Programming Design

Inheritance and Polymorphism

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

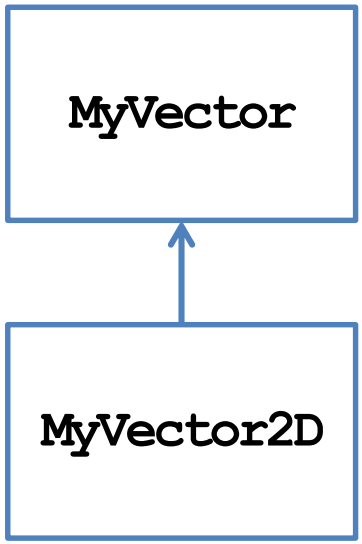
- **Inheritance**
- An example
- Polymorphism

Inheritance

- The three main characteristic/functionalities of OOP:
  - Encapsulation: packaging + data hiding.
  - **Inheritance**.
  - **Polymorphism**.
- Through inheritance, we may **create new classes from existing classes**.
  - A **derived** (**child**) class inherits a **base** (**parent**) class.
  - A child class has (some) members defined in the parent class.
- This is particularly useful when "**XXX is a OOO**".
  - An apple is a fruit.
  - A circle is a shape.
  - A truck is a vehicle.

The first example

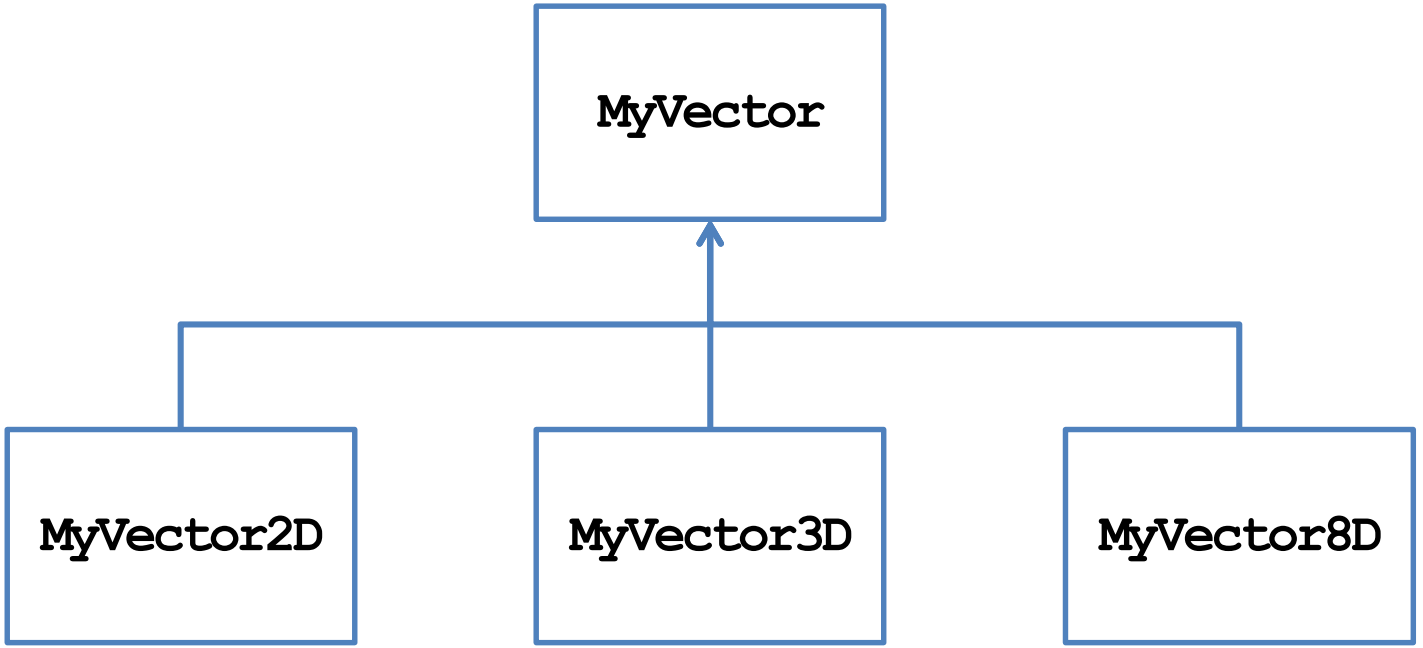
- Recall that we have defined **`MyVector`**.
- A two-dimensional (2D) vector is a vector!
- Let's create a class for 2D vector by inheritance.




```
class MyVector
{
protected: // to be explained
  int n;
  double* m;
public:
  MyVector();
  MyVector(int n, double m[]);
  MyVector(const MyVector& v);
  ~MyVector()
  void print() const;
  // ==, !=, <, [], =, +=
};
```

Brothers and sisters

- One parent class can be inherited by multiple child classes.



Child class MyVector2D

```
class MyVector2D : public MyVector
{
public:
  MyVector2D();
  MyVector2D(double m[]);
};
MyVector2D::MyVector2D()
{
  this->n = 2;
}
MyVector2D::MyVector2D(double m[]) : MyVector(2, m)
{
}
```

```
int main()
{
  double i[2] = {1, 2};
  MyVector2D v(i);
  v.print();
  cout << v[1] << endl;

  return 0;
}
```

- That is all for **MyVector2D**!
  - The modifier **public** will be discussed later.

Inheriting parent class' members

- Members in the parent class are **automatically** defined in the child class.
  - **Except** private members, constructors, and the destructor.
  - A **protected** member can only be accessed by itself and its successors.
- What are the members of **MyVector2D**?

```
class MyVector2D : public MyVector
{
public:
  MyVector2D();
  MyVector2D(double m[]);
};
```

```
class MyVector
{
protected:
  int n;
  double* m;
public:
  MyVector();
  MyVector(int n, double m[]);
  MyVector(const MyVector& v);
  ~MyVector()
  void print() const;
  // ==, !=, <, [], =, +=
};
```

Invoking parent class' constructors

- The parent class' constructor will not be inherited.
- One of them will be invoked **before** the child class' constructor is invoked.
  - Create the parent before creating the child!
- If not specified, the parent's **default** constructor will be invoked.

```
MyVector::MyVector(): n(0), m(nullptr)
{
}

MyVector2D::MyVector2D()
{
  this->n = 2;
  // this->m = nullptr is redundant
}
```

```
int main()
{
  MyVector2D v;

  return 0;
}
```

Invoking parent class' constructors

- To **specify** a parent's constructor to call, use the syntax for member initializer:
  - **Pass appropriate arguments** to control the behavior.

```
MyVector::MyVector(int n, double m[])
{
  this->n = n;
  this->m = new double[n];
  for(int i = 0; i < n; i++)
    this->m[i] = m[i];
}
MyVector2D::MyVector2D(double m[]) : MyVector(2, m)
{
  // not MyVector(2, m) here!
}
```

```
int main()
{
  double i[2] = {1, 2};
  MyVector2D v(i);
  v.print();
  cout << v[1] << endl;

  return 0;
}
```

Invoking copy constructors

- How about the copy constructor?
- If we do not define one for the child, the system provides a **default** one.
- **Before** the child's default copy constructor is invoked, the parent's copy constructor will be **automatically** invoked.

```
MyVector::MyVector(const MyVector& v)
{
  this->n = v.n;
  this->m = new double[n];
  for(int i = 0; i < n; i++)
    this->m[i] = v.m[i];
}
class MyVector2D : public MyVector
{
public:
  MyVector2D();
  MyVector2D(double m[]);
  // no copy constructor
};
```

Invoking copy constructors

- If we define a copy constructor for the child, we must **specify** the constructor we want to invoke!
  - Otherwise the parent's **default** constructor will be invoked.

```
class MyVector2D : public MyVector
{
public:
  MyVector2D();
  MyVector2D(double m[]);
  MyVector2D(const MyVector2D& v) {}
};
```

```
int main()
{
  double i[2] = {1, 2};
  MyVector2D v(i);
  MyVector2D w(v);
  w.print(); // error
  cout << w[1] << endl;

  return 0;
}
```

Using parent's member functions

- Once member variables are set properly, typically all the member functions of the parent can be used with no error.

```
void MyVector::print() const
{
  cout << "(";
  for(int i = 0; i < n - 1; i++)
    cout << m[i] << ", ";
  cout << m[n-1] << ")\n";
}
double& MyVector::operator[](int i)
{
  if(i < 0 || i >= n)
    exit(1);
  return m[i];
}
```

```
int main()
{
  double i[2] = {1, 2};
  MyVector2D v(i);
  v.print();
  cout << v[1] << endl;

  return 0;
}
```

Defining new members for the child

- A child may have **its own members**.
  - The parent has no way to access a child's member.
- Let's define a **`setValue()`** function without using arrays:
  - Note that this should never be a member of **`MyVector`**.
- We may also define new member variables and static members.

```
class MyVector2D : public MyVector
{
public:
  MyVector2D() { this-> n = 2; }
  MyVector2D(double m[]) : MyVector(2, m) {}
  void setValue(double i1, double i2);
};
void MyVector2D::setValue(double i1, double i2)
{
  if(this->m == nullptr)
    this->m = new double[2];
  this->m[0] = i1;
  this->m[1] = i2;
}
```

Invoking parent class' destructor

- When an object of the child class is to be destroyed:
  - First the child's destructor is invoked.
  - **Then** the parent's destructor is invoked **automatically**, even if we do not define a destructor for the child.

```
MyVector::~MyVector()
{
  delete [] m;
}
class MyVector2D : public MyVector
{
public:
  MyVector2D();
  MyVector2D(double m[]);
  // no destructor
};
```

Summary

- Using inheritance to create new classes is so simple!
  - We save time and enhance **consistency**.
  - Pay attention to default constructors, copy constructors, and destructors.
  - If one thing should not be inherited, set it to private.

```
class MyVector2D : public MyVector
{ // change private to protected in MyVector
public:
  MyVector2D() { this-> n = 2; }
  MyVector2D(double m[]) : MyVector(2, m) {}
  void setValue(double i1, double i2);
};
void MyVector2D::setValue(double i1, double i2)
{
  if(this->m == nullptr)
    this->m = new double[2];
  this->m[0] = i1;
  this->m[1] = i2;
}
```

Function overriding

- We may also redefine existing member inherited from a parent.
  - This typically happens to member functions.
  - We say that we **override** the member function.
- As an example, let's override **`print()`**:

```
class MyVector2D : public MyVector
{
public:
  MyVector2D() { this-> n = 2; }
  MyVector2D(double m[]) : MyVector(2, m) {}
  void setValue(double i1, double i2);
  void print() const;
};
void MyVector2D::print() const
{
  cout << "2D: (";
  for(int i = 0; i < n - 1; i++)
    cout << m[i] << ", ";
  cout << m[n-1] << ")\n";
}
```

Function overriding

- To override a parent's member function, define a child's member function with exactly the same **function signature**.
  - A child object will invoke the child's implementation.
  - The parent's implementation becomes hidden to a child object.
- Inside the child class, we may invoke a parent's member function by using `::`.

```
void MyVector2D::print() const
{
  cout << "2D: ";
  MyVector::print();
}
```

  - Use it if consistency can be enhanced.

Overriding a constant function

- What will happen to the following program?

```
int main()
{
  double i[2] = {1, 2};
  const MyVector2D v(i);
  v.print(); // 2D: (1, 2)

  MyVector2D u;
  u.setValue(3, 4);
  u.print(); // (3, 4)

  return 0;
}
```

```
class MyVector
{
  // ...
  void print() const;
};
class MyVector2D : public MyVector
{
  // ...
  void print() { MyVector::print(); }
  void print() const
  {
    cout << "2D: ";
    MyVector::print();
  }
};
```

Overriding a constant function

- How about this?

```
int main()
{
  double i[2] = {1, 2};
  const MyVector2D v(i);
  v.print(); // error!

  MyVector2D u;
  u.setValue(3, 4);
  u.print(); // (3, 4)

  return 0;
}
```

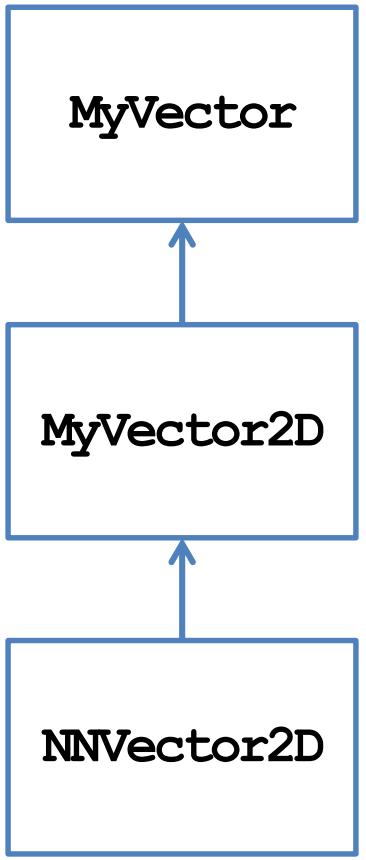
```
class MyVector
{
  // ...
  void print() const;
};
class MyVector2D : public MyVector
{
  // ...
  void print()
  {
    MyVector::print();
  }
};
```

Overriding a member variable?

- Technically, we may override a member variable.
- In general, overriding a parent's member variable is not suggested.
  - Unless you really know what you are doing.
  - After all, we will inheritance because we believe XXX is a OOO. A parent's member variable should be a part of a child!
- Overriding a parent's member function is useful.
- What is the difference between function overloading and function overriding?
- Sometimes we override a member function for efficiency.

Cascade inheritance

- While a child inherits its parent, it may have a grandchild inheriting itself.
- How may we create a class for two-dimensional nonnegative vectors?
  - $\{(x, y) \mid x \geqq 0, y \geqq 0\}$.
- A 2D nonnegative vector **is a** 2D vector!
- Let's use inheritance again.



Child class NNVector2D

- Defining **NNVector2D** is simple:

```
class NNVector2D : public MyVector2D
{
public:
  NNVector2D(); // MyVector2D's
                // constructor?
  NNVector2D(double m[]);
  void setValue(double i1, double i2);
};
NNVector2D::NNVector2D()
{
}
```

```
NNVector2D::NNVector2D(double m[])
{
  this->m = new double[2];
  this->m[0] = m[0] >= 0 ? m[0] : 0;
  this->m[1] = m[1] >= 0 ? m[1] : 0;
}
void NNVector2D::setValue
  (double i1, double i2)
{
  if(this->m == nullptr)
    this->m = new double[2];
  this->m[0] = i1 >= 0 ? i1 : 0;
  this->m[1] = i2 >= 0 ? i2 : 0;
}
```

- What happens when an **NNVector2D** object is created?
  - If we do not specify a parent's constructor, the default one will be invoked.

Child class NNVector2D

- An alternative implementation:

```
NNVector2D::NNVector2D(double m[]) : MyVector2D(m)
{
  if(m[0] < 0)
    this->m[0] = 0;
  if(m[1] < 0)
    this->m[1] = 0;
}
```

Cascade inheritance

- In general, a class has all the protected and public members (excluding constructors and destructors) of its predecessors.
- When an object is created:
  - Constructors are invoked from the oldest class to the youngest class.
  - Each constructor can specify a **one-level-above** constructor to invoke.
  - Only one level!
- When an object is destroyed:
  - Destructors are invoked from the youngest to the oldest.

Inheritance visibility

- Recall that we added the modifier **`public`** when **`MyVector2D`** inherits **`MyVector`** and when **`NNVector2D`** inherits **`MyVector2D`**.
  - This modifier specifies the **inheritance visibility**.
  - It shows how this child modify the member visibility set by its predecessors.
- When one inherits something from its parent, it may **narrow** the **visibility** of these members.
  - E.g., if my parent set its to protected, I may set it to private.
  - E.g., if my parent set its to private, I cannot set it to public.
- Why only narrowing?

Inheritance visibility

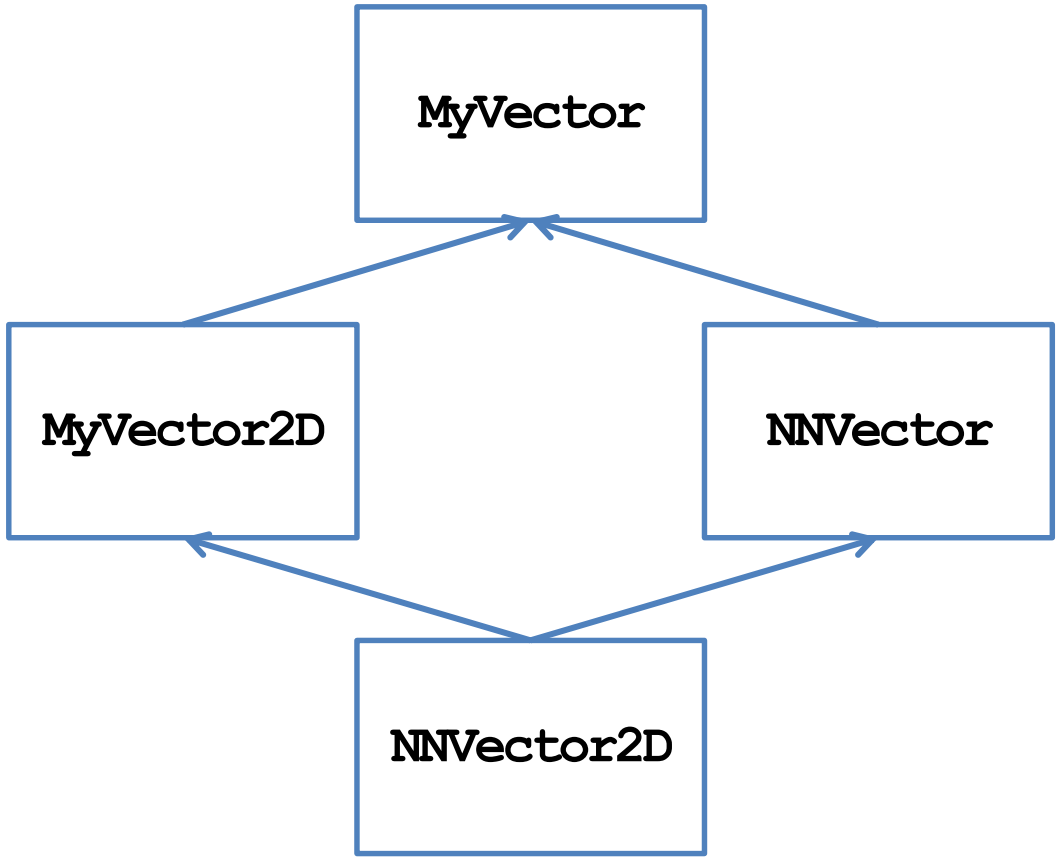
- In general, the visibility of a member in a child class depends on:
  - The member visibility by the parent.
  - The inheritance modifier.

Member visibility by the parent	Inheritance modifier		
	`public`	`protected`	`private`
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	`private`	`private`	`private`

- If you have no idea, just use public inheritance.

Multiple inheritance

- Suppose your friend argues:
  - A two-dimensional vector is a vector.
  - A nonnegative vector is a vector.
  - A two-dimensional nonnegative vector should be the child of them!
- Does that make sense?



Multiple inheritance

- In C++, **multiple inheritance** is allowed.
- However, it is not recommended!
  - In some other object-oriented programming languages (e.g., Java), multiple inheritance is forbidden.
- If there are multiple parents:
  - Whose constructor/destructor goes first?
  - Whose variables are stored in the front?
  - May I inherit from my sister? May I inherit from my grandaunt?
- We also suggest you not to do multiple inheritance (even though it has been used in C++ standard library).

Outline

- Inheritance
- **An example**
- Polymorphism

An RPG game

- In a typical Role-Playing Game (RPG), a player plays the role of a character, who keep beating enemies (monsters, bad guys, or other players' characters).
  - By beating enemies, one earns experience points to advance to higher levels and become stronger.
- In many RPGs, one can choose the **occupation** for her character(s). The occupation typically affects the **ability** of a character (e.g., a warrior and a wizard are quite different).
  - Characters with different occupations have different attributes and behave differently. However, **they are all characters**.
- Given a class **`Character`** that defines some general features of an RPG character, let's create two new classes **`Warrior`** and **`Wizard`**.

Class Character

- The class **Character** includes the name, current level, accumulated experience points, and three ability levels: power, knowledge, and luck.
  - When a character joins your team, she/he may be at any level.
  - For all characters in our game, the number of experience points required for level k is $100(k-1)^2$.
  - The number 100 is stored as a static constant **EXP_LV**.

```
class Character
{
protected:
  static const int EXP_LV = 100;
  string name;
  int level;
  int exp;
  int power;
  int knowledge;
  int luck;
};
```

Class Character

- There is a **constructor**:
  - To create a character, we must specify all its attributes except the experience point.
  - A new character at level k always starts with $100(k-1)^2$ experience points.
- There is a public function **`print()`**:
  - It prints out the current status of a character.

```
class Character
{
protected:
  static const int EXP_LV = 100;
  string name;
  int level;
  int exp;
  int power;
  int knowledge;
  int luck;
public:
  Character(string n, int lv,
            int po, int kn, int lu);
  void print();
};
```

Class Character

- There is a public function **beatMonster(int exp)**:
  - It is invoked when the character beats a monster.
  - **exp** is the number of experience points earns in this battle.
  - This function increments the accumulated experience points and brings up one's level when possible.

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

Class Character

- There is a private function **levelUp()**:
  - The character's **level** will be incremented.
  - However, her abilities will **remain the same** because characters of different occupations should get different improvements.
  - This should be specified in **Warrior** and **Wizard**.
- Finally, let's add a public member function **getName()** to return the name of a character.

```
class Character
{
protected:
  static const int EXP_LV = 100;
  // the six attributes
  void levelUp
    (int pInc, int kInc, int lInc);
    // protected member function
public:
  Character(string n, int lv,
            int po, int kn, int lu);
  void print();
  void beatMonster(int exp);
  string getName();
};
```

Implementation of Character

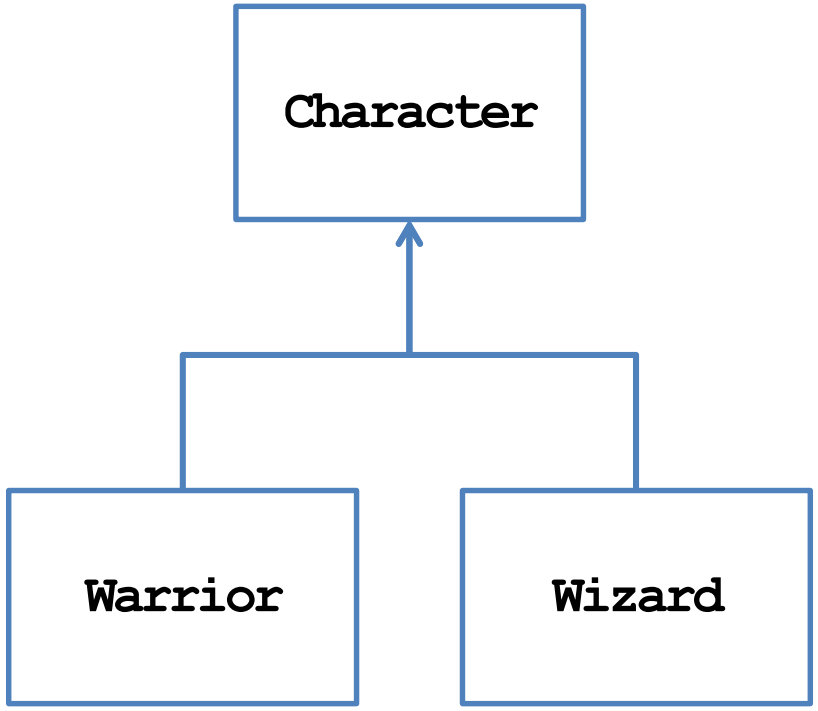
```
Character::Character(string n, int lv, int po, int kn, int lu)
  : name(n), level(lv), exp(pow(lv - 1, 2) * EXP_LV),
    power(po), knowledge(kn), luck(lu)
{
}
void Character::print() {
  cout << this->name // Mikasa: 100 (980100/1000000), 1000-500-500
       << ": " << this->level
       << " (" << this->exp << "/" << pow(this->level, 2) * EXP_LV << "), "
       << this->power << "-" << this->knowledge << "-" << this->luck << "\n";
}
string Character::getName()
{
  return this->name;
}
```

Implementation of Character

```
void Character::beatMonster(int exp)
{
  this->exp += exp;
  while(this->exp >= pow(this->level, 2) * EXP_LV)
    this->levelUp(0, 0, 0); // No improvement when advancing to the next level
}
void Character::levelUp(int pInc, int kInc, int lInc) {
  this->level++;
  this->power += pInc;
  this->knowledge += kInc;
  this->luck += lInc;
}
```

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 PO_LV = 10;
  static const int KN_LV = 5;
  static const int LU_LV = 5;
public:
  Warrior(string n, int lv = 1)
    : Character(n, lv, lv * PO_LV, lv * KN_LV, lv * LU_LV) {}
  void print() { cout << "Warrior "; Character::print(); }
  void beatMonster(int exp) // function overriding
  {
    this->exp += exp;
    while(this->exp >= pow(this->level, 2) * EXP_LV)
      this->levelUp(PO_LV, KN_LV, LU_LV);
  }
};
```

Classes Warrior and Wizard

```
class Wizard : public Character
{
private:
  static const int PO_LV = 4;
  static const int KN_LV = 9;
  static const int LU_LV = 7;
public:
  Wizard(string n, int lv = 1)
    : Character(n, lv, lv * PO_LV, lv * KN_LV, lv * LU_LV) {}
  void print() { cout << "Wizard "; Character::print(); }
  void beatMonster(int exp) // function overriding
  {
    this->exp += exp;
    while(this->exp >= pow(this->level, 2) * EXP_LV)
      this->levelUp(PO_LV, KN_LV, LU_LV);
  }
};
```

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 **wasting spaces**?

```
class Team
{
private:
  int warriorCount;
  int wizardCount;
  Warrior* warrior[10];
  Wizard* wizard[10];
public:
  Team();
  ~Team();
  // some other functions
};
```

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.
  - Characters' 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 addWar(string name, int lv);
  void addWiz(string name, int lv);
  void warBeatMonster(string name, int exp);
  void wizBeatMonster(string name, int exp);
  void printWar(string name);
  void printWiz(string name);
};
```

Outline

- Inheritance
- An example
- **Polymorphism**

Polymorphism

- The key flaw is to create two arrays, one for warriors and one for wizards.
  - 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 **`Character`**s!
  - 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**.

Why a parent variable for a child value?

- What happens to the following program?

```
int main
{
  Parent p1(1, 2);
  Child c1(3, 4, 5);
  Parent p2 = c1; // OK: 5 is discarded
  // Child c2 = p1; // Not OK: no v3
  return 0;
}
```

```
class Parent
{
protected:
  int x;
  int y;
public:
  Parent(int a, int b) : x(a), y(b) {}
};
class Child : public Parent
{
protected:
  int z;
public:
  Child(int a, int b, int c)
    : Parent(a, b)
  { z = c; }
};
```

Examples of polymorphism

- For example, we may do this:
  - A **`Character`** variable can store a **`Warrior`** or a **`Wizard`** object.
  - Because a warrior/wizard is a character!
- Alternatively, we may do this with pointers:

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

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

Polymorphism with functions

- Polymorphism is useful with **functions**:

```
void printInitial(Character c)
{
  string name = c.getName();
  cout << name[0];
}
int main
{
  Warrior alice("Alice", 10);
  Wizard bob("Bob", 8);
  printInitial(alice);
  printInitial(bob);
  return 0;
}
```

Polymorphism with arrays

- Polymorphism is useful with **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];
  // 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 memberCount;
  Character* member[10];
public:
  Team();
  ~Team();
  void addWarrior(string name, int lv);
  void addWizard(string name, int lv);
  void memberBeatMonster(string name, int exp);
  void printMember(string name);
};
```

Class Team with Polymorphism

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

```
void Team::addWarrior(string name, int lv)
{
  if(memberCount < 10)
  {
    member[memberCount] = new Warrior(name, lv);
    memberCount++;
  }
}
void Team::addWizard(string name, int lv)
{
  if(memberCount < 10)
  {
    member[memberCount] = new Wizard(name, lv);
    memberCount++;
  }
}
```

Class Team with Polymorphism

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

```
void Team::printMember(string name)
{
  for(int i = 0; i < memberCount; i++)
  {
    if(member[i]->getName() == name)
    {
      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.
  - Experience points are accumulated, but abilities remain the same.
- Why?

```
int main()
{
  Character* c[3];
  c[0] = new Warrior("Alice", 10);
  c[1] = new Wizard("Sophie", 8);
  c[2] = new Warrior("Amy", 12);
  c[0]->beatMonster(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, the default setting is to invoke the parent's implementation.
- To invoke the child's one, we need **late binding** and **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* ap = &b;
  a.a();
  a.f();
  // a.b();
  ap->a();
  ap->f();
  // ap->b();
  return 0;
}
```

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 set 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 set at the **run time**.

```
class A
{
protected:
  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 may discard values

- Why **`p2.print()`** must be the parent class' **`print()`**?

```
int main
{
  Child c(3, 4, 5);
  Parent p = c; // 5 is discarded
  p.print(); // which print()?
  return 0;
}
```

```
class Parent
{
protected:
  int x;
  int y;
public:
  Parent(int a, int b) : x(a), y(b) {}
  void print() { cout << x << " " << y; }
};
Class Child : public Parent
{
protected:
  int z;
public:
  Child(int a, int b, int c) : P(a, b)
    { z = c; }
  void print() { cout << z; }
};
```

Late binding does not discard values

- Is it possible for **`p2->print()`** to be the child class' **`print()`**?

```
int main
{
  Child c(3, 4, 5);
  Parent* pPtr = &c; // 5 is good
  pPtr->print(); // which print()?
  return 0;
}
```

- To invoke the child's implementation, we need to declare **virtual functions**.

```
class Parent
{
protected:
  int x;
  int y;
public:
  Parent(int a, int b) : x(a), y(b) {}
  void print() { cout << x << " " << y; }
};
Class Child : public Parent
{
protected:
  int z;
public:
  Child(int a, int b, int c) : Parent(a, b)
    { z = c; }
  void print() { cout << z; }
};
```

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).
  - A child cannot declare a parent's function as virtual (it is of no use).
- In summary, we need:
  - Late binding + virtual functions.

```
class Parent
{
protected:
  int x;
  int y;
public:
  Parent(int a, int b) : x(a), y(b) {}
  virtual void print() { cout << x << " " << y; }
};
Class Child : public Parent
{
protected:
  int z;
public:
  Child(int a, int b, int c) : Parent(a, b)
    { z = c; }
  void print() { cout << z; }
};
```

Virtual functions

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

```
class Character
{
protected:
  // ...
public:
  // ...
  virtual void beatMonster(int exp);
  virtual void print();
};
```

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

```
int main()
{
  Character* c[3];
  c[0] = new Warrior("Alice", 10);
  c[1] = new Wizard("Sophie", 8);
  c[2] = new Warrior("Amy", 12);
  c[0]->beatMonster(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()`** 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!

Polymorphism is everywhere

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

```
class MyVector
{
  // ...
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; // good?
  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.

Polymorphism is everywhere

- The same thing happens to **the copy constructor**:

```
void printInitial(Character c)
{
  string name = c.getName();
  cout << name[0];
}
int main
{
  Warrior alice("Alice", 10);
  Wizard bob("Bob", 8);
  printInitial(alice); // Character's copy constructor
  printInitial(bob); // Character's copy constructor
  return 0;
}
```

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.