Creative Software Programming

9 – Polymorphism 1

Today's Topics

- What is Polymorphism?
- Pointers, References and Inheritance
- Polymorphism in C++
- Virtual Function
- Virtual Destructor
- Caution: Object Slicing

What is Polymorphism?

- From a Greek word: “poly” means "many, much" and “morphism” means "form, shape“

- The ability to create a variable, a function, or an object **that has more than one form**. [wikipedia] - 다형성 (多形性).

- In other words,
  - Ability of type A to appear as and be used like another type B
  - Ability to provide **access to entities of different types through single interface**

- One of the fundamental OOP principles

Real-world Examples

- Steering wheel + accelerator + brake in trucks or cars.

  the same interface for *entities of different types*

- Volume + channel control in TV or DVD player remotes.

  the same interface for *entities of different types*

- Shutter button for film or digital cameras.

  the same interface for *entities of different types*

Types of Polymorphism

- **Subtype polymorphism (today's topic)**
  - Ability to **access a derived class object** through **its base class interface**
  - Often simply referred to as just "polymorphism".

- Ad hoc polymorphism
  - Allows functions with the same name act differently for each type
  - Overloading in C++

- Parametric polymorphism
  - Allows a function or a data type to be written generically
  - Templates in C++

- Coercion polymorphism
  - (Implicit or explicit) casting in C++

An Example of Subtype Polymorphism

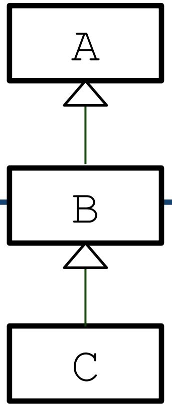
```
class Animal {
 public:
  virtual string Talk() = 0;
};

class Cat : public Animal {
 public:
  virtual string Talk() { return "Meow!"; }
};

class Dog : public Animal {
 public:
  virtual string Talk() { return "Woof!"; }
};

void LetsHear(Animal& animal) {
  cout << animal.Talk() << endl;
}

int main() {
  Cat cat;
  LetsHear(cat);

  Dog dog;
  LetsHear(dog);

  return 0;
}
```

Pointers, References and Inheritance

- To use polymorphism in C++, you first have to understand **how to use pointers and references with inheritance**

- Recall that inheritance implies "is-a" relationship
  - A car is a vehicle.
    A truck is a vehicle.
    A cart is a vehicle.
    ...

Pointers with Inheritance



- A base class `(B)` pointer can store
  - the address of the base class `(B)` object
  - the address of its derived class `(C)` object
  - CANNOT store the address of the object of the parent of the base class `(A)`

… because C is a B, but A is not a B

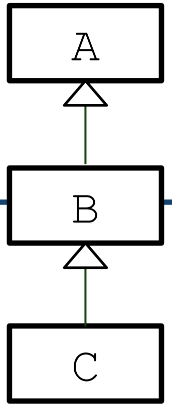
```cpp
#include <iostream>
using namespace std;

class Person {
 public:
  void Talk() {
    cout << "talk" << endl;
  }
};

class Student : public Person {
 public:
  void Study() {
    cout << "study" << endl;
  }
};

class CSStudent : public Student {
 public:
  void WriteCode() {
    cout << "write_code" << endl;
  }
};
```

```cpp
int main() {
  Person* p1 = new Person;
  Person* p2 = new Student;
  Person* p3 = new CSStudent;

  Student* s1 = new Person;  // error
  Student* s2 = new Student;
  Student* s3 = new CSStudent;

  delete p1;
  delete p2;
  delete p3;

  delete s1;
  delete s2;
  delete s3;

  return 0;
}
```

```cpp
#include <iostream>
using namespace std;

class Person {
 public:
  void Talk() {
    cout << "talk" << endl;
  }
};

class Student : public Person {
 public:
  void Study() {
    cout << "study" << endl;
  }
};

class CSStudent : public Student {
 public:
  void WriteCode() {
    cout << "write_code" << endl;
  }
};
```

```cpp
int main() {
  Student st;

  Person* person_st = &st;  // ok
  Student* student_st = &st;  // ok
  CSStudent* csstudent_st = &st;  //error!

  CSStudent csst;

  Person* person_csst = &csst;  // ok
  Student* student_csst = &csst;  // ok
  CSStudent* csstudent_csst = &csst;  //ok

  return 0;
}
```

Pointers with Inheritance



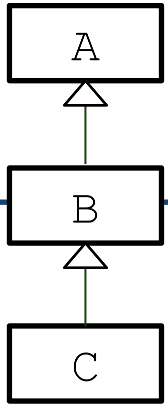
- A derived class `(B)` pointer can access
  - the members of its base class `(A)`
  - the members of the derived class `(B)`
  - CANNOT access the members of its child class `(C)`

```cpp
#include <iostream>
using namespace std;

class Person {
 public:
  void Talk() {
    cout << "talk" << endl;
  }
};

class Student : public Person {
 public:
  void Study() {
    cout << "study" << endl;
  }
};

class CSStudent : public Student {
 public:
  void WriteCode() {
    cout << "write_code" << endl;
  }
};
```

```cpp
int main() {
  Student st;
  Person* person_st = &st;

  person_st->Talk();
  person_st->Study();  // error!
  person_st->WriteCode();  // error!

  return 0;
}
```

```cpp
int main() {
  Student st;
  Student* student_st = &st;

  student_st->Talk();
  student_st->Study();
  student_st->WriteCode();  // error!

  return 0;
}
```

References with Inheritance



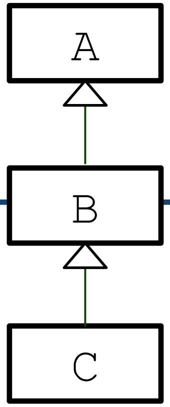
- A base class `(B)` reference can refer to
  - the base class `(B)` object
  - its derived class `(C)` object
  - CANNOT refer to the object of the parent of the base class `(A)`

- Exactly the same as the pointers!

```cpp
#include <iostream>
using namespace std;

class Person {
 public:
  void Talk() {
    cout << "talk" << endl;
  }
};

class Student : public Person {
 public:
  void Study() {
    cout << "study" << endl;
  }
};

class CSStudent : public Student {
 public:
  void WriteCode() {
    cout << "write_code" << endl;
  }
};
```

```cpp
int main() {
  Student st;

  Person& person_st = st;  // ok
  Student& student_st = st;  // ok
  CSStudent& csstudent_st = st;  //error!

  CSStudent csst;

  Person& person_csst = csst;  // ok
  Student& student_csst = csst;  // ok
  CSStudent& csstudent_csst = csst;  //ok

  return 0;
}
```

References with Inheritance



- A derived class `(B)` reference can access
  - the members of its base class `(A)`
  - the members of the derived class `(B)`
  - CANNOT access the members of its child class `(C)`

- Exactly the same as the pointers!

```cpp
#include <iostream>
using namespace std;

class Person {
 public:
  void Talk() {
    cout << "talk" << endl;
  }
};

class Student : public Person {
 public:
  void Study() {
    cout << "study" << endl;
  }
};

class CSStudent : public Student {
 public:
  void WriteCode() {
    cout << "write_code" << endl;
  }
};
```

```cpp
int main() {
  Student st;
  Person& person_st = st;

  person_st.Talk();
  person_st.Study(); // error!
  person_st.WriteCode(); // error!

  return 0;
}
```

```cpp
int main() {
  Student st;
  Student& student_st = st;

  student_st.Talk();
  student_st.Study();
  student_st.WriteCode(); // error!

  return 0;
}
```

Polymorphism in C++

- Subtype polymorphism *(will be referred to as just "polymorphism" in this lecture)* in C++ requires **references** or **pointers**
  - In C++, Polymorphic behavior is only possible when an object is referenced by a reference or a pointer

- **A derived class object is treated as if it were its base class type** by accessing through a pointer or reference!

Polymorphism in C++

- In this example,
- Derived class objects (`Student st`, `CSStudent csst`)
- are treated as if they were their base class type (`Person`)
- by accessing through references (`person_st`, `person_csst`)

```
int main() {
  Student st;
  CSStudent csst;

  Person& person_st = st;
  Person& person_csst = csst;

  person_st.Talk();
  person_csst.Talk();
  ...
}
```

Quiz #1

```
#include <iostream>
using namespace std;

class Person {
 public:
  void Talk() { cout << "talk"; }
};

class Student : public Person {
 public:
  void Study() { cout << "study"; }
};

class CSStudent : public Student {
 public:
  void WriteCode() {cout << "write_code"; }
};

class Faculty : public Person {
 public:
  void Teach() { cout << "teach"; }
};
```

```
int main() {
  Person ps;
  Student st;
  CSStudent csst;
  Faculty fc;

  Person* p1 = &ps;       // 1
  Person* p2 = &st;       // 2
  Person& p3 = csst;      // 3
  Person& p4 = fc;        // 4

  Student& s1 = ps;       // 5
  Student* s2 = &csst;    // 6
  Student* s3 = &fc;      // 7

  p4.teach();        // 8
  s2->talk();        // 9
}
```

- What line number generates a compile error?

Recall: Overriding Member Function

- You can override a member function to provide a custom functionality of the derived class.

```
// Vehicle class.

class Vehicle {
 public:
  Vehicle() {}
  void Accelerate();
  void Decelerate();

  LatLng GetLocation() const;
  double GetSpeed() const;
  double GetWeight() const;

 private:
  LatLng location_;
  double speed_;
  double weight_;
};
```

```
// Car class.
class Car : public Vehicle {
 public:
  Car() : Vehicle() {}

  int GetCapacity() const;

  // Override the parent's GetWeight().
  double GetWeight() const {
    return Vehicle::GetWeight() +
        passenger_weight_;
  }
 private:
  int capacity_;
  double passenger_weight_;
};
```

Overriding in CSStudent Example

```
#include <iostream>
using namespace std;

class Person {
 public:
  void Talk() {
    cout << "I'm a person" << endl;
  }
};

class Student : public Person {
 public:
  void Talk() {
    cout << "I'm a student" << endl;
  }

  void Study() {
    cout << "study" << endl;
  }
};
```

```
class CSStudent : public Student {
 public:
  void Talk() {
    cout << "I'm a CS student" << endl;
  }

  void WriteCode() {
    cout << "write_code" << endl;
  }
};

int main() {
  CSStudent csst;
  csst.Talk();
  // Output: "I'm a CS student"

  Person& person_csst = csst;
  person_csst.Talk();
  // Output: "I'm a person" ??

  return 0;
}
```

Why is `Person::talk()` called instead of `CSStudent::talk()`?

- By default, C++ compiler matches a function call with the correct function definition ***at compile time*** based on ***declared type*** (called ***static binding***).

- Base class pointers and references only know the base class members *at compile time*.

More Examples

```
int main() {
  Person p;
  Student st;
  CSStudent csst;

  Person& person_p = p;
  Person& person_st = st;
  Person& person_csst = csst;

  person_p.Talk();      // Person::Talk()
  person_st.Talk();     // Person::Talk()
  person_csst.Talk();   // Person::Talk()

  Student& student_st = st;
  Student& student_csst = csst;

  student_st.Talk();     // Student::Talk()
  student_csst.Talk();   // Student::Talk()

  return 0;
}
```

How to get polymorphic behavior?

- But this is not what we want!

- We often want to customize the behavior of the same member function in each derived class
  - so that we get different behaviors through the same interface → **Polymorphism!**

Like this:

```
Person& person_p = p;
Person& person_st = st;
Person& person_csst = csst;

person_p.Talk();      // Person::Talk()
person_st.Talk();     // Student::Talk()
person_csst.Talk();   // CSStudent::Talk()
```

Virtual Functions

- By declaring the member function **`virtual`**, you can do this!

```
virtual void Talk();
```

- Calling a virtual functions means:
- C++ compiler match a function call with the correct function definition ***at runtime*** based on ***actual type*** (called ***dynamic binding***).

Virtual Functions

- Virtual functions are keys to implement polymorphism in C++.
  - declare polymorphic member functions to be 'virtual',
  - and use the base class pointer to point an instance of the derived class,
  - then the function call from a base class pointer will execute the function overridden in the derived class.

- Where to specify ‘virtual’?
  - Actually, ‘virtual’ keyword is not necessary in the derived class.
  - But specifying ‘virtual’ for all virtual functions in descendant classes is recommended.

Virtual Function Example

```
// Vehicle classes.

class Vehicle {
 public:
  virtual void Accelerate() {
    cout << "Vehicle.Accelerate";
  }
};

class Car : public Vehicle {
 public:
  virtual void Accelerate() {
    cout << "Car.Accelerate";
  }
};

class Truck : public Vehicle {
 public:
  virtual void Accelerate();
    cout << "Truck.Accelerate";
  }
};
```

```
// Main routine.

int main() {
  Car car;
  Truck truck;
  Vehicle* pv = &car;
  pv->Accelerate();
  // Outputs Car.Accelerate.

  pv = &truck;
  pv->Accelerate();
  // Outputs Truck.Accelerate.

  Vehicle vehicle;
  pv = &vehicle;
  pv->Accelerate();
  // Outputs Vehicle.Accelerate.
  return 0;
}
```

Virtual Function Example (w/o virtual)

```cpp
// Vehicle classes.

class Vehicle {
 public:
  void Accelerate() {
    cout << "Vehicle.Accelerate";
  }
};

class Car : public Vehicle {
 public:
  void Accelerate() {
    cout << "Car.Accelerate";
  }
};

class Truck : public Vehicle {
 public:
  void Accelerate();
    cout << "Truck.Accelerate";
  }
};
```

```cpp
// Main routine.

int main() {
  Car car;
  Truck truck;

  Vehicle* pv = &car;
  pv->Accelerate();
  // Outputs Vehicle.Accelerate.
  car.Accelerate();
  // Outputs Car.Accelerate.

  pv = &truck;
  pv->Accelerate();
  // Outputs Vehicle.Accelerate.
  truck.Accelerate();
  // Outputs Truck.Accelerate.

  Vehicle vehicle;
  pv = &vehicle;
  pv->Accelerate();
  // Outputs Vehicle.Accelerate.
  return 0;
}
```

Virtual Functions in CSStudent Example

```
#include <iostream>
using namespace std;

class Person {
 public:
  virtual void Talk() {
    cout << "I'm a person" << endl;
  }
};

class Student : public Person {
 public:
  virtual void Talk() {
    cout << "I'm a student" << endl;
  }

  void Study() {
    cout << "study" << endl;
  }
};
```

```
class CSStudent : public Student {
 public:
  virtual void Talk() {
    cout << "I'm a CS student" << endl;
  }

  void WriteCode() {
    cout << "write_code" << endl;
  }
};

int main() {
  CSStudent csst;
  csst.Talk();
  // Output: "I'm a CS student"

  Person& person_csst = csst;
  person_csst.Talk();
  // Output: "I'm a CS student"

  return 0;
}
```

Another Example

```
void MakePersonTalk(Person* person) {
  person->Talk();
}

int main() {
  vector<Person*> people;
  people.push_back(new Person);
  people.push_back(new Person);
  people.push_back(new Student);
  people.push_back(new Student);
  people.push_back(new Person);
  people.push_back(new Student);
  people.push_back(new CSStudent);
  people.push_back(new CSStudent);

  for (int i = 0; i < people.size(); ++i) {
    MakePersonTalk(people[i]);
  }
  for (int i = 0; i < people.size(); ++i) {
    delete people[i];
  }
  return 0;
}
```

CSStudent Example w/o Virtual Functions

```
#include <iostream>
using namespace std;

class Person {
 public:
  void Talk() {
    cout << "I'm a person" << endl;
  }
};

class Student : public Person {
 public:
  void Talk()  {
    cout << "I'm a student" << endl;
  }

  void Study() {
    cout << "study" << endl;
  }
};
```

```
class CSStudent : public Student {
 public:
  void Talk() {
    cout << "I'm a CS student" << endl;
  }

  void WriteCode() {
    cout << "write_code" << endl;
  }
};

int main() {
  CSStudent csst;
  csst.Talk();
  // Output: "I'm a CS student"

  Person& person_csst = csst;
  person_csst.Talk();
  // Output: "I'm a person"

  return 0;
}
```

Quiz #2

```cpp
#include <iostream>
using namespace std;

class Person {
 public:
  virtual void Talk() { cout << "a "; }
};

class Student : public Person {
 public:
  void Talk() { cout << "b "; }
};

class CSStudent : public Student {
 public:
  void Talk() { cout << "c "; }
};

class Faculty : public Person {
 public:
  void Talk() { cout << "d "; }
};
```

```cpp
int main() {
  Person ps;
  Student st;
  CSStudent csst;
  Faculty fc;

  Person* p1 = &ps;
  Person& p4 = fc;
  Student* s2 = &csst;

  p1->Talk();
  p4.Talk();
  s2->Talk();
}
```

- What is the expected output of this program? (If a compile error is expected, just write down "error").

Destructor and Virtual

```
class A {
public:
  A() { cout << " A" << endl; }
  ~A() { cout << " ~A" << endl; }
};

class AA : public A {
public:
  AA() { cout << " AA" << endl; }
  ~AA() { cout << " ~AA" << endl; }
};

int main() {
  AA* pa = new AA;   // OK: prints ' A AA'.
  delete pa;        // prints ' ~AA ~A'.
  return 0;
}
```

Destructor and Virtual

- What happens if a derived class object is **'deleted' by its base class pointer?**

```
class A {
public:
  A() { cout << " A"; }
  ~A() { cout << " ~A"; }
};

class AA : public A {
public:
  AA() { cout << " AA"; }
  ~AA() { cout << " ~AA"; }
};

int main() {
  A* pa = new AA;   // OK: prints ' A AA'.
  delete pa;        // Hmm..: prints only ' ~A'.
  return 0;
}
```

Virtual Destructor

- What happens if a derived class object is **'deleted' by its base class pointer?**

- If the base class destructor **is not `virtual`**,
  - only the base class destructor is called
  - the derived class destructor is **not** called

- **This may cause memory leak**
  - Think about this case: A derived class destructor has the code that `delete` its member variables which are assigned by `new` in its constructor

```cpp
#include <iostream>
using namespace std;

class Shape {
 public:
  Shape() {}
  ~Shape() {}
};

class Rectangle : public Shape {
 private:
  int* width;
  int* height;

 public:
  Rectangle() {
    width = new int;
    height = new int;
    cout << "Rectangle()" << endl;
  }

  ~Rectangle() {
    delete width;
    delete height;
    cout << "~Rectangle()" << endl;
  }
};
```

```cpp
int main() {
  Shape* shape1 = new Rectangle;
  delete shape1;

  return 0;
}
```

Virtual Destructor

- What happens if a derived class object is **'deleted' by its base class pointer?**

- If the base class destructor **is `virtual`**,
  - **the derived class destructor is called**
  - and then base class destructors is called (reverse order of constructor calls)

```cpp
#include <iostream>
using namespace std;

class Shape {
 public:
  Shape() {}
  virtual ~Shape() {}
};

class Rectangle : public Shape {
 private:
  int* width;
  int* height;

 public:
  Rectangle() {
    width = new int;
    height = new int;
    cout << "Rectangle()" << endl;
  }

  virtual ~Rectangle() {
    delete width;
    delete height;
    cout << "~Rectangle()" << endl;
  }
};
```

```cpp
int main() {
  Shape* shape1 = new Rectangle;
  delete shape1;

  return 0;
}
```

When do we need a virtual destructor?

- A destructor of a base class **should be** `virtual` if
  - its descendant class instance is **deleted by the base class pointer.** (..or)
  - any of member function is virtual (which means it's a polymorphic base class).

```
class A {
public:
  A() { cout << " A"; }
  virtual ~A() { cout << " ~A"; }
};

class AA : public A {
public:
  AA() { cout << " AA"; }
  virtual ~AA() { cout << " ~AA"; }
};

int main() {
  A* pa = new AA;   // OK: prints ' A AA'.
  delete pa;        // OK: prints ' ~AA ~A'.
  return 0;
}
```

Virtual Destructor

- Note that constructors cannot be `virtual`
  - "virtual" allows us to call a function knowing only an interfaces and not the exact type of the object.
  - But to create an object, you need to know the exact type of what you want to create.
  - Bjarne Stroustrup's C++ Style and Technique FAQ: [Why don't we have virtual constructors?]

Quiz #3

```cpp
#include <iostream>
using namespace std;

class Person {
 public:
  virtual ~Person() { cout << "a "; }
};

class Student : public Person {
 public:
  ~Student() { cout << "b "; }
};

class CSStudent : public Student {
 public:
  ~CSStudent() { cout << "c "; }
};

class Faculty : public Person {
 public:
  ~Faculty() { cout << "d "; }
};
```

```cpp
int main() {
  Person* p1 = new Faculty;
  Person* p2 = new CSStudent;
  delete p2;
  delete p1;
}
```

- What is the expected output of this program? (If a compile error is expected, just write down "error").

CAUTION: Copying a derived class object to a base class object

```
class Animal{
public:
    virtual void MakeSound() { cout << "(none)" << endl; }
};

class Dog : public Animal{
public:
    virtual void MakeSound() { cout << "bark" << endl; }
};

int main() {
    Animal animal;
    animal.MakeSound();  // "(none)"

    Dog dog;
    dog.MakeSound();  // "bark"

    // A typical way for polymorphism
    Animal& good dog = dog;
    goodDog.MakeSound();  // "bark"

    // ???
    Animal bad dog = dog;
    badDog.MakeSound();  // "(none)"
}
```

CAUTION: Avoid Object Slicing

- In C++, **object slicing** occurs when a derived class object is copied to a base class object.
  - Additional attributes of a derived class object are “sliced off”

```
class Base { int x, y; };

class Derived : public Base { int z, w; };

int main() {
    Derived d;
    Base b = d; // Object Slicing,  z and w of d are sliced off
}
```

- Note that **C++ polymorphism** works only with references or pointers, **not with objects.**

Next Time

- Next lecture:
  - 10 - Polymorphism 2