LAB8
INHERITANCE

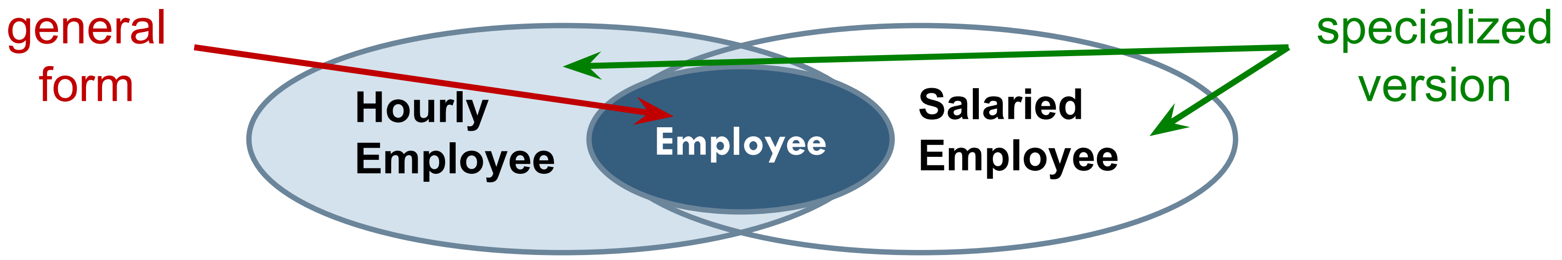
Department of Electronics Engineering
National Chiao Tung University

Outline

- Inheritance basics
  - Derived classes
  - Protected: qualifier
  - Redefining member functions
- Programming with Inheritance
  - Assignment operators and copy constructors
  - Destructors in derived classes
- Exercise

Derived Classes (1/3)

- What's Inheritance
  - Purpose: Object-Oriented Programming
  - Provides abstraction dimension:
    - Defines general form of class, then specialized versions inherit properties of general class



    - Add new/modify base's functionality for it's appropriate use

Derived Classes (2/3)

- Base class
  - "General" class from which others derive
- Derived class
  - New class
  - Automatically has base class's:
    - Member variables
    - Member functions
  - Can then add additional member functions and variables

Derived Classes (3/3)

- *Example*:
  - Base Class
    - All employees have a name, SSN and net pay
  - An Employee may be…
    - **SalariedEmployee**
    - **HourlyEmployee**
  - Each is "subset" of employees

Base Class: Employee

- Class definition is just like what we done before
  - *Example:*

```
class Employee {
string name;
string ssn;
double netPay;
    public:
Employee( );
Employee(string theName, string theSSN);
string getName( ) const;
string getSSN( ) const;
double getNetPay( ) const;
// …
    };
```

Derived Classes: HourlyEmployee

- Derived class interface only lists new or "to be redefined" members
  - *Example:*

```
class HourlyEmployee: public Employee {   // public inheritance
    double wageRate;
double hours;
     // …
};
```

- All the variables (name, ssn, netPay …) which belong to Employee, the HourlyEmployee also has them

Pointer's Conversion

- Because a HourlyEmployee is also an Employee, we can make the Employee's pointer point to the HourlyEmployee
  - *Example*: (upcasting)

```
HourlyEmployee h;
Employee *pe= &h;            // "OK," public inheritance
```

- But the inverse conversion (downcasting) will cause a damage (An Employee is not a HourlyEmployee certainly)

Access Controls

- The same access control rules still apply in a inheritance relationship
  - Generally, data member won't be public (packaging)
  - Use interface to access these data members
    - *Example*:

```
void HourlyEmployee::print() const {
cout<< "Name is " << getName( ) << endl;
        // OK, if getName( )  is a public member (interface)
}
void HourlyEmployee ::print() const {
cout<< "Name is " << name<< endl;
        // ERROR, if name is a private member
}
```

Belongs to Employee::

Belongs to Employee::

Constructor

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

Belongs to HourlyEmployee::

```
HourlyEmployee::HourlyEmployee(string theName, string theSSN, double wr)
: Employee(theName, theSsn), wageRate(wr) {// … }
              // initialize base and non-static data members
```

Belongs to Employee::

- Initialize base and non-static data members using their corresponding ctors
  - Default ctor for base/derived should be well defined

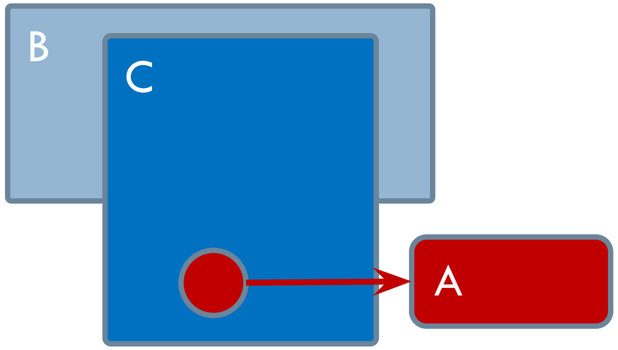
Execution Order of Constructor

```
struct A {
   A() { cout<< "ctorA" << endl; }
   ~A() { cout<< "dtorA" << endl; } };

struct B {
   B() { cout<< "ctorB" << endl; }
   ~B() { cout<< "dtorB" << endl; } };

struct C : public B {
   A a;
   C()::B(), a() { cout<< "ctorC" << endl; }
   ~C() { cout<< "dtorC" << endl; } };

int main() {
   C c;
   return 0; }
```



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

Copy Ctor & Assignment Operator

- Copy ctors and copy assignment operators are never inherited (should be rewrited)
  - *Example:*

```
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; }
};
```

Call B's constructor

Rewrite copy ctor

Rewrite Assignment operater

Protected Members

- Protected members (data and functions)
  - Its name can be used by member functions and friends of the class only in which it is declared, and by member functions and friends of classes derived from this class

```
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 is derived from B
    b_priv= 1;         // error
    b_prot();          // ok
    b_pub();    // ok
    // … }

void func(B& b) {  // a global function
    b.b_priv= 1;      // error
    b.b_prot();       // error
    b.b_pub();// ok
    // … }
```

Different Kind of Inheritance

- Like a member, a base class can be declared private, protected, or public
  - class X : public B { / * … */ };   // public inheritance
    - Public inheritance models is "is-a" relationship
  - class Y : protected B { / * … */ };   // protected inheritance
  - class Z : private B { /* … */ };   // private inheritance
    - both model are "is-implemented-in-terms-of" relationship

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

Is-a vs. Has-a

- "Is-a" relationship is modeled by public inheritance
  - class HourlyEmployee : public Employee { /* … */ };
    - It says a HourlyEmployee **is an** Empoyee

- "Has-a" relationship is modeled through composition
  - Also called layering
  - class Employee {
    string name;
    string ssn;
     // … };
    - It says every Employee **has a** name and a ssn

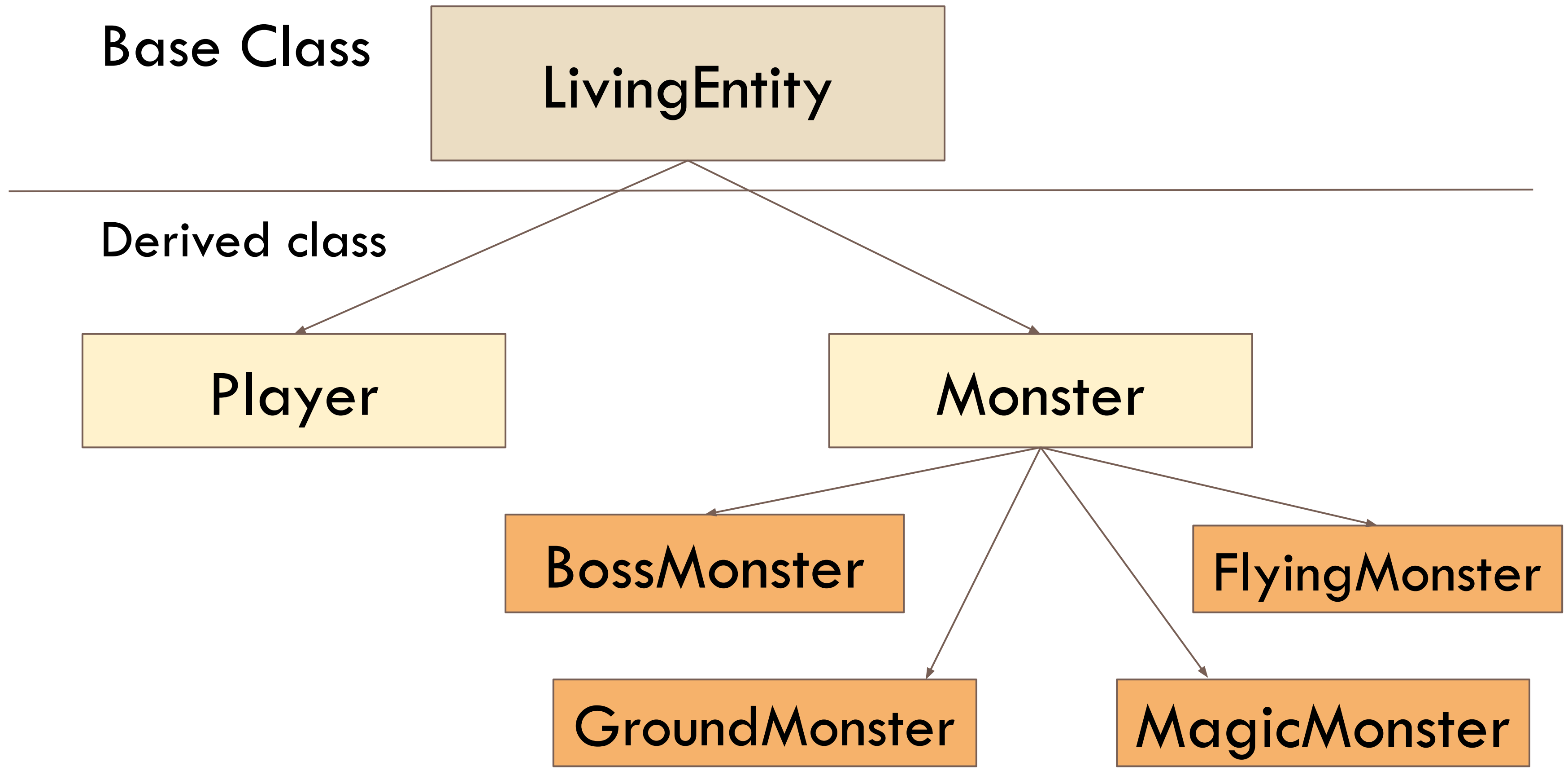
Lab8 Exercise -battle system

- do the implementation of various classes for players and monsters, simulate battles between them, and track player progression throughout the game.
- player can gain experience after they beat monsters
- if they get enough experience they can level up, after they level up they get more attackdamage and health.

You should….

1. complete .cpp file
2. you cannot change main.cpp and .h file
3. completely same as TA’s output
4. see detail by main.cpp

Inheritance

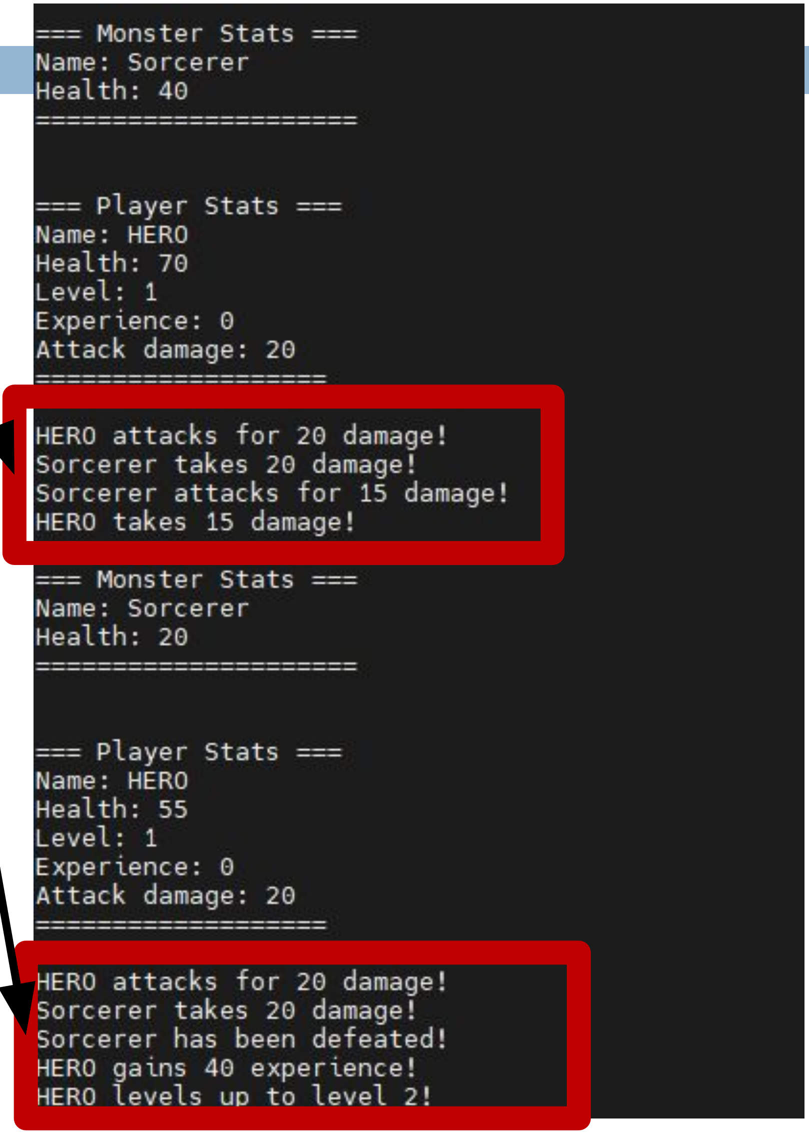


BattleEvent

- class BattleEvent :use member function for other classes to complete the battle.
- if one of the characters dies, end up the battle
- When the battle starts, the player attacks the monster first, if the monster doesn’t die ,it attacks the player, so on…

here is a part of the battle

you only need to write cout



```
=== Monster Stats ===
Name: Sorcerer
Health: 40
=====================


=== Player Stats ===
Name: HERO
Health: 70
Level: 1
Experience: 0
Attack damage: 20
===================
HERO attacks for 20 damage!
Sorcerer takes 20 damage!
Sorcerer attacks for 15 damage!
HERO takes 15 damage!
=== Monster Stats ===
Name: Sorcerer
Health: 20
=====================


=== Player Stats ===
Name: HERO
Health: 55
Level: 1
Experience: 0
Attack damage: 20
===================
HERO attacks for 20 damage!
Sorcerer takes 20 damage!
Sorcerer has been defeated!
HERO gains 40 experience!
HERO levels up to level 2!
```

class LivingEntity

- Attributes: health ,damage,name.
- Methods:
  - **bool takeDamage(int)**:reduce health by enemy's damage ;return 0 if someone died ,else return 1.
    health must be >= 0
  - **bool isAlive()**: Checks if the entity is still alive
  - **getHealth()**, **getName()**: Accessors for player information
  - print()

class player

- Attributes: Experience(initial to 0), Level(initial to 1).
- Methods:
  - **attack(monster)**: use damage to attack monsters, if defeating it ,you get the experience = original health of monster *0.5
  - **gainExperience(int)**: Increases player experience points,if your experience >=level*level*20 -> level up.
    Don’t need to initial the experience
  - **levelUp()**: Levels up the player, increasing health = health+level*20 and damage=level*20.
  - **restorehealth(int)**: Restores entity's health by a specified amount.
  - **getName()**, **getLevel()**, **getExperience()**: Accessors for player information
  - print()

class monster & other monster class

- Attributes: getexperience(=original health *0.5)
- Methods:
    - **attack(player)**: use damage to attack players
    - **getexperience(int)**: Accessors for getexperience(player can get this experience when they defeat the monster)
    - print()

you only need to write constructors for derived class from monster class

:BossMonster,GroundMonster,FlyingMonster,MagicMonster

test by inputs

input1.txt : test normal attack by monster or player ,get experience,restore health,you win,or you lose.

input2.txt : test input1 + level up

input3.txt : test battleevent with one monster.

input4.txt : test battleevent with several monsters.

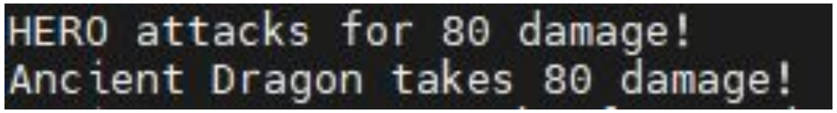
you can check main.cpp for detail

Output Requirement

- every time use attack function you should output :

  "(nameA) attack for (damage) damage!"

  "(nameB) takes (damage) damage!"

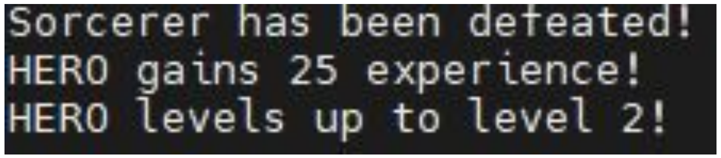


```
HERO attacks for 80 damage!
Ancient Dragon takes 80 damage!
```

- if you beat monster and get experience output :

  "(nameA) has been defeated! "

  "(nameA) gains (experience) experience!"



```
Sorcerer has been defeated!
HERO gains 25 experience!
HERO levels up to level 2!
```

- if you level up output :

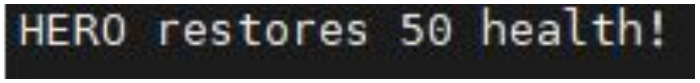
  "(nameA) levels up to level (level)! "

- if you dies output :

  "(nameA) has been defeated! "

- if you restore health :

  "(nameA) restores (amount) health!"



```
HERO restores 50 health!
```

Submission

- Compile
  - g++ -std=c++11 main.cpp LivingEntity.cpp Monster.cpp BattleEvent.cpp Player.cpp -o Lab08
- Run
  - ./Lab08 [input filename]

- OJ
  /home/share/demo_OOP112_2 Lab 08

- Ask TA for Demo