Polymorphism -
Interface and Virtual Functions

한양대학교 컴퓨터소프트웨어학부

2017년 2학기

Polymorphism

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

- A common interface for different types of objects.
- Real-world examples (in functionality):
  - Steering wheel + accelerator + brake in cars.
  - Volume control + channel control in TV remotes.
  - Shutter button for film or digital cameras.
- Message passing mechanism.

Polymorphism and Class Hierarchy

- The parent class has common properties and functionalities of the child classes.
  - Public functions in the base class defines an interface.

```
// Vehicle class.

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

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

```
// Car and truck class.

class Car : public Vehicle {
  // ...
};

class Truck : public Vehicle {
  // ...
};

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

Polymorphism and Class Hierarchy

- Public functions in the base class defines an interface.
- Problem happens when the child classes overrides the parent's interface functions.

```
// Vehicle, Car, and Truck class.

class Vehicle {
 public:
  void Accelerate();  // A
  // ...
};

class Car : public Vehicle {
 public:
  void Accelerate() {  // B
    // Operation specific to cars.
  }
  // ...
};
```

```
class Truck : public Vehicle {
 public:
  void Accelerate() {  // C
    // Operation specific to trucks.
  }
  // ...
};

int main() {
  Car car;
  Truck truck;
  Vehicle* pv = &car;
  if (...) pv = &truck;
  pv->Accelerate();  // A, B, or C?
  ...
}
```

Virtual Functions

Virtual functions are keys to implement polymorphism in C++.

1. Declare polymorphic member functions to be `'virtual'`.
2. Use the base class pointer to point an instance of the derived class.
3. The function call from a base class pointer will execute the function overridden in its own class definition.

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

```
// 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 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 Destructor

What happens if an object is 'deleted' by its base class pointer?

```
struct A {
  A() { cout << " A"; }
  ~A() { cout << " ~A"; }
};

struct AA : public A {
  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

A destructor of a base class can be, and should be virtual if

- its descendant class instance is deleted by the base class pointer.
- any of member function is virtual.

```
struct A {
  A() { cout << " A"; }
  virtual ~A() { cout << " ~A"; }
};

struct AA : public A {
  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 Destructors

- Recall: destructors needed to de-allocate dynamically allocated data
- Consider:

  Base *pBase = new Derived;

  …

  delete pBase;

  - Would call base class destructor even though pointing to Derived class object!
  - Making destructor ***virtual*** fixes this!
- Good policy for all destructors to be virtual

Casting

- Consider:

```
Pet vpet;
Dog vdog;
...
vdog = static_cast<Dog>(vpet);  //ILLEGAL!
```

- Can’t cast a pet to be a dog, but:

```
vpet = vdog;          // Legal!
vpet = static_cast<Pet>(vdog);  //Also legal!
```

- Upcasting is OK
  - From descendant type to ancestor type

Downcasting

- Downcasting dangerous!
  - Casting from ancestor type to descended type
  - Assumes information is "added"
  - Can be done with dynamic_cast:

    ```
    Pet *ppet;
    ppet = new Dog;
    Dog *pdog = dynamic_cast<Dog*>(ppet);
    ```
    - Legal, but dangerous!
- Downcasting rarely done due to pitfalls
  - Must track all information to be added
  - All member functions must be virtual

Pure Virtual Function

- What if you cannot define the base class' member function? (no 'default' behavior)

```
// Shape classes.

struct Shape {
  virtual void Draw() const {
    // What should we do here?
  }
};

struct Rectangle : public Shape {
  virtual void Draw() const {
    // Draw a rectangle.
  }
};

struct Triangle : public Shape {
  // What if we forget to override
  // Draw() here?
};
```

```
int main() {
  vector<Shape*> v;
  v.push_back(new Rectangle);
  v.push_back(new Triangle);

  for (int i = 0; i < v.size(); ++i) {
    v[i]->Draw();
  }
  for (int i = 0; i < v.size(); ++i) {
    delete v[i];
  }
  return 0;
}
```

Pure Virtual Function

- Pure virtual functions cannot have definitions.
- Pure virtual functions should be overridden.

```
// Shape classes.

struct Shape {
  // Pure virtual Draw function.
  virtual void Draw() const = 0;
};

struct Rectangle : public Shape {
  virtual void Draw() const {
    // Draw a rectangle.
  }
};

struct Triangle : public Shape {
  // What if we forget to override
  // Draw() here? => Error!
};
```

```
int main() {
  vector<Shape*> v;
  v.push_back(new Rectangle);
  v.push_back(new Triangle);

  for (int i = 0; i < v.size(); ++i) {
    v[i]->Draw();
  }
  for (int i = 0; i < v.size(); ++i) {
    delete v[i];
  }
  return 0;
}
```

Pure Virtual Functions

- Base class might not have "meaningful" definition for some of it’s members!
  - It’s purpose solely for others to derive from
- Recall class Figure
  - All figures are objects of derived classes
    - Rectangles, circles, triangles, etc.
  - Class Figure has no idea how to draw!
- Make it a pure virtual function:
  virtual void draw() = 0;

Abstract Base Classes

- Pure virtual functions require no definition
  - Forces all derived classes to define "their own" version
- Class with one or more pure virtual functions is: abstract base class
  - Can only be used as base class
  - No objects can ever be created from it
    - Since it doesn't have complete "definitions" of all it's members!
- If derived class fails to define all pure's:
  - It's an abstract base class too

Overriding

- Virtual function definition changed in a derived class
  - We say it's been "overidden"
- Similar to redefined
  - Recall: for standard functions
- So:
  - Virtual functions changed: ***overridden***
  - Non-virtual functions changed: ***redefined***

Virtual Functions: Why Not All?

- Clear advantages to virtual functions as we've seen
- One major disadvantage: overhead!
  - Uses more storage
  - Late binding is "on the fly", so programs run slower
- So if virtual functions not needed, should not be used

Virtual: How?

- To write C++ programs:
  - Assume it happens by "magic"!
- But explanation involves late binding
  - Virtual functions implement late binding
  - Tells compiler to "wait" until function is used in program
  - Decide which definition to use based on calling object
- Very important OOP principle!

Interface Class

An interface class is a class only with pure virtual functions.

- A design pattern.
- No member variables or non-virtual functions.
- Defines an interface to a service -
  what does the class do, and how it should be used.

```
struct Shape {
  virtual ~Shape() {}
  virtual void Draw() const = 0;
  virtual int GetArea() const = 0;
  virtual void MoveTo(int x, int y) = 0;
};

void DrawShapes(const vector<Shape*>& v) {
  for (int i = 0; i < v.size(); ++i) v[i]->Draw();
}
```


Q&A