

---

# **Creative Software Design**

## **9 – Polymorphism 1**

Yoonsang Lee

Fall 2020

# 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 the same 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 to work 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;
    Dog dog;
    letsHear(cat);
    letsHear(dog);
    return 0;
}
```

# Pointers, References and Inheritance

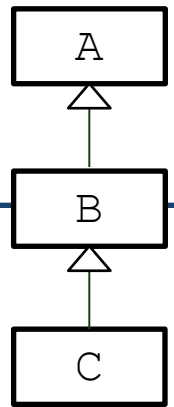
---

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





```
#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 << "writeCode" << endl;
    }
};
```

```
int main()
{
    Student* s1 = new Person; // error
    // A Person CANNOT be regarded as a Student.

    Student* s2 = new Student;
    // A Student is regarded as a Student

    Student* s3 = new CSStudent;
    // A CSStudent is regarded as a Student

    Person* p1 = new Person;
    Person* p2 = new Student;
    Person* p3 = new CSStudent;

    delete p1;
    delete p2;
    delete p3;

    delete s1;
    delete s2;
    delete s3;

    return 0;
}
```

```
#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 << "writeCode" << endl;
    }
};
```

```
int main()
{
    Student st;

    Person* person_st = &st; // ok
    // A Student is regarded as a Person.

    Student* student_st = &st; // ok
    // A Student is regarded as a Student.

    CSStudent* csstudent_st = &st; //error!
    // A Student CANNOT be regarded as a CSStudent.

    CSStudent csst;

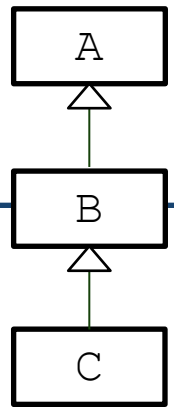
    Person* person_csst = &csst; // ok
    Student* student_csst = &csst; // ok
    CSStudent* csstudent_csst = &csst; //ok

    return 0;
}
```

# Pointers with Inheritance

---

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



```

#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 << "writeCode" << endl;
    }
};

```

```

int main()
{
    Student st;
    Person* person_st = &st;
    // A Student is regarded as a Person.

    person_st->talk();
    person_st->study(); // error!
    person_st->writeCode(); // error!
    // You cannot call them because not
all Persons are Students or CSStudents.

    return 0;
}

```

```

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

    student_st->talk();
    student_st->study();
    student_st->writeCode(); // error!

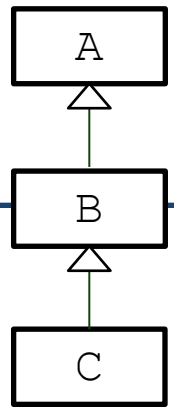
    return 0;
}

```

# References with Inheritance

---

- A class (B) reference can refer to
  - its own class (B) object
  - its derived class (C) object
  - CANNOT refer to its base class (A) object
  
- Exactly the same as the pointers!



```
#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 << "writeCode" << endl;
    }
};
```

```
int main()
{
    Student st;

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

    CSStudent csst;

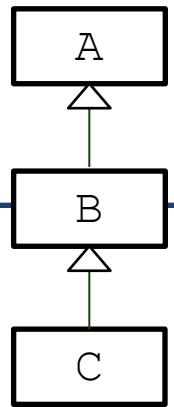
    Person& person_csst = csst; // ok
    Student& student_csst = csst; // ok
    CSStudent& csstudent_csst = csst; //ok

    return 0;
}
```

# References with Inheritance

---

- A class (B) reference can access
  - the members of its base class (A)
  - the members of its own class (B)
  - CANNOT access the members of its derived class (C)
  
- Exactly the same as the pointers!



```

#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 << "writeCode" << endl;
    }
};

```

```

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

    person_st.talk();
    person_st.study(); // error!
    person_st.writeCode(); // error!

    return 0;
}

```

```

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

---

- Go to <https://www.slido.com/>
- Join #csd-hyu
- Click "Polls"
  
- Submit your answer in the following format:
  - **Student ID: Your answer**
  - e.g. **2017123456: 4)**
  
- Note that you must submit all quiz answers in the above format to be checked as "attendance".

# 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 << "writeCode" << 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 / reference to refer an instance of the derived class,
  - then the function call from a base class pointer / reference 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)

```
// 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";
    }
};
```

```
// 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 << "writeCode" << 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;
}
```

# Quiz #2

---

- Go to <https://www.slido.com/>
- Join #csd-hyu
- Click "Polls"
  
- Submit your answer in the following format:
  - **Student ID: Your answer**
  - e.g. **2017123456: 4)**
  
- Note that you must submit all quiz answers in the above format to be checked as "attendance".

# 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

```
#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;
    }
};
```

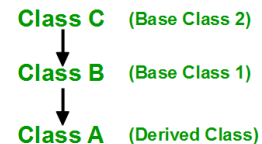
```
int main()
{
    Shape* shapel = new Rectangle;
    delete shapel;

    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)

## Order of Inheritance



## Order of Constructor Call

1. C() (Class C's Constructor)
2. B() (Class B's Constructor)
3. A() (Class A's Constructor)

## Order of Destructor Call

1. ~A() (Class A's Destructor)
2. ~B() (Class B's Destructor)
3. ~C() (Class C's Destructor)

```
#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;
    }
};
```

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

    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 interface 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

---

- Go to <https://www.slido.com/>
- Join #csd-hyu
- Click "Polls"
  
- Submit your answer in the following format:
  - **Student ID: Your answer**
  - e.g. **2017123456: 4)**
  
- Note that you must submit all quiz answers in the above format to be checked as "attendance".



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

```
#include <iostream>
using namespace std;
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& goodDog = dog;
    goodDog.makeSound(); // "bark"

    // ???
    Animal badDog = 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

---

- Labs in this week:
  - Lab1: Assignment 9-1
  - Lab2: Assignment 9-2
  
- Next lecture:
  - 10 - Polymorphism 2