

# IM 1003: Programming Design

## Polymorphism

Ling-Chieh Kung

Department of Information Management  
National Taiwan University

May 19, 2014

# Outline

- **Motivation**
- Basic ideas and the first example
- Virtual functions

# Class Character

- Recall our class **Character** for our RPG (role-play game):

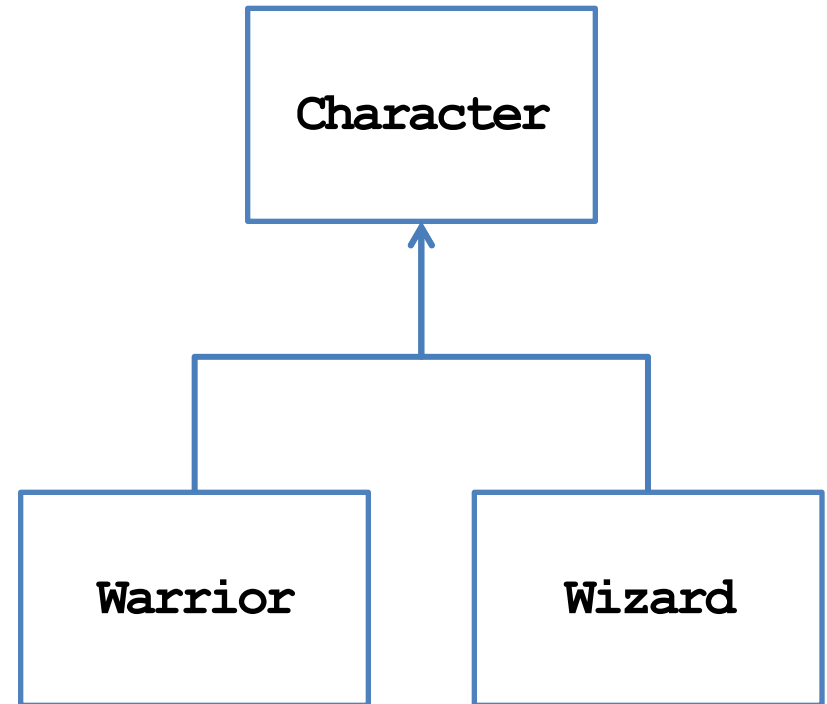
```
class Character
{
protected:
    string name;
    int level;
    int exp;
    int power;
    int knowledge;
    int luck;
    static const int expForLevel = 100;
    void levelUp(int pInc, int kInc, int lInc);
public:
    Character(string n, int lv, int po, int kn, int lu);
    void beatMonster(int exp);
    void print();
    string getName();
};
```

# Class Character

```
Character::Character(string n, int lv, int po, int kn, int lu)
    : name(n), level(lv), exp(pow(lv - 1, 2) * expForLevel), power(po), knowledge(kn), luck(lu) {}
void Character::beatMonster(int exp) {
    this->exp += exp;
    if(this->exp >= pow(this->level, 2) * expForLevel)
        this->levelUp(0, 0, 0); // No improvement when advancing to the next level
}
void Character::print() {
    cout << this->name
         << ": Level " << this->level << " (" << this->exp << "/" << pow(this->level, 2) * expForLevel
         << "), " << this->power << "-" << this->knowledge << "-" << this->luck << "\n";
}
void Character::levelUp(int pInc, int kInc, int lInc) {
    this->level++; this->power += pInc; this->knowledge += kInc; this->luck += lInc;
}
string Character::getName() {
    return this->name;
}
```

# Character, Warrior, and Wizard

- **Character** should **not** be used to create an object.
  - No improvement when advancing to the next level.
  - Personal attributes for improvements per level are not defined.
- We define two derived classes **Warrior** and **Wizard**:
  - **Character** is an **abstract class**.
  - **Warrior** and **Wizard** are **concrete classes**.



# Classes Warrior and Wizard

```
class Warrior : public Character
{
private:
    static const int powerPerLevel = 10;
    static const int knowledgePerLevel = 5;
    static const int luckPerLevel = 5;
public:
    Warrior(string n) : Character(n, 1, powerPerLevel, knowledgePerLevel, luckPerLevel) {}
    Warrior(string n, int lv)
        : Character(n, lv, lv * powerPerLevel, lv * knowledgePerLevel, lv * luckPerLevel) {}
    void print() { cout << "Warrior "; Character::print(); }
    void beatMonster(int exp)
    {
        this->exp += exp;
        if(this->exp >= pow(this->level, 2) * expForLevel)
            this->levelUp(powerPerLevel, knowledgePerLevel, luckPerLevel);
    }
};
```

# Classes Warrior and Wizard

```
class Wizard : public Character
{
private:
    static const int powerPerLevel = 4;
    static const int knowledgePerLevel = 9;
    static const int luckPerLevel = 7;
public:
    Wizard(string n) : Character(n, 1, powerPerLevel, knowledgePerLevel, luckPerLevel) {}
    Wizard(string n, int lv)
        : Character(n, lv, lv * powerPerLevel, lv * knowledgePerLevel, lv * luckPerLevel) {}
    void print() { cout << "Wizard "; Character::print(); }
    void beatMonster(int exp)
    {
        this->exp += exp;
        if(this->exp >= pow(this->level, 2) * expForLevel)
            this->levelUp(powerPerLevel, knowledgePerLevel, luckPerLevel);
    }
};
```

# Some questions

- We may create **Warrior** and **Wizard** objects in our program.
  - May we **prevent** one from creating a **Character** object?
- A team has at most ten members.
  - We create two arrays, one for warriors and one for wizards. Each of them has a length of 10.
  - Why **waste spaces**?

```
class Team
{
private:
    int warriorCount;
    int wizardCount;
    Warrior* warrior[10];
    Wizard* wizard[10];
public:
    Team();
    ~Team();
    void addWarrior(string name, int lv);
    void addWizard(string name, int lv);
    void warriorBeatMonster(string name, int exp);
    void wizardBeatMonster(string name, int exp);
    void printWarrior(string name);
    void printWizard(string name);
};
```



# Some questions

- We may need to add a warrior/wizard, let a warrior/wizard beat a monster, and print the current status of a warrior/wizard.
  - Their names are all different.
- Either we write two functions for a task, or write just one.
  - Two: tedious and inconsistent.
  - One: Inefficient.

```
class Team
{
private:
    int warriorCount;
    int wizardCount;
    Warrior* warrior[10];
    Wizard* wizard[10];
public:
    Team();
    ~Team();
    void addWarrior(string name, int lv);
    void addWizard(string name, int lv);
    void warriorBeatMonster(string name, int exp);
    void wizardBeatMonster(string name, int exp);
    void printWarrior(string name);
    void printWizard(string name);
};
```

# Outline

- Motivation
- **Basic ideas and the first example**
- Virtual functions

# Polymorphism

- The key flaw is to create two arrays, one for warriors and one for wizard.
  - May we use **only one array** to store the ten members?
  - But **Warrior** and **Wizard** are different classes.
- While they are different classes, they have **the same base class**.
  - They are all **Characters!**
  - May we declare a **Character** array to store **Warrior** and **Wizard** objects?
- We can. This is called **polymorphism**.
  - In C++, the way we implement polymorphism is to  
*“Use a variable of a parent type to store a value of a child type.”*

# Variables vs. values

- Let's differentiate a **variable's type** and a **value's type**.
- A variable can store values and must have a type.
  - E.g., a **double** variable is a **container** which “should” store a **double** value.
- A value is the thing that is stored in a variable.
  - E.g., **12.5** or **7**.
- A value has its own type, which may be **different** from the variable's type.
- In C++, a **parent variable** can store a **child object**.
  - A **Character** variable can store a **Warrior** or a **Wizard** object.
  - Because a warrior/wizard is a character!

# Examples of polymorphism

- For example, we may do this:

```
int main
{
    Warrior w("Alice", 10);
    Character c = w; // copy constructor
    cout << c.getName() << endl; // Alice
    return 0;
}
```

- Or we may do this with pointers:

```
int main
{
    Warrior w("Alice", 10);
    Character* c = &w;
    cout << c->getName() << endl; // Alice
    return 0;
}
```

# Polymorphism with arrays

- Polymorphism is useful typically with **functions** or **arrays**:

```
int main
{
    Character* c[3];
    c[0] = new Warrior("Alice", 10);
    c[1] = new Wizard("Sophie", 8);
    c[2] = new Warrior("Amy", 12);
    for(int i = 0; i < 3; i++)
        c[i]->print();
    for(int i = 0; i < 3; i++)
        delete c[i];
    // do not delete [] c;
    return 0;
}
```

```
int main
{
    Character c[3]; // error! Why?
    Warrior w1("Alice", 10);
    Wizard w2("Sophie", 8);
    Warrior w3("Amy", 12);
    c[0] = w1;
    c[1] = w2;
    c[2] = w3;
    for(int i = 0; i < 3; i++)
        c[i].print();
    return 0;
}
```

# Class Team with Polymorphism

- With polymorphism, we may redefine the class **Team**:

```
class Team
{
private:
    int warriorCount;
    int wizardCount;
    Warrior* warrior[10];
    Wizard* wizard[10];
public:
    Team();
    ~Team();
    void addWarrior(string name, int lv);
    void addWizard(string name, int lv);
    void warriorBeatMonster(string name, int exp);
    void wizardBeatMonster(string name, int exp);
    void printWarrior(string name);
    void printWizard(string name);
};
```

```
class Team
{
private:
    int memberCount;
    Character* member[10];
public:
    Team();
    ~Team();
    void addMember
        (string name, int lv, char occupation);
    void memberBeatMonster(string name, int exp);
    void printMember(string name);
};
```

# Class Team with Polymorphism

- With polymorphism, we may redefine the class **Team**:

```
Team::Team()
{
    this->memberCount = 0;
    for(int i = 0; i < 10; i++)
        member[i] = NULL;
}
Team::~Team()
{
    for(int i = 0;
        i < this->memberCount;
        i++)
        delete this->member[i];
}
```

```
void Team::addMember
(string name, int lv, char occupation)
{
    if(this->memberCount < 10)
    {
        if(occupation == 'R')
            this->member[this->memberCount] = new Warrior(name, lv);
        else if(occupation == 'D')
            this->member[this->memberCount] = new Wizard(name, lv);
        this->memberCount++;
    }
}
```



# Class Team with Polymorphism

- With polymorphism, we may redefine the class **Team**:

```
void Team::memberBeatMonster(string name, int exp)
{
    for(int i = 0; i < this->memberCount; i++)
    {
        if(this->member[i]->getName() == name)
        {
            this->member[i]->beatMonster(exp);
            break;
        }
    }
}
```

```
void Team::printMember(string name)
{
    for(int i = 0; i < this->memberCount; i++)
    {
        if(this->member[i]->getName() == name)
        {
            this->member[i]->print();
            break;
        }
    }
}
```

# Remaining questions

- We still cannot prevent one from creating a **Character** object.
- What happens to the following program:
  - No “Warrior ” and “Wizard ” printed out.
  - No experience point accumulated.
- Why?

```
int main
{
    Character* c[3];
    for(int i = 0; i < 3; i++)
        c[i]->print();
    c[0] = new Warrior("Alice", 10);
    c[1] = new Wizard("Sophie", 8);
    c[2] = new Warrior("Amy", 12);
    c[0]->beatMonstor(10000);
    for(int i = 0; i < 3; i++)
        c[i]->print();
    for(int i = 0; i < 3; i++)
        delete c[i];
    return 0;
}
```

# Invoking an overridden function

- Suppose a parent variable stores a child value (or a parent pointer pointing to a child object).
- If we use the parent variable (pointer) to invoke an overridden function, which implementation will be invoked?
- The default setting is to invoke the parent's implementation.
- To invoke the child's one, we need **virtual functions**.

```
class A
{
public:
    void a() { cout << "a\n"; }
    void f() { cout << "af\n"; }
};

class B : public A
{
public:
    void b() { cout << "b\n"; }
    void f() { cout << "bf\n"; }
};
```

```
int main()
{
    B b;
    A a = b;
    a.a();
    a.f();
    // a.b();
    return 0;
}
```

```
int main()
{
    B b;
    A* a = &b;
    a->a();
    a->f();
    // a->b();
    return 0;
}
```

# Outline

- Motivation
- Basic ideas and the first example
- **Virtual functions**

# Early binding vs. late binding

- When we do **A a = b** or **A\* a = &b**, we are using polymorphism.
- For **A a = b**, the system does **early binding**:
  - **a** occupies only four bytes for storing **i**.
  - **a** does not have a space for storing **j**.
  - Its type is determined to be **A** at **compilation**.
- For **A\* a = &b**, the system does **late binding**:
  - **a** is just a pointer.
  - It can point to an **A** object or a **B** object.
  - Its type can be determined at the **run time**.

```
class A
{
private:
    int i;
public:
    void a() { cout << "a\n"; }
    void f() { cout << "af\n"; }
};

class B : public A
{
private:
    int j;
public:
    void b() { cout << "b\n"; }
    void f() { cout << "bf\n"; }
};
```

# Early binding vs. late binding

- But we still see the parent's implementation being invoked. Why?

```
int main()
{
    A a;
    B b;
    A* who = &a;
    who->f(); // af
    who = &b;
    who->f(); // af

    return 0;
}
```

- To ask the system invoke the child's implementation, declare **virtual functions**.

# Virtual functions

- If we declare a parent's member function to be **virtual**, its invocation priority will be lower than a child's (if we use late binding).
  - To do so, simply add **the modifier virtual** into the function header:
  - The child's implementation is invoked!
- No need to do that at the child's side.
  - A parent can declare its function as a virtual function.
  - A child cannot declare a parent's function as virtual (it is of no use).
- In summary, we need:
  - Late binding + virtual functions.

```
class A
{
private:
    int i;
public:
    void a() { cout << "a\n"; }
    virtual void f() { cout << "af\n"; }
};
```

# Virtual functions

- For our **Character** class, simply declare **beatMonster()** and **print()** as virtual.

```
class Character
{
protected:
    // ...
public:
    Character(string n, int lv, int po, int kn, int lu);
    virtual void beatMonster(int exp);
    virtual void print();
    string getName();
};
```

- Warrior** and **Wizard** override the two functions. Now their implementations get invoked.

```
int main
{
    Character* c[3];
    for(int i = 0; i < 3; i++)
        c[i]->print();
    c[0] = new Warrior("Alice", 10);
    c[1] = new Wizard("Sophie", 8);
    c[2] = new Warrior("Amy", 12);
    c[0]->beatMonstor(10000);
    for(int i = 0; i < 3; i++)
        c[i]->print();
    for(int i = 0; i < 3; i++)
        delete c[i];
    return 0;
}
```



# Abstract classes

- The two virtual functions are different in their natures:
  - `print()` is invoked in the children's implementations.
  - `beatMonster()` is not. In fact, it should not be invoked by any one.
- We may set `beatMonster()` to be a **pure virtual function**:

```
class Character
{
    // ...
    virtual void beatMonster(int exp) = 0;
};
```

- Now we do not need to implement it.
- Moreover, we **cannot** create **Character** objects!

# Late binding is required

- Even if we declare virtual functions, they do not work in the following program:

```
int main
{
    Character c[3]; // Suppose we add a default constructor
    Warrior w1("Alice", 10);
    Wizard w2("Sophie", 8);
    Warrior w3("Amy", 12);
    c[0] = w1;
    c[1] = w2;
    c[2] = w3;
    for(int i = 0; i < 3; i++)
        c[i].print();
    return 0;
}
```

- Late binding (by using pointers) is required.

# Polymorphism is everywhere

- Recall **MyVector**, its overloaded operator `==`, and its child **MyVector2D**.

```
class MyVector
{
    // ...
private:
    int n;
    double* m;
public:
    // ...
    bool operator==(const MyVector& v) const;
};
```

```
int main()
{
    double d[3] = {1, 2, 3};
    MyVector v1(3, d);
    MyVector2D v2(4, 5);
    cout << v1 == v2 << endl; // allowed?
    return 0;
}
```

- Why can the program run?
- In fact, we may also compare **MyVector2D** with **MyVector**, **MyVector2D** with **MyVector2D**, **NNVector** with **MyVector**, **NNVector** with **MyVector2D**, etc.

# Summary

- Polymorphism is a technique to make our program clearer, more flexible and more powerful.
  - It is based on **inheritance**.
  - It is tightly related to **function overriding**, **late binding**, and **virtual functions**.
- The key action is to “use a parent pointer to point to a child object”.
- To implement late binding, you need to
  - Declare and override virtual functions.
  - Do late binding by using parent pointers to point to child objects.