프로그래밍 언어론
C++ 구조 분석

목차

목차 1
소개 1
연재 가이드 1
필자소개 1
필자 메모 1
Introduction 2
클래스와 인스턴스 2
은닉성 4
상속성 5
다형성 8
마법은 없다 11
참고자료 11

소개

C++은 C에서 제공하지 못하는 방대한 양의 언어적인 메커니즘을 제공한다. 그러한 C++의 중요한 언어적인 메커니즘과 C++ 컴파일러가 어떻게 그것을 처리하고 있는지, 왜 그렇게 처리하고 있는지에 대해서 살펴본다.

연재 가이드

운영체제: Windows XP
개발도구: Visual Studio 2005
기초지식: C/C++/어셈블리어 문법
응용분야: COM 프로그래밍

필자소개

신영진 pop@jiniya.net, http://www.jiniya.net
최고를 꿈꿨다. 하지만 이제는 더 이상 최고를 꿈꾸지 않는다. 연날리기를 하면서 올해는 최고가 아닌 최선을 소망했다. 사람들은 항상 최고를 꿈꾼다. 랑데부홈런을 날리는 그런 멋진 순간. 해발고도 9000미터에 달하는 히말라야 정상에 서는 아찔한 순간이 자신의 삶에 찾아오기를 고대한다. 요즘은 그런 결과보다는 과정에 좀 더 의미를 두고 싶다.

필자 메모

Introduction

청소기를 켜면서 한번이라도 청소기 모터가 어떻게 먼지를 흡입할 수 있는지에 대해서 생각해 본 적이 있는지, 핸드폰을 사용하면서 그것이 어떻게 주파수를 사용하는지, 기지국을 넘나들 때 어떤 원리로 교환되는지에 대해서 고민해본 적이 있는지, MP3를 들으면서 어떻게 수십 메가에 달하는 웨이브 파일이 그렇게 작은 파일로 압축이 되는지에 대해서 의문을 가져본 적이 있는지 생각해보자.

아마 일반적인 사람이라면 이러한 것들에 대해서 고민해 본 적이 없을 것이다. 이 글을 읽고 있는 당신이 괴짜라면 한 번쯤 생각해 봤을지도 모르겠다. 어쨌든 이러한 질문들은 복잡하기만 할 뿐 기계를 사용하는데 전혀 도움을 주지 않는다. 단지 사람들은 청소기의 전원을 켜고 바닥을 문지르면 먼지가 먼지통으로 들어간다는 것, 핸드폰의 SEND 버튼을 누르면 친구한테 전화를 걸 수 있다는 사실, 과거의 CD는 고작 20곡 남짓 저장할 수 있었지만 MP3에는 수백 곡을 저장할 수 있다는 것만 기억할 뿐이다. 그 밖의 다른 사실은 다 부가적인 것이다.

일반적인 사용, 이용의 수준에서는 그러한 지식이면 충분하다. 하지만 그러한 단계를 넘어서 예술의 경지로 끌어올려야 하는 순간에는 내부 구조를 이해하는 것이 필수적이다. 최고의 피아니스트가 되기 위해서는 피아노가 어떻게 소리를 내는지 이해할 필요가 있다는 말이다. 마찬가지로 자동차의 내부구조에 대해서 문외한인 사람이 베스트 드라이버가 되기란 힘들다. 개발자도 마찬가지다. 자신이 사용하는 언어, 시스템의 내부 구조에 대한 이해 없이 그것을 제대로 알 고 있다는 생각은 자만이고 착각이다.

흔히 하는 우스갯소리 중에 C언어는 2시간 배워서 20년을 써먹고, C++은 20년을 배워서 2시간을 써먹는다는 이야기가 있다. 그만큼 C++은 크고 방대한 언어라는 이야기다. 따라서 이 지면에 담을 수 있는 내용은 그 중 극히 일부분일 수 밖에 없다. 여기서는 C++의 근간을 이루고 있는 몇 가지 언어적인 핵심 메커니즘과 그것을 C++ 컴파일러가 어떻게 처리하고 있는지, 왜 그렇게 처리했는지에 대해서 간략히 다루도록 한다. 지면에 설명된 내용은 모두 Visual C++ 2005 컴파일러를 기준으로 하고 있다. 표준안은 직접적인 구현에 대해서는 언급하지 않기 때문에 컴파일러에 따라 세부적으로 다른 방식으로 구현하고 있을 수 있다.

클래스와 인스턴스

C와 C++의 가장 큰 차이점을 나타내는 키워드 하나를 꼽으라면 누구나 class를 선택할 것이다. 그만큼 C++에 있어서 클래스는 중요한 키워드이자 개념이다. C++ 아버지라 할 수 있는 Bjarne Stroustrup 아저씨가 설계한 초기 작품의 이름이 C with Classes란 것을 보더라도 그러한 사실을 잘 알 수 있다. 따라서 클래스를 이해하는 것이 C++의 구조를 이해하는데 가장 큰 첫 번째 걸음이 될 것이다.

이 글은 C++의 문법은 알고 있는 독자들을 대상으로 하고 있기 때문에 클래스의 사용 방법에 대한 내용을 구구절절 설명하지 않을 계획이다. 대신 C++을 처음 배우는 사람들이 가장 헷갈려 하

는 클래스와 그 인스턴스의 관계에 대해서만 간략히 다루도록 하겠다.

<리스트 1>에 간단한 Car 클래스가 나와 있다. 처음 C++이란 언어를 접하는 분들이 오해하는 가장 큰 이슈는 <리스트 2>에 나와 있다. 바로 Car와 car의 관계다. 오해하는 부분을 정리해보면 다음 두 가지로 요약된다.

1. 메모리에는 Car만 존재하고, 그것으로 인스턴스화 되는 것은 단지 참조할 뿐이다.
2. 메모리에는 Car의 모든 것이(멤버 변수, 멤버 함수) 인스턴스 별로 저장된다.

안타깝게도 두 가지 생각 모두 잘못된 상식이다. 실제로는 어떻게 처리되는지를 살펴보도록 하자. <리스트 2>에 나타난 어셈블리 코드를 보면 알 수 있듯이 멤버 함수는 멤버 데이터와 완전히 별도로 처리된다. 즉 클래스의 인스턴스는 멤버 데이터만 저장할 수 있는 공간만 가지고 있다고 생각하면 된다. 멤버 함수는 같은 클래스에서는 똑같이 사용된다. 그렇다면 어떻게 다른 클래스인지를 자동적으로 알고 그곳에 저장할까? 그것은 멤버 함수 호출 규약에 있다. C++ 컴파일러는 내부적으로 멤버 함수를 호출할 때 그것의 인스턴스 정보를 함수에 같이 전달한다. <리스트 2>에서는 ecx에 그 정보를 담고 있다. 각 멤버 함수는 넘어온 해당 인스턴스 정보를 this를 통해 참조해서 인스턴스에 맞는 정보를 처리할 수 있는 것이다. 실제로 이렇게 호출이 일어난 후의 인스턴스 메모리 공간을 살펴보면 <화면 1>과 같이 8바이트의 데이터 공간만 존재한다는 것을 볼 수 있다.

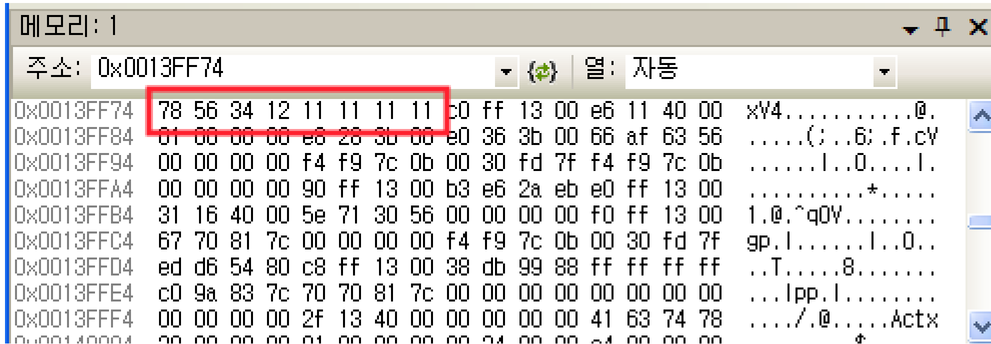
리스트 1 Car 클래스

```
class Car
{
private:
    int m_speed;
    int m_fuel;

public:
    Car() { Speed(0); Fuel(0); }

    int Speed() { return m_speed; }
    void Speed(int speed) { m_speed = speed; }

    int Fuel() { return m_fuel; }
    void Fuel(int fuel) { m_fuel = fuel; }
};
```

리스트 2 클래스 생성 및 멤버 함수 호출 과정

```
Car car;
00401869 8D 4D F8         lea         ecx,[car]
0040186C E8 8F F7 FF FF   call        Car::Car (401000h)

car.Speed(0x12345678);
car.Fuel(0x11111111);
0040187E 68 11 11 11 11   push        11111111h
00401883 8D 4D F8         lea         ecx,[car]
```

```
00401886 E8 C5 F7 FF FF    call         Car::Fuel (401050h)
```



화면 1 Car 클래스의 메모리 레이아웃

좀 더 정확하게 표현하자면 클래스의 인스턴스에는 멤버 데이터를 위한 공간과 C++ 컴파일러가 클래스를 다루는데 필요한 메타데이터가 저장된다. Car 클래스의 경우 이러한 메타데이터가 전혀 없기 때문에 멤버 데이터만 저장된 것이다. 따라서 메타 데이터가 손상될 수 있기 때문에 클래스로 만든 데이터는 절대로 원시 데이터 타입(primitive data type)처럼 취급해서는 안 된다. 이러한 대표적인 실수가 클래스를 메모리에서 직접 복사하거나(memcpy) 파일에 직접 기록하는(fwrite) 것이다.

은닉성

클래스에 포함된 멤버 변수가 C언어의 구조체 멤버와 다른 점은 접근 제어를 할 수 있다는 점이다. 클래스 멤버 변수는 접근 제어 연산자인 public/protected/private을 통해서 외부에서 접근할 수 있는 단계를 조절할 수 있다.

흔히 C언어에서 C++로 넘어온 개발자가 가지는 잘못된 생각 중의 하나가 이 접근 제어 연산자 때문에 오버헤드가 발생한다고 생각하는 것이다. C언어 개발자들이 성능에 민감한 편이라는 점과 보통 복잡해 보이는 것은 모두 오버헤드를 가진다고 생각한다는 점에서 그런 오해를 하는 것도 이상한 일은 아니라는 생각이 들긴 한다.

은닉성을 구현하는 세 개의 키워드를 통한 접근 제어는 런타임이 아닌 컴파일 타임에 컴파일러에 의해서 검증되는 속성이다. 컴파일러는 각 변수의 접근 권한을 테이블에 기억해 두고 있다가 적절하지 않은 장소에서 접근하는 것에 대해서 경고 메시지를 출력해준다. 따라서 이러한 은닉성에 대한 런타임 오버헤드는 없다. 또한 <리스트 3>과 같이 메모리를 직접 조작하는 방법을 통해서 런타임에 접근제어를 무시하고 변수 값을 변경할 수 있다.

리스트 3 런타임에 private 멤버 데이터를 변경하는 예제

```
int main()
{
    Car car;
    car.Speed(0);
    printf("%d\n", car.Speed());
```

```
    // m_speed 값을 강제로 변경한다.
    int speed = 300;
    memcpy(&car, &speed, sizeof(speed));
    printf("%d₩n", car.Speed());

    return 0;
}
```

그렇다면 왜 C++ 컴파일러는 이러한 검사를 런타임에는 하지 않았을까? 이유는 간단하다. OOP의 이러한 개념은 사람들이 편하기 위해서 만들어진 것이지 기계적인 접근을 검증하기 위해서 만들어진 것이 아니기 때문이다. 인간이 개입하는 소스 코드 단계에서만 검증을 해주는 것으로도 충분히 그 기능을 다한다고 생각한 것이다. 또한 정상적인 C++ 개발자라면 <리스트 3>에 나타난 것과 같은 형태로 클래스의 멤버에 접근하지는 않을 것이기 때문이다. 다른 또 하나의 중요한 이유는 C++의 경우 클래스를 사용하더라도 되도록 기존 C 프로그램에 비해서 오버헤드를 가지지 않도록 설계되었다는 점이다. 따라서 컴파일타임에 검증할 수 있는 사실을 위해 불필요한 런타임 오버헤드를 추가하는 것은 설계 철학에 위배되는 일이라 할 수 있다.

상속성

C++의 클래스가 가지는 또 하나의 재미난 성질은 상속에 있다. 상속을 통해서 우리는 원본 클래스를 직접 수정하지 않고도 새로운 기능을 가진 클래스를 만들어 낸다. 그렇다면 그렇게 상속받은 클래스의 메모리 구조는 어떻게 되는 것일까? 상속된 클래스의 멤버 변수와 메소드들은 메모리에서 어떻게 배치되어 있을까? <리스트 4>에 Car를 상속받은 RacingCar 클래스가 나와있다. 각자 자신이 C++이란 언어의 설계자, 내지는 C++이란 언어의 명세대로 구현하는 개발자라고 가정한다면 이것을 어떻게 만들지 생각해보자.

리스트 4 RacingCar 클래스

```
class RacingCar : public Car
{
private:
    int m_buster;

public:
    RacingCar() { Buster(0); }
    int Buster() { return m_buster; }
    void Buster(int buster) { m_buster = buster; }
};
```

C++ 컴파일러는 간단한 방법을 사용해서 이 문제를 해결했다. 상속받은 부모의 데이터를 먼저 저장하고 이후에 자식의 데이터를 저장하는 방법이다. 즉, Car와 RacingCar의 메모리 구조는 다음 구조체와 같은 형태로 이루어진다는 의미다.

```
struct Car { int m_speed; int m_fuel; };
```

strcut RacingCar { int m_speed; int m_fuel; int m_buster; }

너무나 당연한 생각이어서 달리 이유가 없어 보일 수도 있다. 하지만 이렇게 배치를 한 데는 그렇게 할 수 밖에 없는 중요한 이유가 있다. 바로 멤버 함수의 재활용이다. 아래와 같이 반대로 배치된 상황에서의 멤버 함수 호출을 생각해보면 금방 이해할 수 있다.

struct RacingCar { int m_buster; int m_speed; int m_fuel; }

이제 RacingCar의 Speed 멤버 함수를 호출한다고 생각해보자. Car::Speed 함수는 클래스 시작 주소로부터 오프셋이 0인 지점에 인자로 넘어온 speed 값을 기록한다. 그런데 배치가 위와 같이 바뀌면 m_speed가 아닌, m_buster 값이 변경되는 부작용이 발생한다. 반대로 위에서 살펴본 것과 같이 부모 멤버를 앞쪽에 배치한다면 여전히 오프셋이 0인 지점에는 m_speed가 있기 때문에 부모 클래스의 멤버 함수를 자연스럽게 재활용할 수 있다.

단순히 부모 데이터를 앞쪽에 배치한다고 모든 문제가 해결된 것은 아니다. 다중 상속으로 넘어오면 여전히 멤버 변수 배치는 문제가 된다. <그림 1>에 다중 상속을 받은 클래스 C의 메모리 구조가 나와있다. 이 경우에 A의 멤버 함수는 그대로 사용할 수 있지만 B의 멤버 함수는 그대로 사용하지 못한다. 왜냐하면 변수들의 오프셋이 변경되었기 때문이다.

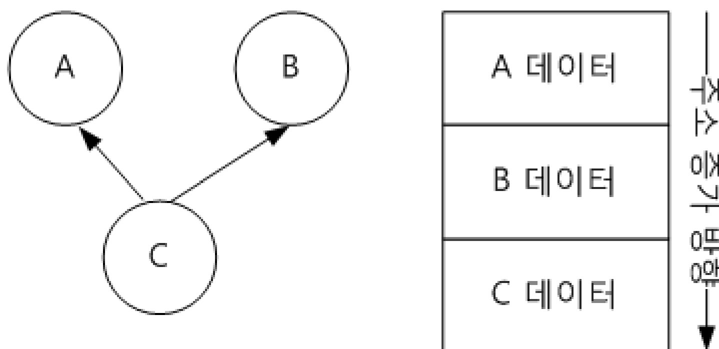


그림 1 다중 상속 클래스의 메모리 구조

C++ 컴파일러는 의외로 간단한 방법을 통해서 이 문제를 해결한다. C 클래스가 B 클래스의 멤버 함수를 호출하는 경우에는 오프셋 보정을 해주는 것이다. 즉, A로 시작하는 클래스의 시작 포인터를 멤버 함수에 전달하는 것이 아니라 그곳에 A 데이터의 크기만큼을 더한 포인터를 전달하는 것이다. 이 경우에 오프셋 계산은 모두 컴파일타임에 결정되지만 B의 멤버 함수를 호출하기 위해서 추가적인 add 연산이 들어가야 하기 때문에 런타임 오버헤드는 발생한다.

끝으로 가장 복잡한 가상 상속에 대해서 살펴보자. 가상 상속이란 <그림 2>와 같이 상속된 경우에 D에서 발생하는 문제를 해결하기 위해서 나온 개념이다. D에는 메모리 상에 A클래스의 내용이 두 번 나타나기 때문에 D에서 참조하는 A 멤버 데이터의 경우 어떤 것을 사용해야 하는지가 애매모호해지는 문제가 발생한다.

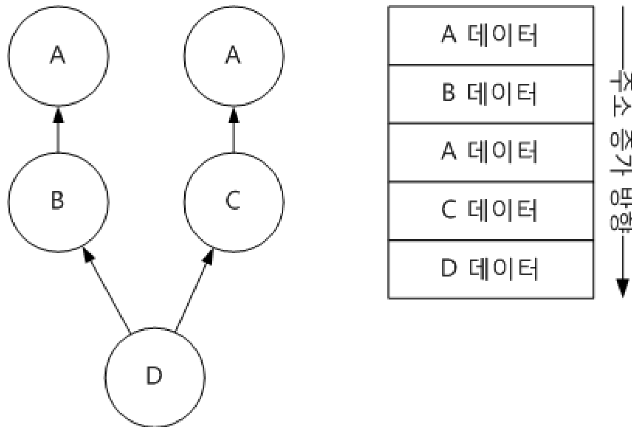


그림 2 다중 상속을 받은 클래스의 메모리 구조

이러한 문제를 해결하기 위해서는 B, C 클래스가 A를 가상 상속을 받도록 수정해 주어야 한다. 이렇게 가상 상속을 받은 경우의 메모리 구조가 <그림 3>에 나와 있다. 이 경우에 컴파일러는 D 인스턴스의 메모리 상에 A 데이터를 한 벌만 가지도록 만든다.

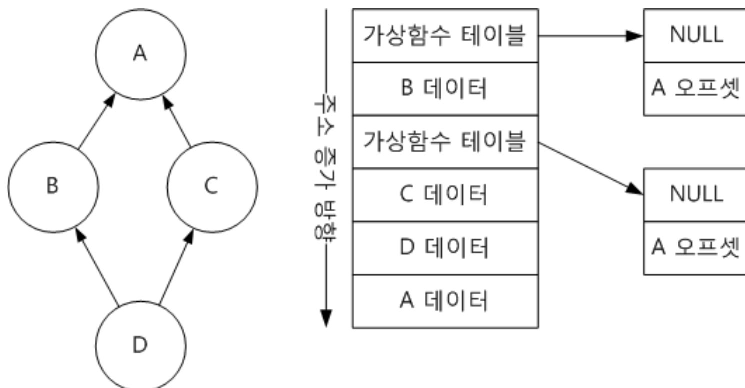


그림 3 가상 상속을 통해 다중 상속을 받은 클래스의 메모리 구조

<그림 3>을 보면 가상 상속 받은 클래스의 경우 메모리 구조가 굉장히 복잡한 것을 알 수 있다. 실제로 메모리 구조 외에도 이를 처리하는 생성/소멸의 메커니즘은 더욱 복잡하다. 우선 가상 상속을 받게 되면 클래스에 virtual 함수가 없더라도 가상함수 테이블이 생성된다. 가상함수 테이블에는 A 클래스의 오프셋이 들어간다. B와 C 클래스는 이를 통해서 실제로 메모리에 존재하는 A 클래스의 데이터를 참조한다.

또한 D 클래스가 초기화될 때, B와 C는 A 클래스를 초기화 시키지 않아야 한다. 물론 B, C 클래스가 선언되어서 초기화 될 때에는 A 클래스를 초기화 해 주어야 한다. 이러한 복잡한 특성을 지원하기 위해서 컴파일러는 B, C, D의 생성자에 인자를 전달하는 방법을 사용한다. 컴파일러는 인자가 0인 경우에는 A 클래스를 초기화 하지 않고, 1인 경우에는 초기화 하는 것과 같은 형태로 처리한다.

다형성

다형성이란 여러 가지 형태를 가졌다는 말이다. 무엇이 여러 가지 형태를 가졌다는 말일까? 동일한 이름을 가진 함수가 여러 가지 형태를 가졌다는 말이다. 아주 단순화시키면 C++에서 같은 이름을 가진 함수를 여러 개 만들 수 있다는 것이라고 말할 수 있다.

이렇게 간단하게 설명을 해버리면 사소한 특징처럼 보인다. 하지만 이는 언어학적으로 굉장히 중요한 특징이다. <리스트 5>에는 다형성을 지원하는 C++로 만든 두 개의 Plus 함수가 있다. 반면 이러한 특징이 지원되지 않는 C에서는 <리스트 6>처럼 함수 이름을 구분 지어서 만들어야 한다. 이제 이 함수들을 학습하는 학습자의 입장에서 생각해보자. C++과 같이 다형성이 지원되는 언어는 뭔가를 더하고 싶다면 Plus를 호출하면 된다고 생각할 수 있다. 하지만 C에서는 int는 PlusInt를, float는 PlusFloat를 사용해야 한다고 일일이 기억해야 한다. 이는 결론적으로 C 버전은 덜 추상화 되었고, 직교성이 떨어진다는 것을 나타낸다.

리스트 5 다형적인 특징을 가진 Plus 함수

```
int Plus(int a, int b);
float Plus(float a, float b);
```

리스트 6 다형적이지 않은 언어에서의 함수들

```
int PlusInt(int a, int b);
float PlusFloat(float a, float b);
```

물론 그렇다고 함수 본문만 다르다고 동일한 함수를 아무 제약 없이 만들 수 있는 것은 아니다. <리스트 7>에 나온 것과 같은 함수들은 생성할 수 없다.

리스트 7 C++에서 만들 수 없는 함수들

```
int Plus(int a, int b); // 기분 좋을 때 더하는 함수
int Plus(int a, int b); // 기분 나쁠 때 더하는 함수
double Plus(int a, int b);
```

그렇다면 C++에서는 왜 이런 제약 사항을 둔 것일까? 첫째로, 리턴 값이 다른 함수를 다형적으로 생성할 수 없도록 한 이유는 컴파일러가 함수 호출을 통해 어떤 함수를 호출할지 결정하는 단계에서 애매모호한 경우가 너무 많기 때문이다. double r = Plus(a,b)와 같은 형태로 호출하는 경우보다는 리턴 값이 생략된 Plus(a,b)와 같은 형태로 호출하는 경우가 많기 때문이다. 두 번째로 함수 원형이 동일한 함수를 다형적으로 생성하지 못하도록 한 것은 언어 자체의 문법적인 제약 사항이 있기 때문이다. <리스트 7>의 동일한 Plus 함수를 상황에 맞게 호출하기 위해서는 추가적인 정보가 필요한데 C++의 함수 호출 문법에는 이러한 정보를 표기할 수 있는 구문이 존재하지 않기 때문이다. 결론적으로 C++의 문법 안에서 컴파일러가 자동적으로 추론할 수 있기 위해서는 어쩔 수 없이 함수 이름, 인자의 순서, 인자의 형태 중 하나는 달라야 하는 것이다.

C++ 컴파일러가 다형성을 구현하기 위해서 사용하는 방법은 의외로 간단하다. C++ 컴파일러는

개발자가 만든 동일한 이름의 함수를 C 버전 함수와 같이 이름이 다른 형태로 만든다. 이름 장식으로 불리는 이 기능은 함수 이름을 컴파일러가 임의로 수정하는 것을 말한다. 즉, 개발자는 Plus란 함수를 만들었지만 그것이 내부적으로는 PlusIntInt와 같은 다른 이름으로 변경되어서 처리된다는 말이다. 컴파일러는 또한 함수 호출을 할 때의 인자를 통해서 자신이 변형한 이름의 함수 중에서 가장 형태가 가까운 것을 선택해서 호출한다. 이 경우에 추론 가능한 호출이 한 가지 이상 존재하는 경우에 컴파일러는 개발자에게 함수 호출이 불분명 하다는 경고 메시지를 표시한다. 이는 은닉성과 마찬가지로 모두 컴파일타임에 결정되는 요소들이기 때문에 다형적인 함수를 지원한다고 해서 런타임 오버헤드가 추가되진 않는다.

클래스에는 가상 함수라는 다형성 메커니즘이 하나 더 있다. 앞서 설명한 함수 오버로딩이 컴파일타임 다형성에 해당한다면 클래스의 가상 함수는 런타임 다형성에 해당한다. <리스트 8>에는 이러한 가상 함수가 추가된 Car, RacingCar 클래스가 나와있다.

리스트 8 virtual 함수가 추가된 Car, RacingCar 클래스

```
class Car
{
    // … 중략 …
    virtual void Accelerate() { Speed(Speed() + 30); }
};

class RacingCar : public Car
{
    // … 중략 …
    virtual void Accelerate() { Speed(Speed() + Buster() + 30); }
};
```

<그림 4>에는 가상 함수가 추가된 Car와 RacingCar 클래스의 메모리 구조가 나와있다. 위쪽에 있는 것이 Car 클래스이고, 아래쪽에 있는 것이 RacingCar 클래스다. 메모리 구조를 살펴보면 vftable이란 것이 추가된 것을 볼 수 있다. 이는 가상 함수 테이블로 각 클래스에서 바인딩될 가상 함수 테이블을 가리킨다. Car 클래스의 vftable에는 Car::Accelerate의 주소가, RacingCar의 vftable에는 RacingCar::Accelerate 주소가 들어있다. 만약 RacingCar에서 Accelerate 함수를 구현하지 않았다면 RacingCar의 vftable에도 Car::Accelerate가 저장된다.

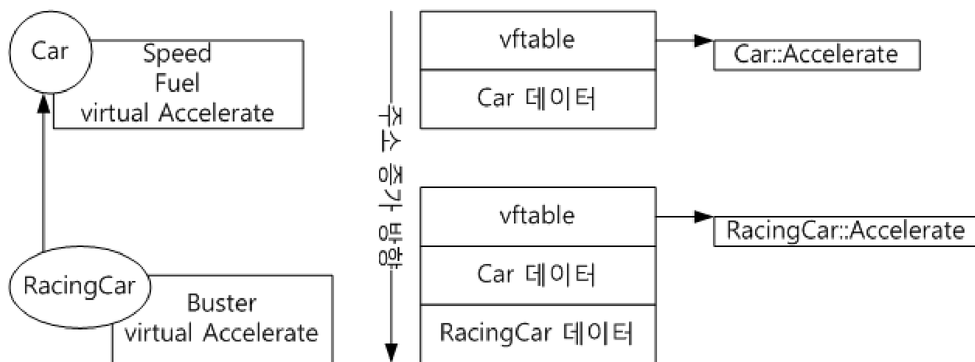


그림 4 Car, RacingCar 메모리 구조

그렇다면 왜 이렇게 가상 함수 테이블을 사용해서 참조하는지 실제로 가상함수를 호출하는 코드를 보면서 살펴보도록 하자. <리스트 9>에는 다양한 방법으로 가상함수를 호출하는 코드가 나와있다. 비슷해 보이는 가상함수 호출이지만 1, 2번과 3, 4번은 틀린 방식으로 처리된다. 우선 1, 2번은 정적으로 가상 함수를 호출하는 경우로 컴파일타임에 모든 정보를 알 수 있다. 따라서 컴파일러는 가상함수 테이블을 참조하지 않고 각각의 클래스에 맞는 함수를 바로 호출한다. 반면 3, 4번의 경우에는 포인터가 가리키는 대상이 무엇인지 컴파일타임에 결정되지 않는다. 이 경우에는 C++ 컴파일러는 참조하고 있는 대상 클래스의 가상함수 테이블을 참조해서 호출하도록 호출 코드를 생성한다. 이 경우에는 가상 함수 테이블을 참조하는 추가적인 연산이 들어가기 때문에 런타임 오버헤드가 발생한다.

리스트 9 가상 함수 호출 코드

```
Car c; c.Accelerate (); // … 1
RacingCar rc; rc.Accelerate (); // … 2

Car *pc;
pc = &c; pc->Accelerate (); // … 3
pc = &rc; pc->Accelerate (); // … 4
```

<그림 5>에는 가상 함수를 포함한 두 클래스로부터 다중 상속을 받은 경우의 메모리 구조가 나와있다. 특징적인 부분은 C 클래스에는 가상 함수 테이블이 두 개가 포함된다는 점이다. A, B와 관련된 가상 함수들을 포함하고 있는 테이블이 분리되어 처리된다. 또 한가지 살펴볼 점은 C에서 구현한 함수는 C에서 구현한 함수가 테이블에 기록되는 반면, C에서 구현되지 않은 함수는 A, B의 함수 주소가 그대로 기록된다는 점이다.

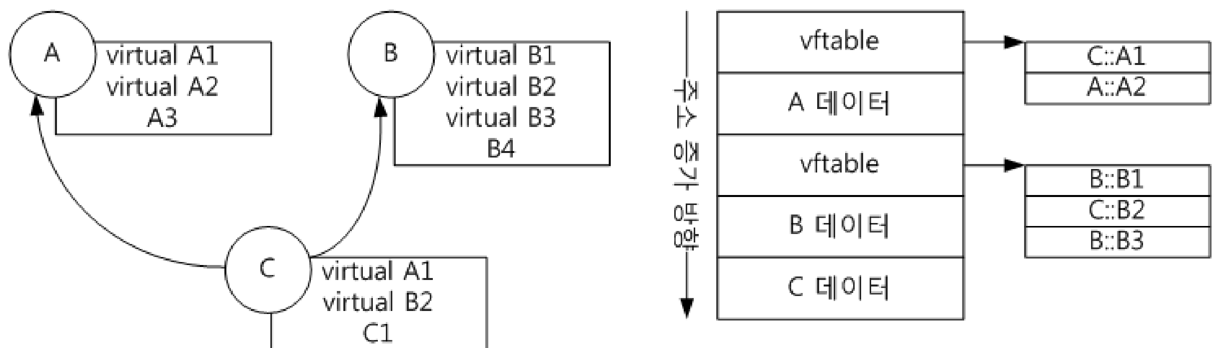


그림 5 virtual 멤버 함수를 포함한 다중 상속 클래스의 메모리 구조

끝으로 가상 함수 테이블을 생성하는 것에 대해서 살펴보자. 가상 함수 테이블은 컴파일 타임에 컴파일러가 생성한다. 각 테이블은 클래스 별로 생성되며, 동일한 클래스의 인스턴스는 동일한 테이블을 참조한다. 즉, 멤버 함수와 같은 식으로 메모리에 배치된다고 생각하면 된다. 또한 각 클래스의 vftable을 초기화하는 것은 해당 클래스의 생성자에서 이루어진다.

마법은 없다.

C++이나 OOP를 처음 접한 많은 개발자들은 "C++은 …", "OOP는 …"이라는 말을 굉장히 자주 한다. 그러면서 다른 언어들을 비방하거나, 다른 비주류 프로그래밍 패러다임을 무시하는 말들을 하기도 한다. 그들에게 C++, OOP는 마법 같은 언어, 패러다임이기 때문이다.

하지만 앞서 살펴본 몇 가지 중요한 OOP 메커니즘을 C++ 컴파일러가 구현하는 방법만 보더라도 그 속에는 마법이 없다는 것을 알 수 있다. 우리가 해야 하는 수많은 귀찮은 일을 컴파일러가 대신 해주는 것뿐이다. 그곳엔 단지 조금 똑똑한 컴파일러, 조금 편리한 표현식이 존재할 뿐이다. 좀 더 많은 개발자들이 무대 뒤편에서 벌어지는 일에도 관심을 가졌으면 하는 생각을 해본다.

참고자료

EFFECTIVE C++,2nd Edition
Scott Meyers, Addison-Wesley Professional

More Effective C++
Scott Meyers, Addison-Wesley Professional

Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions
Herb Sutter, Addison-Wesley Professional

More Exceptional C++
Herb Sutter, Addison-Wesley Professional

Exceptional C++ Style : 40 New Engineering Puzzles, Programming Problems, and Solutions
Herb Sutter, Addison-Wesley Professional

Efficient C++ : Performance Programming Techniques
Dov Bulka, David Mayhew, Addison-Wesley Professional