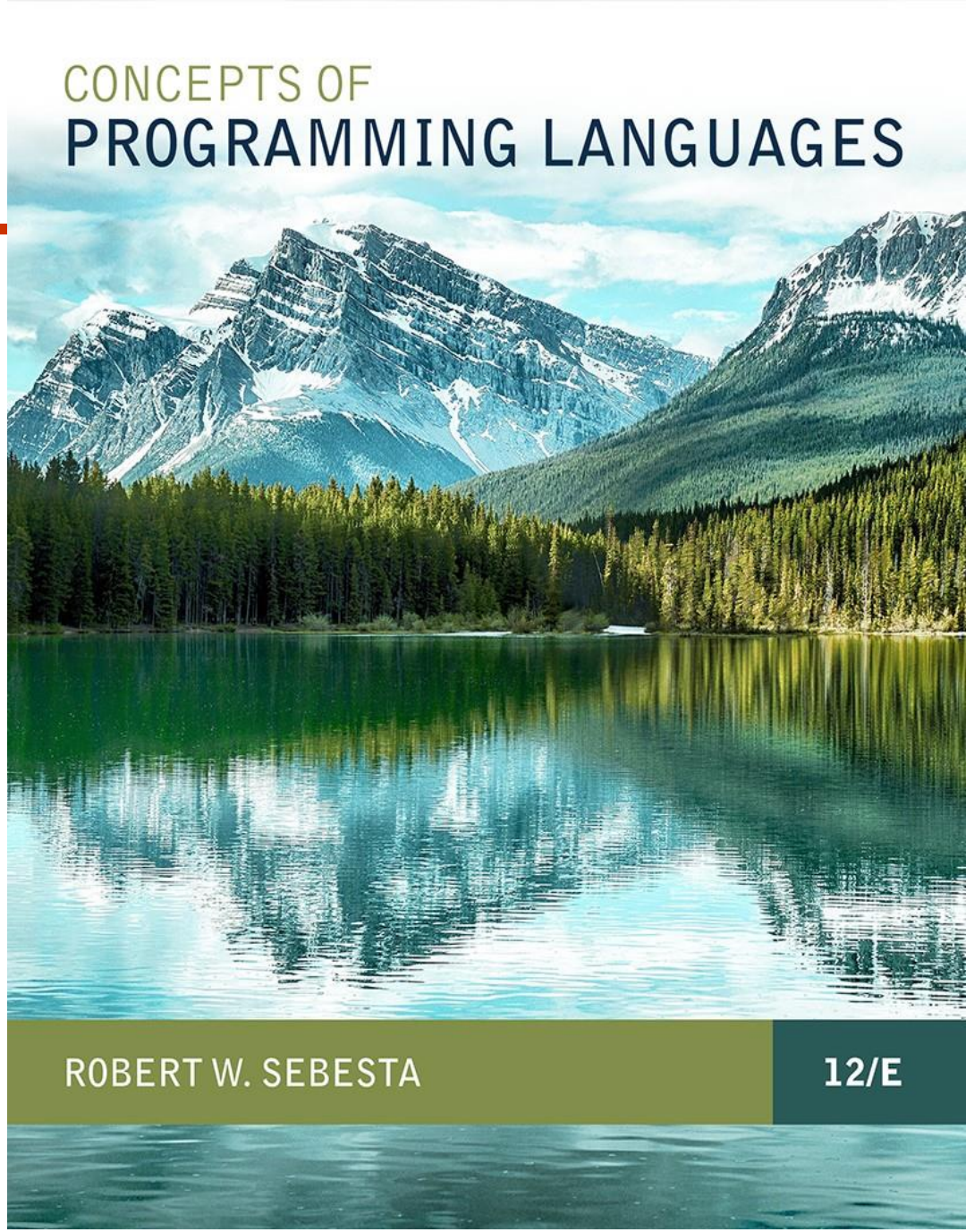
Chapter 12

Support for Object-Oriented Programming



ISBN 0-321-49362-1

Chapter 12 Topics

- 소개
- 객체지향 프로그래밍
- 객체지향언어의 설계고려사항
- C++에서 객체지향 프로그래밍
- Java에서 객체지향 프로그래밍
- 객체지향 구조의 구현

서론

- 다양한 종류의 객체지향 프로그래밍(OOP) 언어
  - 절차형 언어이면서 OOP 언어 (e.g., C++)
  - 함수형 프로그램 지원 (e.g., CLOS)
  - 다른 패러다임은 지원하지 않으나 명령형구조를 사용 (e.g., Java, C#)
  - 순수 OOP 언어 (e.g., Smalltalk, Ruby)
  - OOP를 지원하는 몇몇 함수형 언어

객체지향 프로그래밍

- 세가지 중요한 언어의 특징:
  - Abstract data types (Chapter 11)
  - Inheritance(상속)
    - OOP의 핵심
  - Polymorphism(다형성)

상속(Inheritance)

- 재사용은 생산성 증진을 가져올 수 있음
  - ADTs는 재사용이 어려움 – 항상 수정 필요
  - 모든 ADTs는 독립적이며 동일 수준 – 프로그램 조직화 필요
- 상속은 현재의 클래스로 새로운 클래스를 정의하도록 허용, 즉 공통 부분은 상속하게 함으로써
- 상속은 위 두 문제(수정문제와 프로그램 조직화 문제)의 해결책임
  - reuse ADTs after minor changes
  - define classes in a hierarchy

Object-Oriented Concepts

- ADT : 클래스
- 클래스 사례(class instances) : 객체
- 상속된 클래스 : 파생클래스, 서브클래스
- 새 클래스를 유도한 클래스 : 부모클래스, 수퍼 클래스
- 객체 상에 연산을 정의한 부프로그램 : 매소드
- 메소드 호출 : 메시지
- 객체의 전체 매소드 : 메시지 프로토콜/인터페이스
- 메시지는 두 부분으로 구성 : 메소드 이름과 목표 객체

Object-Oriented Concepts (continued)

- 상속은 캡슐화된 개체의 접근제어로 복잡해짐
  - 클래스는 그 서브클래스로부터 개체들을 은폐
  - 클래스는 그 클라이언트로부터 개체들을 은폐
  - 서브클래스는 읽기가능하나 클라이언트에게는 은폐
- 메소드는 상속되며, 상속된 메소드를 수정 가능
  - 새 메소드는 상속된 메소드를 재정의(*override*)
  - 부모클래스의 메소드는 재정의됨(*overridden*)

Object-Oriented Concepts (continued)

- 파생클래스가 부모클래스와 달라질 수 있는 방법
  1. 파생클래스는 변수나 메소드를 추가할 수 있음
  2. 파생클래스는 상속된 메소드의 동작을 수정 가능

     수정된 메소드는 동일한 이름을 갖고, 동일한 프로토콜을 가질 수도 있음
  3. 부모클래스가 **private** 접근의 변수나 메소드를 정의하면 이들은 파생클래스에서 비가시적임

Object-Oriented Concepts (continued)

- 클래스의 두 가지 변수 :
  - *클래스 변수* – one/class
  - *사례 변수* – one/object
- 클래스의 두 가지 메소드 :
  - *클래스 메소드* – 클래스에 연산 수행
  - *사례 메소드* – 객체에 연산을 수행
- 단일 상속(유도 트리)과 다중 상속(유도 그래프)
- 상속의 한가지 단점 :
  - 클래스간의 상호종속성을 생성 → 유지관리의 복잡성

동적 바인딩(Dynamic Binding)

- 메시지를 메소드 정의에 동적 바인딩 함으로써 제공되는 다형성
- 다형 변수는 클래스에 정의하며 클래스의 객체와 그 후손의 객체를 가리킬 수 있다.
- 클래스 계층이 메소드를 재정의하는 클래스를 포함하고 그 메소드는 다형변수에 의해 호출될 때, 정확한 메소드로 동적 바인딩

동적 바인딩 개념

- 추상 메소드(*abstract method*) :
  - 메소드 정의를 하지 않음
  - 단지 프로토콜만 정의
- 추상 클래스(*abstract class*) : 1개 이상의 가상 메소드(virtual method)를 포함
- 추상 클래스는 사례화될 수 없다.

객체-지향언어의 설계고려사항

- The Exclusivity of Objects
- Are Subclasses Subtypes?
- Single and Multiple Inheritance
- Object Allocation and Deallocation
- Dynamic and Static Binding
- Nested Classes
- Initialization of Objects

객체의 배타성

- 모든 것이 객체인 경우
  - 장점 – elegance and purity
  - 단점 – slow operations on simple objects
- 완벽한 명령형 타입 시스템에 객체를 추가하는 경우
  - 장점 – fast operations on simple objects
  - 단점 – results in a confusing type system (two kinds of entities)
- 기본형 스칼라 타입에 대해서 명령형 타입 구조 + 그 외 모든 구조적 타입은 객체로 구현
  - 장점 – fast operations on simple objects and a relatively small typing system
  - 단점 – still some confusion because of the two type systems

Are Subclasses Subtypes?

- Does an "is-a" relationship hold between a parent class object and an object of the subclass?
  - If a derived class is-a parent class, then objects of the derived class must behave the same as the parent class object
- A derived class is a subtype if it has an is-a relationship with its parent class
  - Subclass can only add variables and methods and override inherited methods in "compatible" ways

단일 상속과 다중 상속

- Multiple inheritance allows a new class to inherit from two or more classes
- Disadvantages of multiple inheritance:
  - Language and implementation complexity (in part due to name collisions)
  - Potential inefficiency – dynamic binding costs more with multiple inheritance (but not much)
- Advantage:
  - Sometimes it is quite convenient and valuable

객체의 할당과 해제

- 객체가 어디에 할당되는가?
  - 만약 객체가 ADT 처럼 동작한다면, 어디든 할당가능
    - Allocated from the run-time stack
    - Explicitly create on the heap (via `new`)
  - 만약 모두 힙-동적이라면, 포인터나 참조변수를 통해 획일적인 방법으로 참조
    - Simplifies assignment – dereferencing can be implicit
  - 만약 모두 스택-동적이라면, 부타입과 관련된 문제가 존재 → *object slicing* (그림 12.4)
- 해제는 explicit or implicit?
  - 해제가 명시적이라면 허상포인터/허상참조의 문제

동적 바인딩과 정적 바인딩

- 메소드에 대한 메시지의 바인딩이 모두 동적이어야 하는가?
  - If none are, you lose the advantages of dynamic binding
  - If all are, it is inefficient
- 사용자가 동적/정적으로 지정하도록 허용해야 한다.

중첩 클래스

- If a new class is needed by only one class, there is no reason to define so it can be seen by other classes
  - Can the new class be nested inside the class that uses it?
  - In some cases, the new class is nested inside a subprogram rather than directly in another class
- Other issues:
  - Which facilities of the nesting class should be visible to the nested class and vice versa

Initialization of Objects

- Are objects initialized to values when they are created?
  - Implicit or explicit initialization
- How are parent class members initialized when a subclass object is created?

Support for OOP in C++

- General Characteristics:
  - Evolved from C and SIMULA 67
  - Among the most widely used OOP languages
  - Mixed typing system
  - Constructors and destructors
  - Elaborate access controls to class entities

Support for OOP in C++ (continued)

- Inheritance
  - A class need not be the subclass of any class
  - Access controls for members are
    - Private (visible only in the class and friends) (disallows subclasses from being subtypes)
    - Public (visible in subclasses and clients)
    - Protected (visible in the class and in subclasses, but not clients)

Support for OOP in C++ (continued)

- In addition, the subclassing process can be declared with access controls (private or public), which define potential changes in access by subclasses
  - Private derivation – inherited public and protected members are private in the subclasses
  - Public derivation public and protected members are also public and protected in subclasses

Inheritance Example in C++

```
class base_class {
  private:
    int a;
    float x;
  protected:
    int b;
    float y;
  public:
    int c;
    float z;
};

class subclass_1 : public base_class { ... };
//      In this one, b and y are protected and
//      c and z are public

class subclass_2 : private base_class { ... };
//     In this one, b, y, c, and z are private,
//     and no derived class has access to any
//     member of base_class
```

재반출(Reexportation) in C++

- A member that is not accessible in a subclass (because of private derivation) can be declared to be visible there using the scope resolution operator (: :), e.g.,

```
class subclass_3 : private base_class {
        base_class :: c;
        …
}
```

Reexportation (continued)

- One motivation for using private derivation
  - A class provides members that must be visible, so they are defined to be public members; a derived class adds some new members, but does not want its clients to see the members of the parent class, even though they had to be public in the parent class definition

Support for OOP in C++ (continued)

- Multiple inheritance is supported
  - If there are two inherited members with the same name, they can both be referenced using the scope resolution operator (::)

```
class Thread { ... }
class Drawing { ... }
class DrawThread : public Thread, public Drawing
  { … }
```

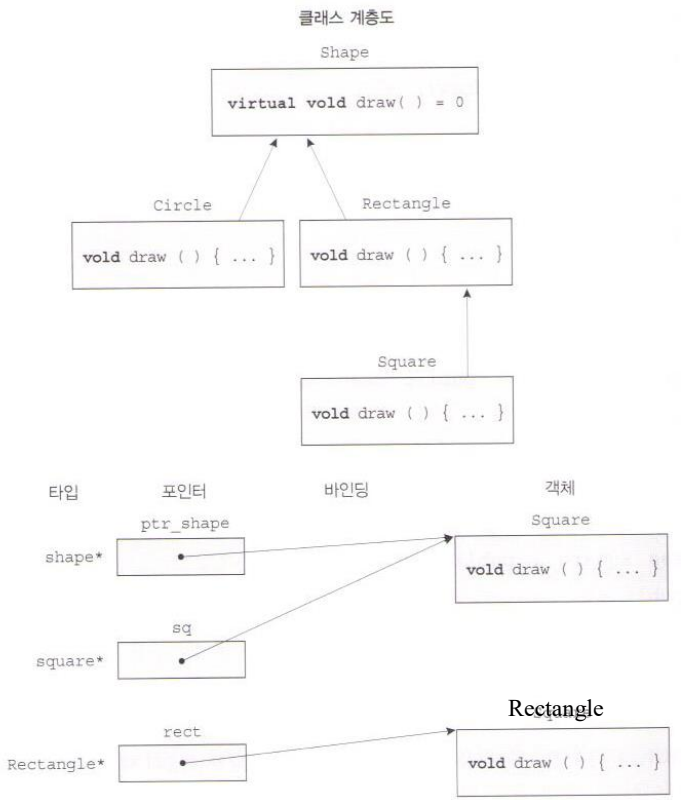
Support for OOP in C++ (continued)

- Dynamic Binding
  - A method can be defined to be `virtual`, which means that they can be called through polymorphic variables and dynamically bound to messages
  - A pure virtual function has no definition at all
  - A class that has at least one pure virtual function is an *abstract class*

Support for OOP in C++ (continued)

```
class Shape {
  public:
    virtual void draw() = 0;
    ...
};
class Circle : public Shape {
  public:
    void draw() { ... }
  ...
};
class Rectangle : public Shape {
  public:
    void draw() { ... }
  ...
};
class Square : public Rectangle {
  public:
    void draw() { ... }
  ...
};
```

```
Square* sq = new Square;
Rectangle* rect = new Rectangle;
Shape* ptr_shape;
ptr_shape = sq;  // points to a Square
ptr_shape ->draw();  // Dynamically
                     // bound to draw in Square
rect->draw();  // Statically bound to
               // draw in Rectangle
```



Support for OOP in C++ (continued)

- If objects are allocated from the stack, it is quite different

```
Square sq;     // Allocates a Square object from the stack
Rectangle rect;  // Allocates a Rectangle object from the stack
rect = sq;    // Copies the data member values from sq object
rect.draw();  // Calls the draw from Rectangle
```

Support for OOP in C++ (continued)

- Evaluation
  - C++ provides extensive access controls (unlike Smalltalk)
  - C++ provides multiple inheritance
  - In C++, the programmer must decide at design time which methods will be statically bound and which must be dynamically bound
    - Static binding is faster!
  - Smalltalk type checking is dynamic (flexible, but somewhat unsafe)
  - Because of interpretation and dynamic binding, Smalltalk is ~10 times slower than C++

Support for OOP in Java

- Because of its close relationship to C++, focus is on the differences from that language
- General Characteristics
  - All data are objects except the primitive types
  - All primitive types have wrapper classes that store one data value
  - All objects are heap-dynamic, are referenced through reference variables, and most are allocated with **`new`**
  - A **`finalize`** method is implicitly called when the garbage collector is about to reclaim the storage occupied by the object

Support for OOP in Java (continued)

- Inheritance
  - Single inheritance supported only, but there is an abstract class category that provides some of the benefits of multiple inheritance (**`interface`**)
  - An interface can include only method declarations and named constants, e.g.,

```
public interface Comparable <T> {
        public int comparedTo (T b);
}
```

  - Methods can be **`final`** (cannot be overriden)

Support for OOP in Java (continued)

- Dynamic Binding
  - In Java, all messages are dynamically bound to methods, unless the method is `final` (i.e., it cannot be overriden, therefore dynamic binding serves no purpose)
  - Static binding is also used if the methods is `static` or `private` both of which disallow overriding

Support for OOP in Java (continued)

- Nested Classes
  - All are hidden from all classes in their package, except for the nesting class
  - Nested classes can be anonymous
  - A *local nested class* is defined in a method of its nesting class
    - No access specifier is used

Support for OOP in Java (continued)

- Evaluation
  - Design decisions to support OOP are similar to C++
  - No support for procedural programming
  - No parentless classes
  - Dynamic binding is used as "normal" way to bind method calls to method definitions
  - Uses interfaces to provide a simple form of support for multiple inheritance

Support for OOP in C#

- Evaluation
  - C# is a relatively recently designed C-based OO language
  - The differences between C#'s and Java's support for OOP are relatively minor

Table 12.1 Designs

DESIGN ISSUE/ LANGUAGE	SMALLTALK	C++	OBJECTIVE-C	JAVA	C#	RUBY
Exclusivity of objects	All data are objects	Primitive types plus objects	Primitive types plus objects	Primitive types plus objects	Primitive types plus objects	All data are objects
Are subclasses subtypes?	They can be and usually are	They can be and usually are if the derivation is public	They can be and usually are	They can be and usually are	They can be and usually are	No subclasses are subtypes
Single and multiple inheritance	Single only	Both	Single only, but some effects with protocols	Single only, but some effects with interfaces	Single only, but some effects with interfaces	Single only, but some effects with modules
Allocation and deallocation of objects	All objects are heap allocated; allocation is explicit and deallocation is implicit	Objects can be static, stack dynamic, or heap dynamic; allocation and deallocation are explicit	All objects are heap dynamic; allocation is explicit and deallocation is implicit	All objects are heap dynamic; allocation is explicit and deallocation is implicit	All objects are heap dynamic; allocation is explicit and deallocation is implicit	All objects are heap dynamic; allocation is explicit and deallocation is implicit
Dynamic and static binding	All method bindings are dynamic	Method binding can be either	Method binding can be either	Method binding can be either	Method binding can be either	All method bindings are dynamic
Nested classes?	No	Yes	No	Yes	Yes	Yes
Initialization	Constructors must be explicitly called	Constructors are implicitly called	Constructors must be explicitly called	Constructors are implicitly called	Constructors are implicitly called	Constructors are implicitly called

객체-지향 구조의 구현

- Two interesting and challenging parts
  - Storage structures for instance variables
  - Dynamic binding of messages to methods

사례 데이터 기억장소

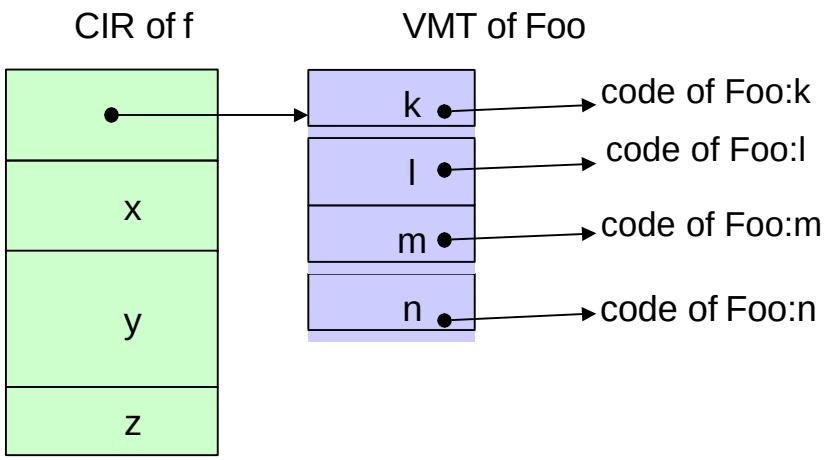
- Class instance records (CIRs) store the state of an object
  - Static (built at compile time)
- If a class has a parent, the subclass instance variables are added to the parent CIR
- Because CIR is static, access to all instance variables is done as it is in records
  - Efficient

메소드 호출의 동적 바인딩

- Methods in a class that are statically bound need not be involved in the CIR; methods that will be dynamically bound must have entries in the CIR
  - Calls to dynamically bound methods can be connected to the corresponding code thru a pointer in the CIR
  - The storage structure is sometimes called *virtual method tables* (vtable)
  - Method calls can be represented as offsets from the beginning of the vtable

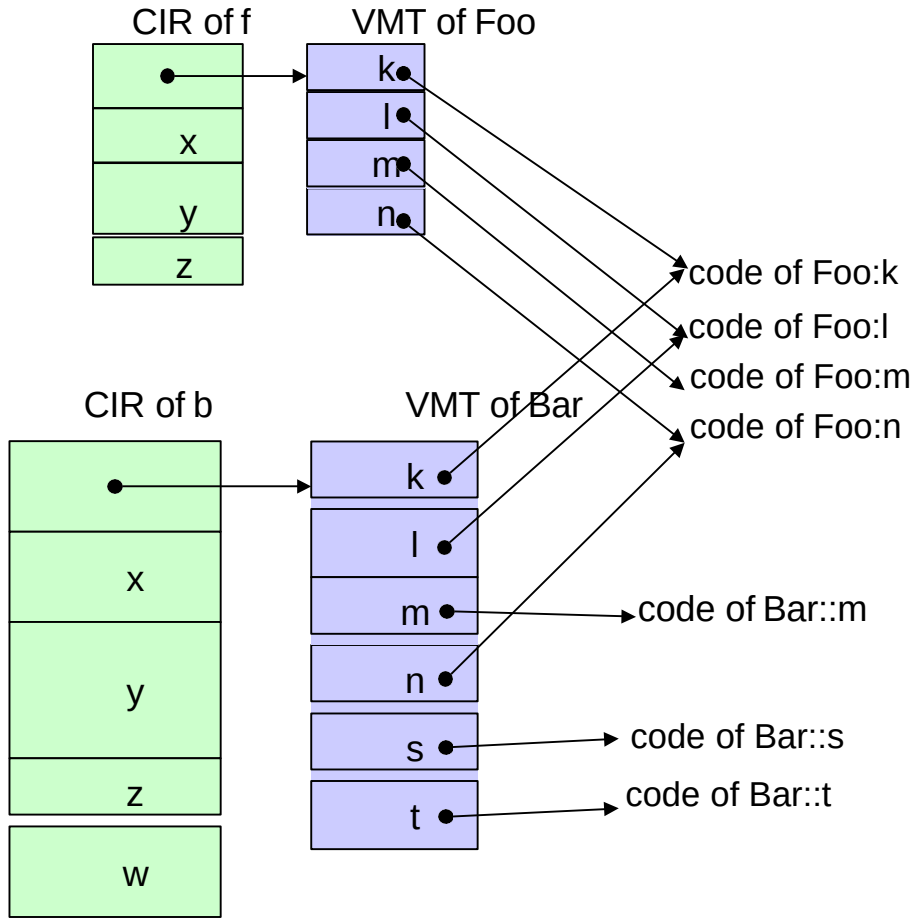
Object and method table

```
class Foo {  int x;  double y;  char z;
public:
  virtual void k ( ... );
  virtual int l ( ... );
  virtual void m ( ... );
  virtual double n ( ... );
...
};
Foo f;
```



Implementing single inheritance

```
class Bar: public Foo {
  int w;
public:
  virtual void m ( ... ); // = override
  virtual double s ( ... );
  virtual char *t ( ... );
   ...
};
Bar b;
```



정적, 동적 바인딩 언어

- 모든 연산에 동적 바인딩
  - Python, Smalltalk
- 디폴트로 동적 바인딩
  - Java는 final keyword로 정적 바인딩
- 디폴트로 정적 바인딩
  - C++, virtual과 override(C++11)로 동적 바인딩
  - C++11, final로 overriding 금지

C++11에서 override와 final 키워드 사용 예

```
class Base {
public:
    Base();
    virtual int fun(int nVal);
    int funNormal(int nVal);
};
class Derived : public Base {
public:
    virtual int fun(float nValue) override;   // error!
    int funNormal(int nVal) override;      // error!
};

class Base1 final  {
public:
    Base1();
};
class Derived1 : public Base1 {      // error!
};

class Base2 {
public:
    Base2();
    virtual int fun() final;
};
class Derived2 : public Base2 {
public:
    Derived2();
    virtual int fun();      // error!
};
```