IM 1003: Computer Programming
Inheritance

Ling-Chieh Kung

Department of Information Management

National Taiwan University

Outline

- **Inheritance**
  - **Basic ideas and the first example**
  - Constructors in child classes
  - Function overriding
  - Inheritance visibility
  - Invoking constructors and destructors

Example

- Recall the class **`Car`** you created for the last homework.
- Rename it as **`Auto`**.
- Suppose we want to create a new class **`Minivan`**.
- A **`Minivan`** can do all the things that an **`Auto`** can do.
- Besides,
  - A **`Minivan`** has four integer static attributes: **`regPer = 7`**, **`regCase = 5`**, **`spePer = 4`**, **`speCase = 8`**.
  - It also has one instance Boolean attributes **`isReg`**.
  - A **`Minivan`** can do **`void flip()`** to flip **`isReg`**.

Example

```
class Auto
{
protected: // explained later
  string plate;
  int mpl;
  int mileage;
  int gas;
public:
  Auto() { Auto("", 0); }
  Auto(string plate, int mpl);
  void drive(int gasUsed);
  void refill(int gasAdded)
  {
    this->gas += gasAdded;
  }
};
```

```
Auto::Auto(string plate, int mpl)
{
  this->plate = plate; this->mpl = mpl;
  this->mileage = 0; this->gas = 0;
}
void Auto::drive(int gasUsed)
{
  if(gasUsed >= this->gas) {
    mileage += mpl * gas;
    gas = 0;
  }
  else {
    mileage += mpl * gasUsed;
    gas -= gasUsed;
  }
}
```

Example

```
class Minivan
{
private:
  string plate;
  int mpl;
  int mileage;
  int gas;
  static int regPer;
  static int regCase;
  static int spePer;
  static int speCase;
  bool isReg;
```

```
public:
  Minivan() { Minivan("", 0); }
  Minivan(string plate, int mpl);
  void drive(int gasUsed);
  void refill(int gasAdded)
  {
    this->gas += gasAdded;
  }
  void flip();
  int getPer(); // later
  int getCase(); // later
};
```


Example

```
Minivan::Minivan(string plate, int mpl)
{
  this->plate = plate;
  this->mpl = mpl;
  this->mileage = 0;
  this->gas = 0;
  this->isReg = true;
}

void Minivan::flip()
{
  if(this->isReg == true)
    this->isReg = false;
  else
    this->isReg = true;
}
```

```
void Minivan::drive(int gasUsed)
{
  if(gasUsed >= this->gas) {
    mileage += mpl * gas;
    gas = 0;
  }
  else {
    mileage += mpl * gasUsed;
    gas -= gasUsed;
  }
}

int Minivan::regPer = 7;
int Minivan::regCase = 5;
int Minivan::spePer = 4;
int Minivan::speCase = 8;
```


Example

- They are very similar!
- In fact, the definition of **`Minivan`** includes everything in the definition of **`Auto`**.
- If we want to define a class for cars, trucks, SUVs, RVs, etc., we will need to write those codes repeatedly.
  - A lot of meaningless work.
  - Potential inconsistency.

Inheritance

- Since we have already completed the class **`Auto`**, it will be great if we can reuse it.
- A minivan is an auto, so we may to create the class **`Minivan`** by using the class **`Auto`**.
- The solution is **inheritance**.

Inheritance

- We can use **inheritance** to create new classes from existing classes.
  - This saves a lot of work on coding.
  - This creates a tighter connection among these classes.
  - This enhances **consistency**.
- One sentence to describe inheritance:
  - One class can inherit another class to "inherit", i.e., obtain, its member variables and member functions.
- The relation is like a **parent** and her **child**.

Inheritance

- When we can say that "XXX" is a "OOO", then usually we can let XXX inherit OOO.
  - A "volleyball" is a "ball".
  - A "volleyball player" is a "person".
  - A "college volleyball player" is a "volleyball player".
  - A "triangle" is a "polyhedron".
  - A "van" is an "automobile".
  - An "economy car" is a "car".
- If XXX inherits OOO, then:
  - OOO is the super class or **base class**.
  - XXX is the sub class or **derived class**.
- Then XXX will have OOO's members.

Example: with inheritance

```
class Minivan : public Auto
{
private:
  static int regPer;
  static int regCase;
  static int spePer;
  static int speCase;
  bool isReg;
public:
  Minivan();
  Minivan(string plate, int mpl);
  void flip();
  int getPer();
  int getCase();
};
```

```
Minivan::Minivan()
{
  Minivan(plate, mgl);
  this->isReg = true;
}

Minivan::Minivan
  (string plate, int mpl)
{
    this->plate = plate;
    this->mpl = mpl;
    this->mileage = 0;
    this->gas = 0;
    this->isReg = true;
}
```


Example: with inheritance

```
int Minivan::getPer()
{
  if(this->isReg == true)
    return Minivan::regPer;
  else
    return Minivan::spePer;
}

int Minivan::getCase()
{
  if(this->isReg == true)
    return Minivan::regCase;
  else
    return Minivan::speCase;
}
```

```
void Minivan::flip()
{
  if(this->isReg == true)
    this->isReg = false;
  else
    this->isReg = true;
}

int Minivan::regPer = 7;
int Minivan::regCase = 5;
int Minivan::spePer = 4;
int Minivan::speCase = 8;
```

Child class definition

- **`class child class : public parent class`**
  `{`
  `// its own members`
  `};`
  – The modifier "**`public`**" will be discussed later.
- The child's members = its own members + its parent's own members (+ its grandparent's + …).
  – After we let **`Minivan`** inherit **`Auto`**, it has attributes **`plate`**, **`mpl`**, **`mileage`**, and **`gas`**.
  – It can invoke **`refill()`**, **`drive()`**, etc.

Child class' own members

- A derived class can define its own member variables and member functions as well as before.
  – Static variables: **`regPer`**, **`regCase`**, **`spePer`**, **`speCase`**.
  – Instance variable: **`isReg`**.
  – Instance function: **`flip()`**, **`getPer()`**, **`getCase()`**.
- Of course, a parent cannot access its child's members.

Main advantages of inheritance

- We do not need to define those common members for a child class again. The codes can be much simpler.
- This also avoids inconsistency between a child and its parent.
- If someday we want to modify a parent class, we will not need to do it again for a child class.

Outline

- **Inheritance**
  – Basic ideas and the first example
  – **Constructors in child classes**
  – Function overriding
  – Inheritance visibility
  – Invoking constructors and destructors

Constructors in child classes

- A parent's constructors will **not** be inherited by its children!
- However, when creating a child object, the system will **invoke its parent's constructor** before the child's constructor.
- If the parent still has a parent, then the grandparent's constructor will be called first.
- We may (and usually we should) indicate which constructor of the parent to invoke. Otherwise, the system will invoke the parent's **default constructor**.

Constructors in child classes

- Suppose **`C`** inherits **`P`**.
- For a constructor of **`C`**:

```
C::C(parameters) : P(arguments for P's constructor)
{
  // something to do for the child
}
```

- Use "**`:`**" to call the parent's constructor, and use arguments to indicate the one you want to call.

Constructors in child classes

- So we may implement an original constructor of **`Minivan`** by invoking a constructor of **`Auto`**.
- How to rewrite the following constructor?

```
Minivan::Minivan(string plate, int mpl)
{
  this->plate = plate;
  this->mpl = mpl;
  this->mileage = 0;
  this->gas = 0;
  this->isReg = true;
}
```


Constructors in child classes

- To specify a constructor of **`Auto`**, we write

```
Minivan::Minivan(string plate, int mpl) : Auto(plate, mpl)
{
  this->isReg = true;
}
```

- Then the following constructor of Auto

```
Auto::Auto(string plate, int mpl)
{
  this->plate = plate;
  this->mpl = mpl;
  this->mileage = 0;
  this->gas = 0;
}
```

does its job before those remaining in the constructor of **`Minivan`**.

Constructors in child classes

- Be careful to invoke the right constructor of the parent.
- Remember that if you do not indicate one, the default constructor of the parent will be invoked.
- Write the default constructor by yourself when possible!

Outline

- **Inheritance**
  - Basic ideas and the first example
  - Constructors in child classes
  - **Function overriding**
  - Inheritance visibility
  - Invoking constructors and destructors

Function overriding

- Let's implement **`void Auto::print();`** to print the four attributes in one line.

```
void Auto::print()
{
  cout << this->plate << " " << this->mpl << " "
       << this->mileage << " " << this->gas << endl;
}
```


Function overriding

- You may use a **`Minivan`** object to invoke **`print()`** since **`Minivan`** is a child of **`Auto`**.

```
Minivan m("ABCDEFG", 10);
m.print();
```

- However, the function **`print()`** is **incomplete** for **`Minivan`**:
  - It does not print the current status according to **`isReg`**.
- We can define another function **`printMinivan()`** to do this.
- However, it will be more meaningful and convenient to **use the same name** **`print()`**.
  - Some other benefits will become clear with polymorphism.
- May function overloading help in this case?

Function overriding

- The capability of function overloading is limited:
  - The parameters must be different.
  - So you can not have two **`print()`**s for **`Auto`**.
- The solution is "**function overriding**".
  - This functionality is specifically for classes with inheritance.

Function overriding

- We are allowed to define two instance functions with the **same signature** (name and parameters) in two different classes.
  - In particular, this is allowed for a parent and a child.
- Suppose a parent has a function **`f()`**:
  - Suppose the child does **not** have **`f()`**: When the child invokes **`f()`**, the **parent's** will be invoked.
  - Suppose the child **has** its own **`f()`**: When the child invokes **`f()`**, the **child's** will be invoked by default.
- Then we say the child's **`f()`** "**overrides**" the parent's.
- In a child's member function, we can still invoke the parent's member function with special indication.

Example: overriding `print()`

```
void Minivan::print() // overriding Auto::print()
{
  Auto::print(); // invoking the parent's print()
  // if no "Auto::", Minivan::print() will be called: recursion!
  if(this->isReg == true)
    cout << "(" << this->regPer << ", "
         << this->regCase << ")" << endl;
  else
    cout << "(" << this->spePer << ", "
         << this->speCase << ")" << endl;
}
```


Exercise

- Suppose a minivan's mileage per litter varies with its modes.
  - In the regular mode, one litter of gas allows the minivan to run **`mpl`** miles.
  - In the special mode, one litter of gas allows the minivan to run only **`ceil(0.8 * mpl)`** miles.
- How would you override the function **`drive(int gasUsed)`** for **`Minivan`**?

Outline

- **Inheritance**
  - Basic ideas and the first example
  - Constructors in child classes
  - Function overriding
  - **Inheritance visibility**
  - Invoking constructors and destructors

Inheritance visibility

- Let's change the **`protected`** modifier in **`Auto`** to **`private`**.
- Then in,

```
void Minivan::refill(int gasAdded)
{
  this->gas += gasAdded;
}
```

  A compilation error will appear, saying that we try to access a **private member** of **`Auto`** outside its class definition.
- Why inheritance fails?

Inheritance visibility

- A private member of the parent is not accessible by any one.
  - Even its children.
  - As a father, one may still choose to leave some properties to himself only.
- This is why we need the third visibility modifier: "**`protected`**".
  - Only **public** and **protected** members of a parent may be left to descendants.
  - Therefore, those members that should be inherited by **`Minivan`** should be **`protected`** instead of **`private`** in **`Auto`**.
- When a child is inheriting those left by its parent, it may modify the visibility of these members **starting from its generation**.
  - This is realized with different **inheritance levels**.

Inheritance visibility

- The way **`Minivan`** inherits **`Auto`** is a **public inheritance**.
  - The visibility specified by the parents will all remain unchanged.

```
class Minivan : public Auto
{
  // ...
};
```

- There are also **protected inheritance** and **private inheritance**.
  - Protected inheritance: A member that is **public** in the parent class will become **protected** starting from the child's generation.
  - Private inheritance: A member that is **public** or **protected** in the parent class will become **private** starting from the child's generation.

Inheritance visibility

- Table of **levels of inheritance**:

member visibility	level of inheritance		
	public	**protected**	**private**
public	**public**	**protected**	**private**
protected	**protected**	**protected**	**private**
private	**private**	**private**	**private**

  – Visibility will only be narrowed.
  – Use public inheritance if you have no idea.

Outline

- **Inheritance**
  – Basic ideas and the first example
  – Constructors in child classes
  – Function overriding
  – Inheritance visibility
  – **Invoking constructors and destructors**

Exercise

- Write a problem with three classes: **G**, **P**, and **C**.
- Let **C** inherit **P** and **P** inherit **G**.
- Create constructors for the three classes with some outputs inside them so that you may see these constructors are invoked.
- Create an object of class **C** and see the constructors of **G** and **P** are really invoked.
- Create destructors for the three classes with some outputs inside them so that you may see these destructors are invoked.
- What is the sequence of invoking destructors?