresponding ctors

Order of ctors and dtors

- Class object is constructed in the bottom-up fashion
  - first: base classes in declaration order
  - second: non-static data members in declaration order
  - third: derived class itself

- Class object is destroyed in reverse order of construction

More about Constructors

- For a derived class with base and class data members
  - ctor of derived class **MUST** invoke one ctor for each of its base and non-static class data members
  - if ctor of derived class does not explicitly invoke a base's or member's ctor
    ➔ that base's or member's **default ctor** is implicitly called

- If there are no ctors defined in a derived class
  - compiler will try to generate a public one automatically
  - the behavior of the generated ctor is to invoke default ctors for all base and non-static class data members

- Namely, **ctors are never inherited**

More about Destructors

- For a derived class with base and class data members
  - dtor of derived class **will automatically** invoke dtor for each of its base and non-static class data members
- If there is no dtor defined in a derived class
  - compiler will try to generate a public one automatically
  - the behavior of the generated dtor is to invoke dtors for all base and non-static class data members
- Similarly, **dtors are never inherited**
- If what compiler-generated dtor does is not what you want ➔ define your own

Example: Put Them Together

```
struct A {
    A() { cout << “ctor A” << endl; }
    ~A() { cout << “dtor A” << endl; }
};
struct B {
    B() { cout << “ctor B” << endl; }
    ~B() { cout << “dtor B” << endl; }
};
struct C : public B {
    A a;
    C() { cout << “ctor C” << endl; }
    ~C() { cout << “dtor C” << endl; }
};
int main() {
    C c;
    return 0;
}
```

```
Output:
========
ctor B
ctor A
ctor C
dtor C
dtor A
dtor B
```

Copy ctors and Assignment Operators

- If you don’t define copy ctor and copy assignment operator for a derived class
  - again, compiler will try to generate public ones for that derived class
- Behavior of compiler generated copy ctor
  - first calls its base’s copy ctor
  - then calls copy ctors for non-static class data members
- Behavior of compiler generated assignment operator
  - first calls its base’s assignment operator
  - then calls assignment operator for non-static class data members
- Similarly, **copy ctors and copy assignment operators are never inherited**
- If what compiler-generated versions do is not what you want ➔ define your own

Case Study (1/2)

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

struct A {
    int d;
    A(int n=0) : d(n) { }
    A(const A& a) : d(a.d) { }              // copy ctor
    A& operator=(const A& a) { d = a.d; return *this; }
};

struct B {
   int d;
   B(int n=0) : d(n) { }
    B(const B& b) : d(b.d) { }
    B& operator=(const B& b) { d = b.d; return *this; }
    virtual ~B() { } // discuss in the next chapter
};
```

Case Study (2/2)

```
struct C : public B {
    A a; int d; int* pi;
    C(int n1=0, int n2=0, int n3 = 0) : B(n1), a(n2), d(n3) {
            pi = new int[10]; for(int i=0; i<10; ++i) pi[i] = i; }
    C(const C& c) : B(c), a(c.a), d(c.d)  { // c is also of type B
            pi = new int[10]; for(int i=0; i<10; ++i) pi[i] = c.pi[i]; }
    C& operator=(const C& c) {
            B::operator=(c); a = c.a; d = c.d; int *tmp = new int[10];
            for(int i=0; i < 10; ++i) tmp[i] = c.pi[i];
            delete[] pi; pi = tmp; return *this; }
    ~C() { delete[] pi; }
};

int main() {
   C c1(10, 100, 1000), c2(c1), c3; // c2 uses copy ctor; c3 uses default ctor
   c3 = c2;    // c3 uses copy assignment operator
   cout << c3.B::d << '\t' << c3.a.d << '\t' << c3.d << endl;
   return 0; }
```

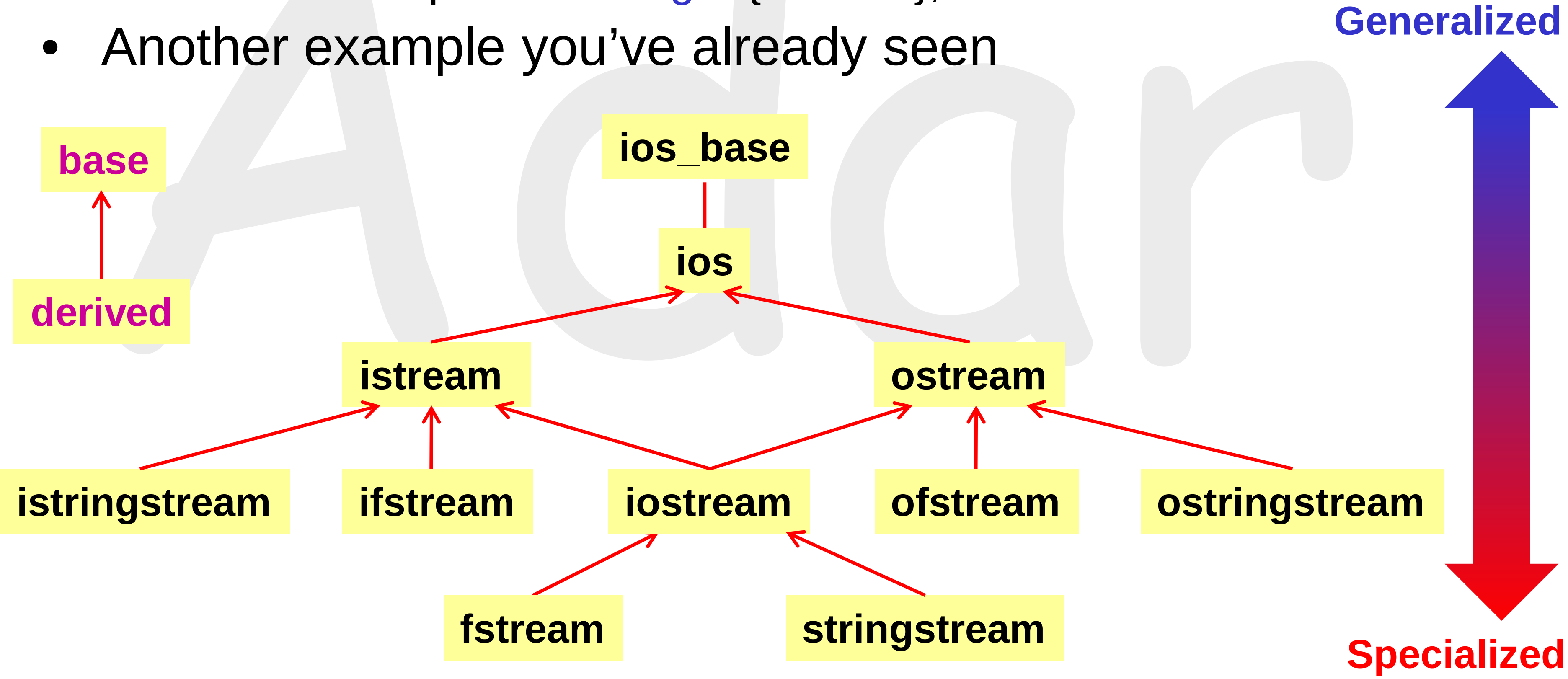
Juinn-Dar Huang jdhuang@mail.nctu.edu.tw

Class Hierarchy

- A derived class can itself be a base class

```
class Employee { /* … */ };
class Manager : public Employee { /* … */ };
class Director : public Manager { /* … */ };
```

- Another example you've already seen



Protected Members (1/2)

Advanced

- A member of a class can be private, protected, or public
  - apply to both data members and member functions
- Private members (data and functions)
  - its name can be used only by member functions and friends of the class in which it is declared
- Public members (data and functions)
  - its name can be used by any functions
- Protected members (data and functions)
  - its name can be used only by member functions and friends of the class in which it is declared, and
  - by member functions and friends of classes derived from this class

Protected Members (2/2)

Advanced

```
Class B {
    int b_priv;
protected:
    void b_prot();
public:
    void b_pub();
};

class D : public B {
public:
    void d_func();
};
```

```
void D::d_func() {  // D's member function
    b_priv = 1;     // error, B's private member
    b_prot ();      // ok, D is B's child
    b_func ();      // ok, can do this in any function
    // …
}

void func(B& b) {  // a global function
    b.b_priv = 1; // error, B's private member
    b.b_prot();     // error, B's protected member
    b.b_func();     // ok, can do this in any function
    // …
}
```

It's usually not a good idea to have protected data members
However, it's sometimes useful to have protected member functions

Access Controls to Base Classes (1/2)

Advanced

- Like a member, a base class can be declared private, protected, or public

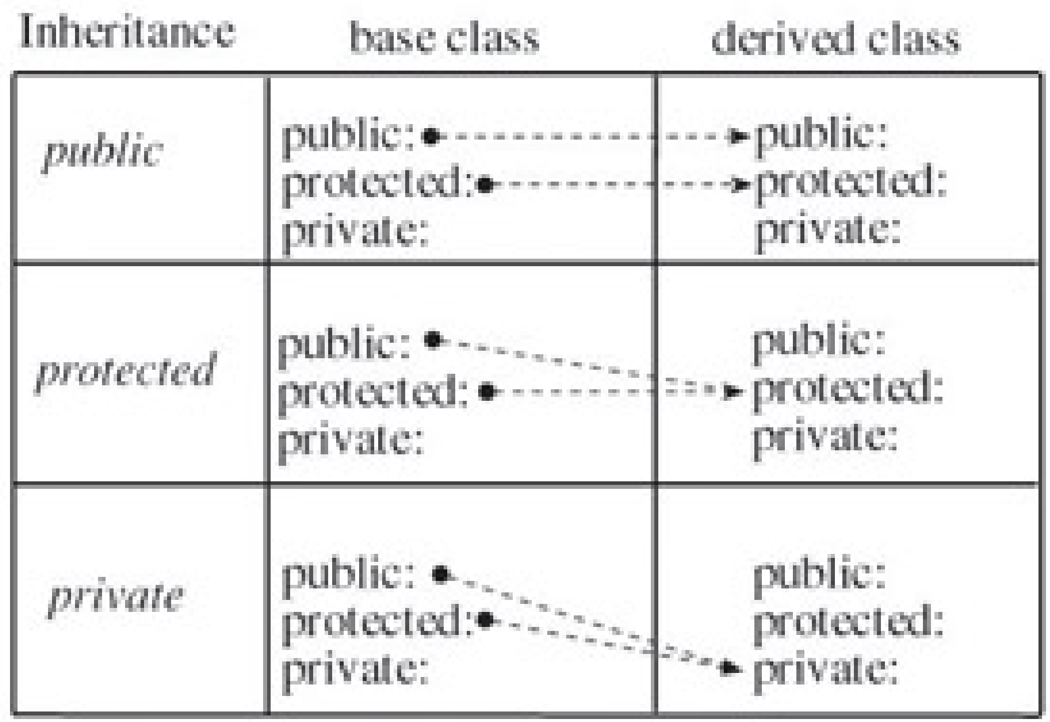
```
class X : public B { / * … */  };           // public inheritance
class Y : protected B { / * … */  };        // protected inheritance
class Z : private B { /* … */  };           // private inheritance
```

- Remind that public inheritance models “is-a” relationship
  - this is the most common form of inheritance
- Protected inheritance and private inheritance model “is-implemented-in-terms-of” relationship
  - beyond the scope of this course

Access Controls to Base Classes (2/2)

Advanced

Member in base class	Type of Inheritance		
	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	no access	no access	no access



Inheritance	base class	derived class
public	public: protected: private:	public: protected: private:
protected	public: protected: private:	public: protected: private:
private	public: protected: private:	public: protected: private:

Is-a vs. Has-a

- “Is-a” relationship is modeled by public inheritance

```
class Manager : public Employee { /* … */  };
// It says a Manager is an Empoyee
```

- “Has-a” relationship is modeled through composition
  - also called layering

```
class Employee {
        string first_name, family_name;
        // …
};
// It says every Employee has a first_name and a family_name
```

Multiple Inheritance (1/2)

Advanced

- A derived class has only one direct base class
  - single inheritance
- In C++, a class can have more than one direct base class
  - multiple inheritance

    class iostream : public istream, public ostream { /* … */ };

- Multiple inheritance raises a bunch of issues
  - just name a few …

```
struct A { int d; void func(); };
struct B { int d; void func(); };
struct C : public A, public B { };
void f() {
        C c;
        c.d = 10;            // error! ambiguous! whose d? A’s or B’s ?
        c.func();            // error! ambiguous! whose func()? A’s or B’s ?
}
```

Multiple Inheritance (2/2)

Advanced

- All the above issues can be resolved, but …
- Trust me, you don't want to know the details now …
  - it's an advance topic and thus beyond the scope of this course
  - you should not attempt using multiple inheritance until you are an experienced C++ programmer
- Use multiple inheritance judiciously
  - e.g., Java only supports single inheritance

Summary (1/2)

- Class represents concept and inheritance represents relationships among classes (concepts)
- Inheritance
  - base class vs. derived class
  - rules about how derived class inherits data members and member functions from base class (and exceptions)
  - rules for access controls
  - ctor/dtor/copy ctor/copy assignment operator in derived class
- Protected members
- Public/protected/private inheritance
- Single inheritance vs. multiple inheritance

Summary (2/2)

- Public inheritance models “is-a” relationship
  - most commonly used inheritance
- Model “has-a” relationship through composition
  - very commonly used technique
- Protected/private inheritance models “is-implemented-in-terms-of” relationship
- Avoid using protected data members
- Use protected member functions judiciously
- Use multiple inheritance judiciously