# 09장 자료 추상화와 객체지향

## 9.5 다중 상속

**예 9.50**
두 기본 클래스에서 유도하기 (반복)

간단한 C++ 예를 다시 보자.

```
class student : public person, public gp_list_node { ...
```

다중 상속을 구현하기 위해 student 객체로의 참조를 person이나 gp_list_node 변수에 대입할 때 필요할 때마다 student 객체의 "person 관점"과 "gp_list_node 관점"을 생성할 수 있어야 한다. 기본 클래스 중 하나(예를 들어 person)에 대해 단일 상속에서 했던 것과 동일한 것을 할 수 있다. 기본 클래스의 자료 멤버를 유도된 클래스 표현의 시작부에 놓게 하고 그 기본 클래스의 상속 메소드를 vtable의 시작부에 놓게 하는 것이 그 예다. 그러면 student 객체로의 참조를 person 변수로 대입할 때 person 변수를 조작하는 코드는 자료 멤버와 vtable의 접두어를 사용할 뿐이다.

**예 9.51**
(반복이 없는) 다중 상속

나머지 기본 클래스(gp_list_node)에 대해 상황은 더 복잡해진다. 유도된 클래스의 시작부에 두 기본 클래스를 모두 놓을 수는 없다. 한 가지 가능한 해결책을 ©(심화학습에 있는) 그림 9.7에 제시한다. 이는 엘리스와 스트로스트룹[ES90, Chap. 10]이 설명한 구현에 느슨히 기반한다. student의 gp_list_node 필드가 person 필드의 뒤에 오기 때문에 student 객체로의 참조를 유형 gp_list_node*의 변수에 대입하는 것은 컴파일 시 상수 오프셋 d를 추가함으로써 우리의 "관점"을 조정하게 요구한다.

student에 대한 vtable을 두 부분으로 나눌 수 있다. 첫 번째 부분은 유도된 클래스와 첫 기본 클래스(person)의 가상 메소드를 나열한다. 두 번째 부분은 두 번째 기본 클래스의 가상 메소드를 나열한다(이미 클래스 person에서 정의한 print_mailing_label 메소드를 도입했었다. gp_list_node가 노드의 내용물을 출력할 수 있는 표현으로 표준 출력에 덤프하는 가상 메소드 debug_print를 정의하는 것을 유사하게 생각할지도 모른다). 3개 이상의 기본

클래스로 일반화하는 것은 직관적이다. 연습문제 9.19를 보라.

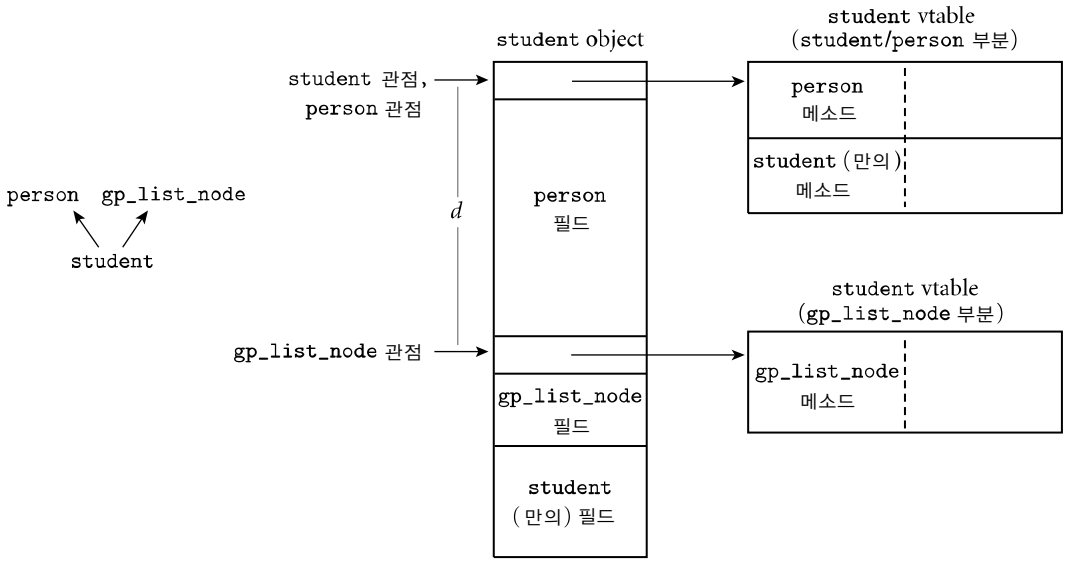



**그림 9.7 | (반복이 없는) 다중 상속의 구현.** 객체의 person 부분의 크기 d는 컴파일 시 상수다. 간접을 실행하기 전에 student 객체의 주소에 d를 더함으로써 vtable의 gp_list_node 부분에 접근한다. 마찬가지로 객체의 주소에 d를 더함으로써 student 객체의 gp_list_node 관점을 만든다. 각 vtable의 항목은 메소드 주소와 그 vtable에 접근할 때 이용하는 관점과 메소드가 정의된 클래스의 관점 간의 부호를 지닌 거리와 동일한 "this 보정" 값으로 이뤄진다.

student 객체의 모든 자료 멤버는 컴파일 시에 상수인 객체의 시작 부분으로부터의 오프셋을 갖는다. 마찬가지로 모든 가상 메소드도 컴파일 시 상수인 vtable의 해당 부분 중 하나의 시작 부분으로부터의 오프셋을 갖는다. vtable의 person/student 부분의 주소를 객체의 시작 부분에 저장한다. vtable의 gp_list_node 부분을 오프셋 d에 저장한다. 클래스 gp_list_node의 객체는 vtable의 gp_list_node를 공유하지 않는다. student가 gp_list_node의 가상 메소드 중 어느 하나라도 재정의했었다면 그 테이블의 내용은 다를 것이기 때문이다.

**예 9.52**
**다중 상속을 이용한 메소드 호출**

원래 person에서 정의한 가상 메소드 print_mailing_label를 호출하기 위해 단일 상속에 대한 9.4.3절에서 보여준 것과 유사한 코드를 사용할 수 있다. 원래 gp_list_node에 정의된 가상 메소드를 호출하기 위해 우선 객체의 주소에 오프셋 d를 더해야 한다. 이는 vtable의 gp_list_node 부분의 주소를 찾기 위함이다. 그러고 나면 호출할 적절한 메소드의 주소를 찾기 위해 이 gp_list_node의 vtable을 인덱싱할 수 있다. 하지만 한 가지 문제가 남아 있다. 그 메소드에 전달할 this의 적절한 값은 무엇인가?

구체적인 예로 (재정의해야 함에도 불구하고) student가 debug_print를 재정의하지 않는다고 가정하자. 이 객체가 클래스 student에 속하면 그 객체의 주소에 d를 더한

값인 gp_list_node 관점을 debug_print에 전달해야 한다. 하지만 한 객체가 debug_print를 재정의하는 일부 클래스(아마도 transfer_student)에 속하면 transfer_student 관점을 debug_print에 전달해야 한다. 동적으로 연결한 메소드를 바운딩한 한 변수(참조나 포인터)를 통해 객체에 접근하고 있다면 컴파일 시 이 경우 중 어떤 것을 적용해야 할지를 말할 수 없다. 더 나쁘게 transfer_student 관점을 생성하는 방법을 알지 못할 수도 있다. 앞에 제시한 코드 중 이 부분을 컴파일할 때 클래스 transfer_student를 만들지 않을 수도 있다. 그래서 그것 안에 gp_list_node 항목이 얼마만큼 나타날지 확실히 알지 못한다!

**예 9.53**

**this 보정**

흔한 해결책은 vtable 항목이 한 쌍의 필드로 구성되는 것이다. 하나는 메소드의 코드의 주소, 나머지는 그 vtable을 찾을 때 사용하는 관점에 더하기 위한 "this 보정" 값이다. ©(심화학습에 있는) 그림 9.7로 돌아가서 student가 debug_print를 재정의한다면 debug_print에 대한 vtable 항목의 "this 보정" 항목은 −d를, 그렇지 않으면 0을 포함한다. (이전에 작성한) 클래스 transfer_student에 대한 vtable의 gp_list_node 부분에서 "this 보정" 필드는 일부 다른 값 −e를 갖을지도 모른다. 일반적으로 "this 보정"은 메소드를 선언한 (그리고 그 vtable에 접근할 때 사용한) 클래스의 관점과 메소드를 정의한 (그래서 서브루틴의 구현이 기대하고 있는) 클래스의 관점 간의 거리다.

변수 my_student가 실행 시간에 일부 객체(의 student 관점)로의 참조를 포함하고 debug_print가 gp_list_node의 3번째 가상 메소드면 my_student.debug_print를 호출하기 위한 코드는 다음과 같을 것이다.

```
r1 := my_student                -- 객체의 student 관점
r1 := r1 + d                    -- 객체의 gp_list_node 관점
r2 := *r1                       -- 적절한 vtable의 주소
r3 := *(r2 + (3−1) × 8)         -- 메소드 주소
r2 := *(r2 + (3−1) × 8 + 4)     -- this 보정
r1 := r1 + r2                   -- this
call *r3
```

여기서 메소드 주소와 this 보정 모두 4바이트 길이라고 가정하고 있다. 일반적인 기계상에서 이 코드는 단일 상속을 이용할 때 필요한 코드보다 (1번의 메모리 접근을 포함해) 3개의 명령어가 많고 정적으로 식별된 메소드에 대한 호출보다 (3번의 메모리 접근을 포함한) 5개의 명령어가 많다.

## 【9.5.1】 의미적 모호성

예 9.54

1개 이상의 기본 클래스에서 발견할 수 있는 메소드

(지금까지 설명해왔던 것 중 일부일 뿐인) 구현의 복잡성과 더불어 다중 상속은 잠재적인 의미상의 문제를 가져온다. gp_list_node와 person 모두가 debug_print 메소드를 정의한다고 가정하자. 유형 student*의 변수 s를 가지고 있고 s->debug_print를 호출한다면 메소드의 어떤 버전을 불러낼 것인가? CLOS와 파이썬에서 유도된 클래스의 헤더에서 처음 나타난 기본 클래스의 버전을 불러낸다. 에펠에서는 이런 모호성을 가진 유도된 클래스를 정의하려 한다면 정적 의미 오류다. C++에서 유도된 클래스를 정의할 수 있지만 모호성이 있는 이름을 가진 멤버를 사용하려 한다면 정적 의미 오류가 된다. 에펠에서 유도된 클래스를 정의할 때 이름 정하기의 충돌을 제거하기 위해 특징 개명 방법을 사용할 수 있다. C++에서 모호성 있는 멤버를 명시적으로 재정의할 것이다.

```
void student::debug_print() {
    person::debug_print();
    gp_list_node::debug_print();
}
```

여기서 기본 클래스 둘 모두의 debug_print 루틴을 호출하는 것은 둘을 명명하기 위한 :: 유효 범위 결정 연산자를 사용해 택했다. 물론 단지 하나만을 선택하거나 처음부터 자신의 코드를 작성하게 택할 수 있다. 새로운 이름을 제공해서 두 루틴에 대한 접근을 준비하게 할 수 있다.

```
void student::debug_print_person() {
    person::debug_print();
}
void student::debug_print_list_node() {
    gp_list_node::debug_print();
}
```

예 9.55

모호한 메소드의 재정의

동등하게 식별된 기본 클래스 메소드의 둘 중 하나나 둘 모두가 가상이고 유도된 클래스에서 이를 재정의하길 원하면 상황은 좀 더 복잡해진다. 스토르스트룹[Str97, Sec. 25.6]에 따르면 "인터페이스" 클래스를 각 기본 클래스와 유도된 클래스 사이에 삽입함으로써 이 문제를 해결할 수 있다.

```
class person_interface : public person {
    virtual void debug_print_person() = 0;
    void debug_print() { debug_print_person(); }
       // person::debug_print를 재정의함
};
```

```
class list_node_interface : public gp_list_node {
    virtual void debug_print_list_node() = 0;
    void debug_print() { debug_print_list_node(); }
       // gp_list_node::debug_print를 재정의함
};
class student : public person_interface, public list_node_interface {
public:
    void debug_print_person() { ...
    void debug_print_list_node() { ...
    ...
};
```

**설계와 구현**

### 다중 상속의 비용

지금까지 다중 상속에 대해 설명해왔던 vtable에서 this 보정을 사용한 구현은 모든 가상 메소드 호출의 오버헤드를 증가시키는 불리한 성질을 가지고 있다. 심지어 다중 상속을 사용하지 않는 프로그램에서도 그렇다. 이런 종류의 의무적인 오버헤드는 언어 설계자(그리고 특히 시스템 언어의 설계자)가 일반적으로 피하려 하는 것이다. 신조에 따라 복잡한 특별 경우는 더 간단하고 일반적인 경우의 효율성을 감소시키지 않아야 한다. 운 좋게도 보정이 0이 아닐 때만 this를 변경하는 비용을 지불하는 다중 상속의 다른 구현이 있다(©심화학습에 있는 연습문제 9.25를 보자).

`student` 객체를 유형 `person*`의 변수 `p`로 대입해 `p->debug_print()`를 호출할 때 발생하는 것을 보기 위한 것은 연습문제 9.20에 남겨둔다.

**예 9.56**

반복이 있는 다중 상속

클래스 `D`가 두 기본 클래스, 공통 기본 클래스 `A`를 상속하는 `B`와 `C`를 상속할 때 좀 더 심각한 모호성은 발생한다. 이 상황에서 클래스 `D`의 객체가 클래스 `A`의 자료 멤버의 한 인스턴스를 포함해야 하는가? 답은 프로그램에 따라 다른 것 같이 보일 것이다. 예를 들면 앞에서 제시한 관리 컴퓨팅 시스템에서 동일한 부서의 교수 모두를 한 연결 리스트에 저장하기를 원한다고 가정하자. 클래스 `student`와 마찬가지로 클래스 `professor`가 `person`과 `gp_list_node`를 모두 상속하기를 바랄 수도 있다.

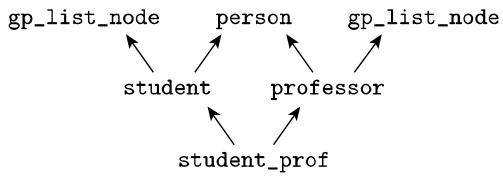
```
class professor : public person, public gp_list_node { ...
```

게다가 교수가 때때로 입학하지 않은 학생으로서 강좌를 듣는다고 가정하자. 이 경우 두 집합의 연산을 모두 지원하는 새로운 클래스를 원한다.

```
class student_prof : public student, public professor { ...
```

클래스 `student_prof`는 `student`와 `professor`로부터 `person`과 `gp_list_`

node를 두 번 상속 받는다. 이에 대해 생각해보면 student_prof는 클래스 person의 자료 멤버에 속하는 한 개의 인스턴스(하나의 이름, 하나의 대학 ID, 하나의 메일 주소)와 클래스 gp_list_node의 자료 멤버에 속하는 두 개의 인스턴스(두 전임자와 두 후임자), 즉 입학하지 않은 학생의 리스트 내로의 연결을 위한 한 집합과 어떤 부서에 대한 교수진 리스트 내로의 연결을 위한 또 다른 집합을 갖기를 원할 것이다.




상속 트리의 각 분기에서 별개의 사본을 가지는 gp_list_node의 경우를 중복 상속이라 한다. 트리의 두 분기 모두에서 하나의 사본을 가지는 person의 경우를 공유 상속이라 한다. 두 경우 모두 반복 상속의 형태다.

**예 9.57**

**C++에서의 공유 상속**

중복 상속은 C++에서 기본이다. 공유 상속은 에펠에서 기본이다. C++에서 기본 클래스를 virtual로 선언하게 지정해 공유 상속을 취할 수 있다.

```
class student : public virtual person, public gp_list_node { ...
class professor : public virtual person, public gp_list_node { ...
```

이 경우 여러 경로에서 상속하면 클래스 person의 멤버를 공유하는 반면 클래스 gp_list_node의 멤버는 중복된다.

**예 9.58**

**에펠에서 중복 상속**

9.2.2절에서 설명한 개명 방법을 통해 개개의 특징의 중복 상속을 에펠에서 얻을 수 있다.

```
class student inherit person; gp_list_node ...
class professor inherit person; gp_list_node ...

class student_prof
inherit
    student
       rename
          prev as prev_student,
          next as next_student
       end;
    professor
       rename
          prev as prev_prof,
          next as next_prof
       end
```

```
feature
    ...
end                -- 클래스 student_prof
```

다른 마지막 이름을 가지고 상속된 특징을 중복한다. 동일한 마지막 이름을 가지고 상속된 특징을 공유한다. 위에서 명시적으로 보여준 것처럼 사용자가 인터페이스 클래스를 중간 중간에 끼워 넣지 않는다면 CLOS에서 다중 상속을 항상 공유한다. CLOS에는 다른 개명 방법이 없다.

## 【9.5.2】 중복 상속

**예 9.59**
**중복 상속의 사용**

중복 상속은 반복이 없는 다중 상속의 구현에 비해 심각한 구현 문제를 이끌어내지 않는다. ©(심화학습에 있는) 그림 9.8에서 제시한 것처럼 (클래스 D의 경우) 이 상속 트리에서 두 다른 경로로 기본 클래스(A)를 상속한 그 표현에서 A의 자료 멤버의 두 사본과 그 vtable의 모든 부분 내에 있는 A의 가상 메소드에 대한 항목들을 갖는다. D 객체의 B 관점에서의 생성(예를 들어 한 D 객체로의 포인터를 B* 변수로 대입하는 경우)은 코드의 실행을 필요로 하지 않을 것이다. C 관점의 생성(예를 들어 C* 변수로 대입하는 경우)은 오프셋 d의 덧셈을 필요로 할 것이다.

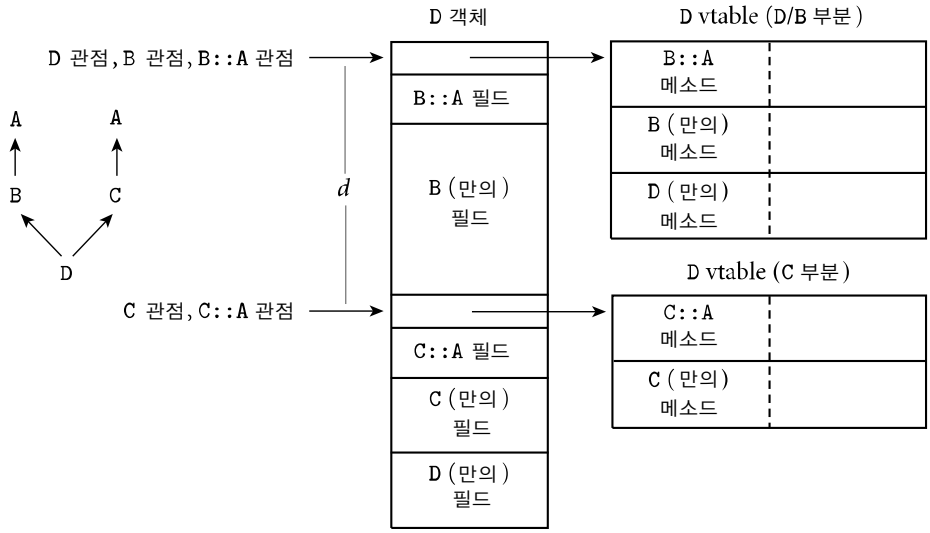



**그림 9.8 | 중복 다중 상속의 구현.** 각 기본 클래스는 클래스 A의 완전한 사본을 포함한다. ©(심화학습에 있는) 그림 9.7과 같이 클래스 D의 vtable을 각 기본 클래스에 대한 vtable로 분리한다. 그리고 각 vtable 항목은 (메소드 주소, this 보정) 순서쌍으로 구성된다.

모호성 때문에 이름으로 D 객체의 A 멤버에 접근할 수 없다. 하지만 B*나 C* 변수로 D 객체로의 포인터를 대입한다면 그것에 접근할 수 있다. 유사한 예로 D 객체로의 포인터를 A 포인터로 직접 대입할 수 없다. 한 관점을 생성하기 위해 A를 선택하는 것에 대해 어떤 근거도 없기 때문이다. 하지만 B*이나 C* 중간 매개체를 통해 대입을 수행할 수 있다.

```
class A { ...
class B : public A { ...
class C : public A { ...
class D : public B, public C { ...
...
A* a;     B* b;       C* c;       D* d;
a = d;       // 오류; 모호함
b = d;       // 괜찮음
c = d;       // 괜찮음
a = b;       // 괜찮음; a := d's B's A
a = c;       // 괜찮음; a := d's C's A
```

©(심화학습에 있는) 예 9.53에서 설명한 것처럼 vtable 항목은 (메소드 주소, this 보정) 순서쌍으로 구성될 필요가 있을 것이다.

## 【9.5.3】 공유 상속

**예 9.60**
**공유 상속을 이용한 메소드의 재정의**

공유 상속은 모호성과 추가적인 구현 복잡성에 대한 새로운 가능성을 도입한다. 이전 하위 절처럼 D가 B와 C에서 상속되고 둘 모두 A에서 상속된다고 가정하자. 하지만 이 경우 A를 공유한다고 가정한다.

```
class A {
public:
    virtual void f();
    ...
};
class B : public virtual A { ...
class C : public virtual A { ...
class D : public B, public C { ...
```

B나 C가 A에 선언된 메소드 f를 재정의한다면 새로운 모호성이 발생한다. 어떤 버전을 D가 상속할 것인가? 가능한 정의 중 하나가 나머지보다 우위를 차지한다면 C++은 그것의 클래스가 나머지 모든 정의에 속하는 클래스의 후계자인 점에서 f로의 참조를 정의하는 것이 명백하다(그래서 유효하다). 위의 특정한 예에서 F는 B나 C 중 하나에서의

f의 재정의된 버전을 상속할 수 있다. 하지만 둘 모두가 그것을 재정의한다면 D의 코드 내에서 f를 사용하려는 시도는 정적 의미 오류일 것이다. 에펠은 모호성을 제어하기 위해 상대적으로 정교한 방법을 제공한다. 둘 이상의 경로에서 재정의된 메소드를 상속하는 클래스는 원하는 것을 지정할 수 있다. 다른 방법으로 개명을 통해 모든 버전으로의 접근을 존속시킬 수 있다.

**예 9.61**

**공유 상속의 구현**

공유 상속을 구현하기 위해 A의 단일 인스턴스가 B와 C 모두의 특정 부분이기 때문에 B와 C 모두의 표현을 메모리 내에 인접하게 만들 수 없음을 인식해야 한다. 사실 ©(심화학습에 있는) 그림 9.9에서 B와 C 모두 인접하게 만들게 선택하지 않았다. 하지만 모든 B, C, D 객체(그리고 유도된 클래스의 객체의 모든 B, C, D 관점)의 표현이 그 객체의 A 부분의 주소를 컴파일 시 상수로 된 그 관점의 시작부로부터의 오프셋에 가지고 있음을 주장한다. A의 자료 멤버에 접근하기 위해 우선 이 주소를 통해 간접적으로 접근하고 나서 A 내에 있는 그 멤버의 오프셋을 적용한다. A에 선언된 n번째 가상 메소드를 호출하기 위해 다음 코드를 실행한다.

```
r1 := my_D_view                 -- 객체의 원래 관점
r1 := *(r1 + 4)                 -- A 관점
r2 := *r1                       -- vtable의 A 부분 주소
r3 := *(r2 + (n-1) × 8)         -- 메소드 주소
r2 := *(r2 + (n-1) × 8 + 4)     -- this 보정
r1 := r1 + r2                   -- this
call *r3
```

이 코드는 가상이 아닌 기본 클래스(©심화학습에 있는 예 9.53)을 위한 코드와 길이에 있어서 동일한 수의 명령어다. 하지만 (A 주소를 통한 간접적인 접근을 위한) 1개 더 많은 메모리 접근을 포함한다. 그 코드는 D에서 유도한 클래스의 객체를 포함해 D와 A 관점이 좀 더 넓게 분리된 객체의 D 관점을 이용해 동작할 것이다. 2행에서 상수 4는 D의 vtable 시작 주소 바로 뒤에 D의 A 부분 주소가 위치한 것과 더불어 4바이트 단위의 주소를 가정한다. 둘 이상의 가상 기본 클래스를 가진 객체에서 그 각 기반에 대응하는 객체 부분의 주소를 객체의 시작부에서의 다른 오프셋에서 찾을 수 있을 것이다.

C++에서 ©(심화학습에 있는) 그림 9.9의 구현 전략은 기본 클래스가 virtual(공유)라고 알고 있기 때문에 동작한다. 가상이 아닌 기본 클래스에 속한 자료 멤버와 가상 메소드에 대해 ©(심화학습에 있는) 그림 9.7과 ©(심화학습에 있는) 그림 9.8의 더 가벼운 검색 알고리즘을 계속 사용한다. 반면 에펠에서는 클래스 계층의 한 수준에 중복을 통해 상속된 특징을 이후에 공유를 통해 상속할 수도 있다. 결과적으로 에펠은 좀 더 정교한 구현 전략을 요구한다(연습문제 9.26을 보자).

C++에서 한 클래스에 있는 vtable의 일정 부분을 중복시키려 한다면 가상 기본 클래스의 가상 메소드에 접근할 때 추가적인 수준의 간접적 접근을 피할 수 있다. 연습문제 9.27에서 이 선택 사항에 대해 알아본다.

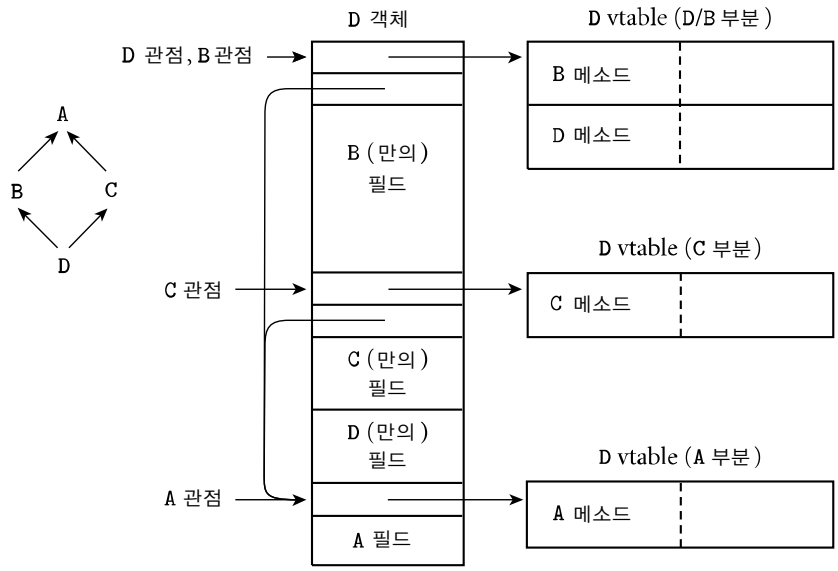



**그림 9.9 | 공유 다중 상속의 구현.** 클래스 B, C와 D의 객체는 컴파일 시 상수로 된 오프셋에 A 구성요소의 주소(이 경우 vtable 주소 바로 다음)를 포함한다. ©(심화학습에 있는) 그림 9.7과 ©(심화학습에 있는) 9.8과 마찬가지로 vtable 항목에서 가상 메소드에 대한 this 보정은 그 메소드를 선언한 클래스의 관점에 대응한다(즉 이를 통해 vtable에 접근할 함).

## 【9.5.4】 혼합 상속

다중 상속의 주제를 떠나기 전에 9.4.2절에서 넘어간 추상 메소드로 전부 만들어진 기본 클래스의 개념으로 잠시 돌아간다. 그런 클래스를 자바에서는 **인터페이스**라 부른다. 그것은 자료 멤버나 그 메소드의 구현 중 어느 것도 가지지 않는다.[1] 그러므로 그것은 다중 상속의 의미 모호성과 구현 복잡성의 대부분에 영향을 받지 않는다.

"실제" 기본 클래스와 임의의 수의 인터페이스에서의 상속을 **혼합 상속**이라 한다. 인터페이스의 가상 메소드를 유도된 클래스의 메소드 "속에 혼합한다." 인터페이스의 상속에 대해 말하는 것은 상황을 자기에게 편리하게 해석하는 것일 수도 있다. 유도된 클래스는 인터페이스의 각 메소드에 대한 정의를 제공해야 하기 때문이다. 하지만 인터페이스는 다형성을 통한 코드 재사용을 편리하게 한다. 서브루틴의 형식 매개변수를 인터페이스 유형을 가지게 선언하면 그 인터페이스를 구현한(상속한) 어느 클래스라도 대응하는 실제 매개변수로 전달할 수 있다. 합법적으로 전달할 수 있는 객체의 클래스는 공통의 클래스 조상을 가질 필요가 없다.

---

주1. 자바는 실제로 인터페이스가 자료 멤버를 가지게 한다. 하지만 그런 멤버는 항상 상수다. 항상 그 값을 인터페이스 선언에 지정해야 한다.

**예 9.62**

**인터페이스를 유도된 클래스로 혼합하기**

한 예로 텍스트 영역에 따라 객체를 정렬하고 웹 브라우저 창 안에 객체의 그래픽 표현을 표시하고(적절하게 감추고 새롭게 함) 사전식 자료 구조에서 이름에 의한 객체로의 참조를 저장하는 일반적인 목적의 자바 코드를 준다고 가정하자. 인터페이스는 이 각 능력을 표현할 것이다. 객체 widget의 복잡한 클래스를 개발했었다면 ©(심화학습에 있는) 그림 9.10에 보이는 것처럼 적절한 인터페이스를 widget에서 유도된 클래스로 혼합해 일반적인 목적 코드를 사용할 수 있다.

**예 9.63**

**혼합 상속의 컴파일 시 구현**

9.4.3절에서 언급했듯이 자바 구현은 대개 메소드를 실행 시간에 이름으로 검색한다. 이 경우에 인터페이스를 구현하는 모든 클래스 속의 메소드 사전에 인터페이스의 메소드를 간단히 추가할 수 있다. 실행 시간 메소드 검색 없는 혼합 상속을 구현하기 위한 한 가지 간단한 접근 방법은 구현된 인터페이스에 대한 vtable의 주소를 이용해 그 클래스에 속하는 객체의 표현을 증대시키는 것이다. 인터페이스와 자료 멤버를 클래스 계층의 일부 수준에 추가하면 vtable 포인터와 자료 멤버를 객체 내 임의의 오프셋에 산재시킬 수도 있다.

```
public class widget { ...
}
interface sortable_object {
    String get_sort_name();
    bool less_than(sortable_object o);
    // 인터페이스의 모든 메소드는 자동적으로 공용이다.
}
interface graphable_object {
    void display_at(Graphics g, int x, int y);
    // Graphics는 그래픽 객체를 렌더링하는
    // 문맥을 제공하는 표준 라이브러리 클래스다.
}
interface storable_object {
    String get_stored_name();
}
class named_widget extends widget implements sortable_object {
    public String name;
    public String get_sort_name() {return name;}
    public bool less_than(sortable_object o) {
       return (name.compareTo(o.get_sort_name()) < 0);
```

**그림 9.10 | 자바의 인터페이스 클래스.** name_widget 내의 sortable_object 인터페이스와 augmented_widget 내의 graphable_object와 storable_object 인터페이스를 구현해 sorted_list.insert, browser_window.add_to_window, dictionary.insert 등과 같은 루틴으로나 그 루틴에 그 클래스의 객체를 전달할 수 있는 능력을 얻게 된다.(이어짐)

```
        // compareTo는 표준 라이브러리 클래스 String의 메소드다.
    }
}
class augmented_widget extends named_widget
          implements graphable_object, storable_object {
    ...         // 좀 더 많은 자료 멤버
    public void display_at(Graphics g, int x, int y) {
    ...         // g의 메소드에 대한 호출들
    }
    public String get_stored_name() {return name;}
}
...
class sorted_list {
    public void insert(sortable_object o) { ...
    public sortable_object first() { ...
    ...
}
class browser_window extends Frame {
    // Frame은 창을 위한 표준 라이브러리 클래스다.
    public void add_to_window(graphable_object o) { ...
    ...
}
class dictionary {
    public void insert(storable_object o) { ...
    public storable_object lookup(String name) { ...
    ...
}
```

그림 9.10 | 자바의 인터페이스 클래스

## ✓ 확인문제

42. 한 클래스가 2개 이상의 기본 클래스를 가질 때 발생하는 의미적 모호성에 대한 몇 가지 예를 제시하라.

43. 중복 상속과 공유 다중 상속 간의 차이점을 설명하라. 각각은 언제 바람직한가?

44. 객체(의 구현)의 반복이 없는 다중 상속은 다중 관점과 vtable의 "this 보정" 항목을 요구하는가?

45. 특정 부모 클래스의 항목에 접근할 때 공유 다중 상속이 추가적인 수준의 간접 접근을 어떻게 필요로 하는지를 설명하라.

46. 자바나 C#이 정의하는 인터페이스는 무엇인가? 혼합 양식 상속과 어떻게 관계가 있는가?

47. 혼합 상속이 다중 상속의 다른 양식보다 구현하기 더 간단한 이유는?

---

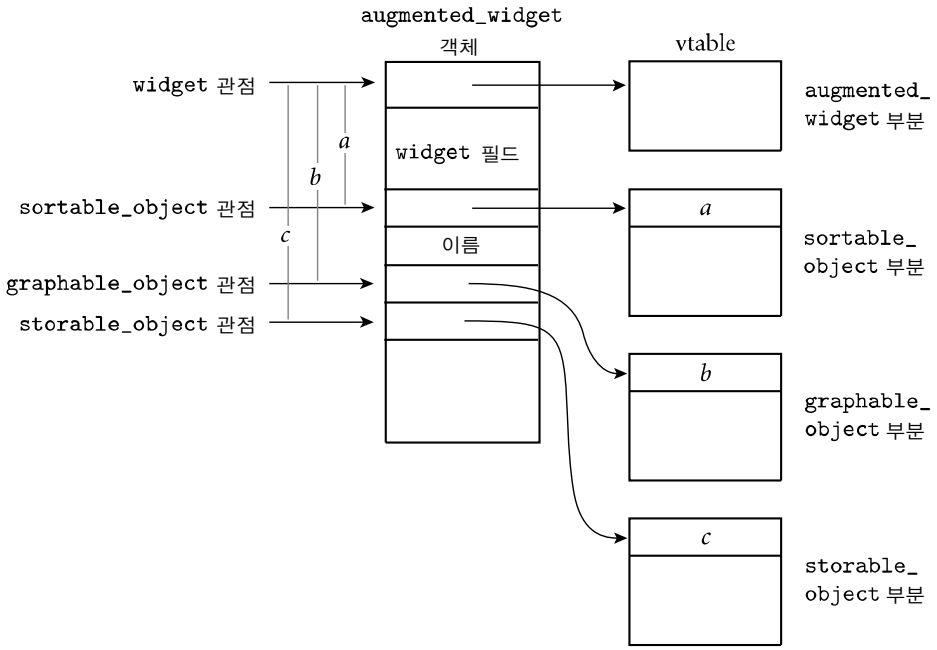



**그림 9.11 | 혼합 상속의 구현.** 클래스 augmented_widget의 객체는 4개의 vtable 주소를 가지고 있다. 하나는 (그림 9.4과 마찬가지로) 클래스 자신을 위한 것이고 셋은 구현된 인터페이스를 위한 것이다. 인터페이스 루틴에 전달된 객체의 관점은 적절한 vtable 포인터를 직접 가리킨다. 그러고 나서 vtable은 객체 자체로의 포인터를 재생성하기 위해 메소드 모두가 사용하는 하나의 this 보정으로 시작한다.