ECE326

PROGRAMMING LANGUAGES

Lecture 8 : Inheritance and Runtime Polymorphism

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2019

Subtyping

- In some languages, means same thing as inheritance
- However, subtyping inherits only the *interface*
  - Forms a strict “is-a” relationship
    - Allows for substitution where parent type is expected
  - Has semantic relationship with parent type
    - E.g. Dog being a subtype of Animal has semantic meaning
- Inheritance
  - Parent class may have no semantic relationship
    - E.g. Chat class inherits from Network class to gain its implementation for making connections to other participants over the Internet

Inheritance

- Creates new class (subclass) based on existing one(s)
  - Acquires all attributes and behaviours of parent (base class)
  - Retains all existing implementation
    - Enables code reuse
  - Can replace (override) existing implementation
  - Can extend to support new behaviours
    - Add more functionality
- Used interchangeably
  - Subclass – child class – derived class
  - Super class – parent class – base class

Inheritance in Python

```
class Animal:
    def __init__(self, age, weight):
        self.age, self.height = age, height
    def move(self, location):
        print(“It moved to %s”%location)

class Dog(Animal):
    def __init__(self, age, weight, name=“Marley”):
        self.age, self.height = age, height
        self.name = name
    def move(self, location):
        print(“%s moved to %s”%(self.name, location))

>> dog = Dog(5, 35.2)
>> dog.move(“the park”)
Marley moved to the park
```

Runtime Polymorphism

- Choosing behaviour through single interface at runtime
  - E.g. C++ virtual function
  - Decides which implementation of a virtual function to call
- There are no non-virtual functions in Python
  - Also no pure virtual functions (why?)
  - Everything can be overridden
    - By child class or by manual manipulation
    - Sometimes by accident
  - If a child class has an attribute of the same name, it will take precedence over the parent's

super

- Allows accessing attribute of the super class without having to specify “which” super class
  - Very important for multiple inheritance

```
class Dog(Animal):
    def __init__(self, age, weight, name=“Marley”):
        Animal.__init__(self, age, weight)
        self.name = name
```

```
class Dog(Animal):
    def __init__(self, age, weight, name=“Marley”):
        super().__init__(age, weight)
        self.name = name
```

Dynamic Dispatch

- A *polymorphic* operation has different implementations
- Dynamic dispatch determines which to call at *runtime* based on *context*
  - Context can include caller's type and input types
- Static Dispatch
  - Ad-hoc polymorphism
  - Knows which function to call based on their signatures
  - Can be done at compile-time
    - E.g. function overloading, operator overloading, …etc

Single Dispatch

- C++ virtual functions
- Context is based solely on type of instance
  - *Not* the reference type of the variable

```
struct A {
    virtual void foo() {
        cout << “A::foo\n”;
    }
};

struct B : public A {
    virtual void foo() override {
        cout << “B::foo\n”;
    }
};
```

```
B b = B();

// reference type is A
// instance type is B
A * ap = &b;

// prints B::foo
ap->foo();
```

Override Keyword

- Available since C++11
- Compile-time error if virtual function does not override
  - Helps detect unexpected bugs on signature change

```
struct A {
	virtual void foo(long a) {
		cout << “A::foo ” << a << endl;
	}
};
struct B : public A {
	virtual void foo(int a) override {
		cout << “B::foo” << a << endl;
	}
};
```

Error: B:foo marked ‘override’ but does not override

Final Keyword

- Error when attempting to override a final function
  - Helps prevent accidental overriding

```
struct B : public A {
	virtual void foo(long a) final {
		cout << “B::foo ” << a << endl;
	}
};
struct C : public B {
	virtual void foo(long a) {
		cout << “C::foo” << a << endl;
	}
};
```

Error: overriding final function B::foo()

Final Keyword

- Error when attempting to inherit from a final class

```
struct B final : public A {
     virtual void foo(long a) {
          cout << “B::foo ” << a << endl;
     }
};
struct C : public B {
     …
};
```

Error: cannot derive from final base ‘A’

Virtual Table

- Implements dynamic dispatch in C++
- A lookup table to resolve virtual function calls
  - Implemented as an array of function pointers

```
struct A {
    virtual void foo() {}
    virtual void bar() {}
};

struct B : public A {
    virtual void foo() override {}
};

struct C : public A {
    virtual void bar() override {}
};
```

Virtual Table

- For each class in the hierarchy, a VTable is created

```
struct A {
    virtual void foo() {}
    virtual void bar() {}
};

struct B : public A {
    virtual void foo() override {}
};

struct C : public A {
    virtual void bar() override {}
};
```

A::vtable

- A::foo
- A::bar

B::vtable

- B::foo
- A::bar

C::vtable

- A::foo
- C::bar

Virtual Table Pointer

- In the base class, a hidden pointer is added
  - Both base and derived class will have this pointer (why?)

```
struct A {
     /* points to array of member
      * function pointers          */
     void * __vptr;
     virtual void foo() {}
     virtual void bar() {}
};
```

Virtual Table Pointer

- During instantiation, it will be set based on its type
  - __vptr will point to the corresponding virtual table for its instance type

```
struct A {
    void * __vptr;
};

struct B : public A {
    void * __vptr; // inherited
};

struct C : public A {
    void * __vptr; // inherited
};
```

A::vtable

A::foo
A::bar

B::vtable

B::foo
A::bar

C::vtable

A::foo
C::bar

Virtual Function Call

- Instead of calling function directly, goes through VTable
- Each virtual function has a fixed index in Vtable
  - Index based on order of appearance in class definition
  - Use __vptr and index to call the actual function

```
B b = B();
A * ap = &b;

// goes through virtual table, calls B::foo
ap->foo();

// the actual implementation
// uses syntax for member function pointer call
(ap->*__vptr[0])();
```

Multiple Dispatch

- Context also includes input parameter types
- **Julia** – dynamically-typed just-in-time compiled language
  - Used by scientific communites for its high performance
- Example
  - Suppose we have a polymorphic function, *eat*, where an instance of type Animal eats some an instance of type Food

```
abstract type Animal end
abstract type Food end

eat(eater:Animal, meal:Food) = println(“yum!”)
```

Multiple Dispatch

- Now let’s add some real animals and food…

```
struct Dog <: Animal end
struct Lion <: Animal end
struct Sheep <: Animal end
```

```
struct Carrot <: Food end
struct Beef <: Food end
```

- We want to make sure some animals reject some food

```
eat(eater:Lion, meal:Carrot) = println(“yuk!”)
eat(eater:Sheep, meal:Beef) = println(“puke!”)
```

```
>> kimba = Lion()
>> eat(kimba, Carrot())
yuk!
```

```
>> fido = Dog()
>> eat(fido, Beef())
yum!
```

Multiple Dispatch

- How to do the same thing in C++?

```
struct Animal {
     virtual void eat(Carrot * meal) { cout << “yum!\n”; }
     virtual void eat(Beef * meal) { cout << “yum!\n”; }

     /* Question: why won’t ‘eat(Food * meal)’ work?
};

struct Lion : public Animal {
     virtual void eat(Carrot * meal) { cout << “yuk!\n”; }
};
…
```

- What if there were more animals and food?
- What if we add other parameters? (e.g. time of day)

Multiple Dispatch

- Emulating multiple dispatch in C++
  - Use *N*-dimensional array of function pointers

```
enum Animal_ID {
     DOG = 0,
     LION = 1,
     SHEEP = 2,
};
```

```
enum Food_ID {
     CARROT = 0,
     BEEF = 1,
};
```

```
void (* matrix[3][2])(Animal *, Food *) = {
     { animal_eat_food, animal_eat_food, },
     { lion_eat_carrot, animal_eat_food, },
     { animal_eat_food, sheep_eat_beef,  },
};

// both Animal and Food class have associated ID field
void eat(Animal * a, Food * f) {
     matrix[a->animal_id][f->food_id](a, f);
}
```

Late Binding

- Associates *name* with an operation at runtime
  - Type *unknown* until use (e.g., evaluation)
- Early Binding
  - Type is known at time of instantiation
- Usage of term sometimes conflated
  - Can mean dynamic dispatch or “duck typing” (defer to later lecture)
  - Quote from father of OOP
    - “OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme *late-binding* of all things.”

      – Alan Kay

Late vs. Early

```
class Point:
    def __str__(self):
        return str((self.x, self.y))
    def move(self, dx, dy):
        self.x, self.y = self.x + dx, self.y + dy
```

- Late Binding

```
>> a = Point(2, 3)
>> def move2(self, x, y):
..     self.x = x
..     self.y = y

>> Point.move = move2
>> a.move(5, 5)
>> print(a)
(5, 5)
```

- Early Binding

```
>> a = Point(2, 3)
>> def move2(self, x, y):
     …

# assume you can do this
>> Point.move = move2
>> a.move(5, 5)
>> print(a)
(7, 8)
```

Binding vs. Dispatch

- Both are examples of runtime polymorphism
- Late binding is concerned with the *object*
  - Calling method by *name*
  - Name resolved to method by *object type*
  - Object behaviour can change after instantiation
- Dynamic dispatch is concerned with the *operation*
  - Calling method by *context*
    - Context determines which implementation to call
  - Object behaviour remains the same after instantiation