IM 1003: Computer Programming
Polymorphism

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Outline

- **Polymorphism**
  - **Preparation**
  - Basic ideas and the first example
  - Virtual functions

Parent and child classes

- We have defined two classes:

```
class Auto
{
protected:
  string plate;
  int mpl;
  int mileage;
  int gas;
public:
  // ...
};
```

```
class Minivan : public Auto
{
private:
  static int regPer;
  static int regCase;
  static int spePer;
  static int speCase;
  bool isReg;
public:
  // ...
};
```


Parent and child classes

- Suppose we have defined a member function **`hasHigherMpl()`** in **`Auto`**, which compares a given **`Auto`**'s **`mpl`** with that of the **`Auto`** invoking this function.

```
bool Auto::hasHigherMpl(Auto a)
{
  if(this->mpl > a.mpl)
    return true;
  else
    return false;
}
```

Parent and child classes

- We have also defined a member function **print()** in **Auto** and then overrode it in **Minivan**.
- We will use these functions to illustrate the idea of polymorphism.

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

```
void Minivan::print()
{
  Auto::print();
  cout << " ";
  if(this->isReg == true)
    cout << "(" << this->regPer << ", "
         << this->regCase << ")";
  else
    cout << "(" << this->spePer << ", "
         << this->speCase << ")";
}
```


Outline

- **Polymorphism**
  - Preparation
  - **Basic ideas and the first example**
  - Virtual functions

Comparisons among different classes

- Consider the **hasHigherMpl()** function of **Auto**. This allows us to compare an **Auto** with another **Auto**.

```
Auto a1("car1", 10);
Auto a2("car2", 12);
cout << a1.hasHigherMpl(a2); // 0 or 1?
```

- What if we want to compare an **Auto** with a **Minivan**?
- We "may" use **function overloading**.

Comparisons among different classes

- With function overloading, we may define another **hasHigherMpl()** whose parameter is a **Minivan**.
- If there is another class **Truck**, we may define one more.
- Two things are bad:
  - All these **hasHigherMpl()** are almost **identical**.
  - Whenever we create one more type of auto, we need to **modify the parent class** **Auto**.

```
bool Auto::hasHigherMpl(Auto a)
{
  if(this->mpl > a.mpl)
    return true;
  else
    return false;
}

bool Auto::hasHigherMpl(Minivan m)
{
  if(this->mpl > m.mpl)
    return true;
  else
    return false;
}
```

Comparisons among different classes

- We want to compare:
  - An **`Auto`** with an **`Auto`**
  - An **`Auto`** with a **`Minivan`**
  - A **`Minivan`** with an **`Auto`**
  - A **`Minivan`** with a **`Minivan`**.
- "It seems that" we need
  - Two overloaded instance functions in **`Auto`**.
  - Two overloaded instance functions in **`Minivan`**.
- With a parent class **`Auto`** and n child classes, "it seems that" we need $(n + 1)^2$ almost identical instance functions!
- Does inheritance help?

Comparisons among different classes

- Fortunately, inheritance allows us to define only $n + 1$ functions in the parent class **`Auto`**.
  - Then all child classes inherit these functions.
- But the two drawbacks are still there:
  - We still need $n + 1$ almost identical **`hasHigherMpl()`**.
  - When we create a child class, we need to modify the parent class **`Auto`**.
- Can we do better?

Store different types of autos

- Suppose in a program, there are all kinds of autos: sedans, trucks, minivans, etc.
- We want to store all these autos in arrays.
  - In C++, all elements in an array must have the same type.
  - Do we need to prepare one separate array for each type of autos?
  - May we store all of them in one single array?

Polymorphism

- The three principles of OOP are
  - Encapsulation
  - Inheritance
  - Polymorphism
- **Polymorphism**: a lot of appearances.
  - One thing can **behave differently** in different situations.
- It requires inheritance.
  - It can be applied only on ancestor-descendent relationships.
- It is the most difficult to understand.
- However, it can be very useful and powerful.
  - At least it will help us solve the two problems we just mentioned.

Variables vs. values

- To apply it, first we need to 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 (put into a container). E.g., `12.5` or `7`.
  - Note that the value has its own type, which may be **different** from the variable/container's type.
- In C++, the way we implement polymorphism is to

"Use a variable of a parent type to store a value of a child type."

Polymorphism

- Suppose we have the following two classes:

```
class A // A is B's parent
{
public:
  void f() { cout << "AAA!\n"; }
};
class B : public A  // B is A's child
{
public:
  void f() { cout << "BBB!\n"; }
};
```

- Then we can write…
- Though this is allowed, what is `a`?
  - It is an `A` object or a `B` object?

```
B b;
A a = b;
```


Polymorphism

- Similarly, we may write

```
Minivan m;
Auto a = m;
```

  - Is `a` an `Auto` or a `Minivan`?
- This is exactly "using a variable of a parent type to store a value of a child type".
- Let's go back to our example with classes `A` and `B`.
- What will happen if we invoke `f()`, the overridden function?
  - Easier: How about `b.f()`?
  - Harder: How about `a.f()`?

```
B b;
A a = b;
```


Polymorphism

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

- No matter what is the type of the value `a` contains, because `a`'s type is `A`, `a.f()` will call `A::f()`.
- It is because at the time of **compilation**, the compiler does not know what value `a` will contain when `a.f()` is executed.
  - `a.f()` is bound with the container `a`'s type `A`. This is called "**early binding**".

How polymorphism helps

- Thanks to polymorphism, because **`Minivan`** inherits **`Auto`**, an **`Auto`** variable can store a **`Minivan`** value.
- Thus, the following program is valid:

```
Auto anAuto;
Minivan aMinivan;
Auto who;
who = anAuto; // no error
who.print();
who = aMinivan; // no error
who.print();
```


How polymorphism helps

- Therefore, we can simply define one single function **`hasHigherMpl(Auto a)`** in **`Auto`**.
- The parameter's type is **`Auto`**. Because **`Minivan`** is a child of **`Auto`**, **`a`** can store the value of an **`Auto`** or a **`Minivan`**.

```
Auto a1("car1", 10);
Minivan m("minivan1", 9);
Auto a2("car2", 12);
a1.hasHigherMpl(a2); // no error
a1.hasHigherMpl(m); // no error
```

- So only one definition of **`hasHigherMpl()`** is enough!

Polymorphism

- The most frequently used applications of polymorphism are
  - In a **function parameter**.
  - In an **array**.
- We can have an array of **`Auto`** to store **`Minivan`**, **`Sedan`**, **`Truck`**, etc., without multiple separated arrays.

Outline

- **Polymorphism**
  - Preparation
  - Basic ideas and the first example
  - **Virtual functions**

How to invoke the right function?

- Consider the next example:

```
Auto a("car1", 10);
Minivan m("minivan1", 9);
Auto who[2];
who[0] = a;
who[1] = m;
who[0].print(); // four attributes
who[1].print(); // still four attributes orz
```

- **a** can only invoke **Auto::print()**, since the rule of polymorphism is to invoke the overridden function according to **the type of the container**, not the type of the value.

Virtual functions

- The solution is to use "**virtual functions**" to do "**late binding**".
  - A virtual function is still an instance function.
  - However, it implements late binding.
- If a virtual function is overridden, it will be invoked according to **the value's type**, not the container's type.
- Declaring a virtual function:

```
class Auto
{
  // ...
  virtual void print(); // virtual function
};
```

  - We do not need to declare virtual in child classes. However, doing so makes the program clearer.

Virtual functions

- To implement late binding, we need to do one more thing: Using pointers instead of "real objects".
- When we write **Auto a;**, the compiler creates a real **Auto** **object**.
  - It allocates a memory space for the **four** instance variables.
- No matter what value is assigned to **a**, **a** is still an **Auto** object.
  - In particular, if a **Minivan** is assigned to **a**, **isReg** will be **discarded**.
  - It is thus impossible to print out anything regarding **isReg**.
- However, when we write **Auto* a;**, the compiler only creates an **Auto** **pointer**.
  - It can point to an **Auto**, a **Minivan**, or any descendent of **Auto**.
- Therefore, we will use a pointer to "mimic" an object.

Virtual functions

- A parent pointer can point to a child object.
- The compiler will determine the function to invoke during the running time (late binding).

```
Auto a("car1", 10);
Minivan m("minivan1", 9);
Auto* who;
who = &a;
who->print(); // four attributes
who = &m;
who->print(); // six attributes
```

```
Auto* who = NULL;
who = new Auto("car1", 10);
who->print(); // four attributes
delete who;
who = new Minivan("minivan1", 9);
who->print(); // six attributes
delete who;
```

Example

- Why in the following program, still only four attributes are printed out for **`Minivan`** values?
- How to modify it?
  – You also need to use **pointers** as function parameters to implement late binding.

```
void print(Auto autos[], int n)
{
  for(int i = 0; i < n; i++)
    autos[i].print(); // four attributes
}

int main()
{
  Auto a("a", 10);
  Minivan m1("m1", 8), m2("m2", 9);
  Auto autos[3]; // early binding
  autos[0] = a; autos[1] = m1; autos[2] = m2;
  print(autos, 3);
  return 0;
}
```


Summary

- It 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 variable/container of a parent type to store a value of a child type".
- To implement late binding, you need to
  – Declare **virtual functions** and
  – Use parent **pointers** to point to child objects.