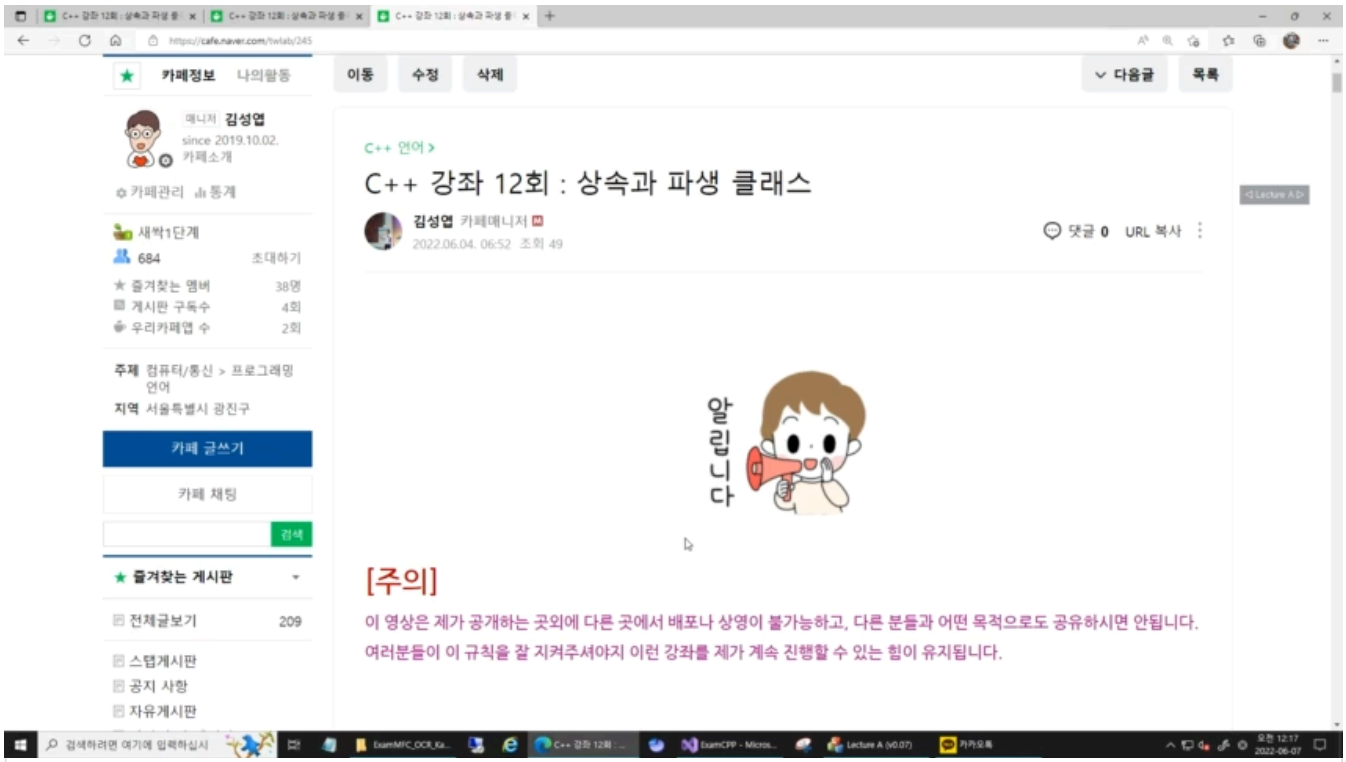
C++ 강좌 12회 : 상속과 파생 클래스

김성엽 카페매니저 + 구독 1:1 채팅
2022.06.04. 06:52 조회 346

[주의]

이 영상은 제가 공개하는 곳외에 다른 곳에서 배포나 상영이 불가능하고, 다른 분들과 어떤 목적으로도 공유하시면 안됩니다. 여러분들이 이 규칙을 잘 지켜주셔야지 이런 강좌를 제가 계속 진행할 수 있는 힘이 유지됩니다.

이번 강좌에서는 상속(inheritance)과 파생 클래스(Derived Classes)에 대해 설명하겠습니다.



C++ 강좌 12회 : 상속과 파생 클래스

1. 중복 코드가 생기는 또 다른 이유

프로그램에 **중복된 소스 코드가 많다는 뜻은 프로그램의 기능에 변화가 생기면 변경할 코드가 많다는 뜻**입니다. 그래서 프로그래밍 언어는 소스 코드를 작성할 때 **동일하거나 유사한 소스 코드가 반복해서 나열되는 것은 줄이기 위해 함수, 반복문, 구조체 같은 기술을 제공**하는 것입니다.

따라서 저는 소스 코드의 중복을 줄일 수 있는 문법을 적극적으로 사용해서 **중복 코드를 만들지 않는 것이 개발자가 지켜야할 기본적인 자세**라고 생각합니다. 하지만 저처럼 전문적으로 프로그램을 개발하는 사람들에게 **프로그램 개발은 일**이기 때문에 이런 기본 원칙에 대한 생각이 흔들릴 때가 있습니다. 왜냐하면 프로그램 개발을 일의 관점에서보면 단순히 기능 구현만 중요한 것이 아니라 **개발 기간과 비용 그리고 효율성과 안정성도 고려**해야 하기 때문입니다.

예를 들어, 두 개의 정숫값을 더하는 기능이지만, 고객이 기능 변경을 요구할 가능성이 있다고 판단해서 아래와 같이 해당 기능을 함수로 만들었다고 합시다. 그리고 Add 함수를 믿고 프로그램의 많은 곳에서 이 함수를 호출해서 사용했다고 합시다. (이 예시에서는 a+b처럼 간단한 코드를 사용했지만 이 코드가 50줄 이상이 될수도 있다고 생각해야 합니다.)

```
int Add(int a, int b)
{
    return a + b;
}
```

그런데 예상대로 고객이 덧셈 기능에 대해 문제를 제기했습니다. 정숫값을 더할 때 해당 값이 음수라면 양수로 변경해서 더해야 하다고 기능 변경을 요청한 것입니다. 다행이 정숫값을 더하는 기능을 함수로 만들었기 때문에 아래와 같이 Add 함수에 조건문을 추가해서 이 기능 변경에 대처할 수 있습니다.

```
int Add(int a, int b)
{
    if(a < 0) a = -a;  // a 변수가 음수이면 양수로 변경한다.
    if(b < 0) b = -b;  // b 변수가 음수이면 양수로 변경한다.
    return a + b;
}
```

하지만 무조건 음수를 양수로 변경해서 더하면 안되고 상황에 따라 적용해야 한다면 어떻게 해야 할까요? 즉, 위 함수에 추가된 조건문을 무조건 사용하면 안되고 선택적으로 사용해야 한다고 합시다. 학생들을 가르치는 입장에서는 아래와 같이 **a_use_abs_flag 매개 변수를 추가하고 이 변수 값이 1이면 조건문을 사용하게 하고 0이면 조건문을 처리하지 않게 함수를 수정**해서 해결할 것입니다.

```
int Add(int a, int b, int a_use_abs_flag)
{
    // 음수를 양수로 변경하는 기능 사용 여부 (0은 사용 안 함, 1은 사용 함)
    if(a_use_abs_flag == 1) {
        if(a < 0) a = -a;  // a 변수가 음수이면 양수로 변경한다.
        if(b < 0) b = -b;  // b 변수가 음수이면 양수로 변경한다.
    }
    return a + b;
}
```

그리고 이렇게 수정하면 Add 함수의 코드만 수정해서 되는 문제가 아니라, **Add 함수를 호출하는 코드에도 인자를 한 개씩 추가해야 하기 때문에 Add 함수를 호출하는 코드를 모두 찾아서 수정**해야 합니다. 예를 들어, 기존에는 Add(2, 3)처럼 함수를 호출했었는데, Add 함수가 호출되는 상황이 단순 덧셈이면 Add(2, 3, 0)으로 변경되어야하고 음수를 양수로 변경한뒤 덧셈을 해야 한다면 Add(2, 3, 1)로 변경되어야 한다는 뜻입니다.

그런데 이 상황이 업무 상황이고 **Add 함수를 호출한 곳이 많은데 비해 상대적으로 음수를 양수로 변경할 곳이 적어**도 개발자들이 위와 같은 선택을 할까요? 아마도 중복 코드가 발생하는 상황을 무시하고 Add 함수의 내용을 복사해서 아래와 같이 AddAbs 함수를 추가하고 필요한 곳에만 Add 함수를 AddAbs 함수로 변경해서 호출하는 개발자도 있을 것입니다.

```
int Add(int a, int b)
{
    return a + b;
}

int AddAbs(int a, int b)
{
    if(a < 0) a = -a;  // a 변수가 음수이면 양수로 변경한다.
    if(b < 0) b = -b;  // b 변수가 음수이면 양수로 변경한다.
    return a + b;
}
```

중복 코드를 만드는 것이 좋은 방법은 아니지만 본인이 개발자이고 일의 특성상 아래와 같은 상황이 추가된다고 했을 때, 이 원칙을 지킬 것인지 스스로 한 번 생각해 보세요.

◆ Add 함수를 호출한 곳이 1000 번 이상인데, 음수를 양수로 변경해서 더하는 경우가 5번 미만인 경우
◆ Add 함수가 다른 프로젝트에도 사용되어 버전 관리를 위해 다른 프로젝트에 있는 소스도 수정이 필요한 경우
◆ 협조 관계가 좋지 않은 다른 팀과 이 함수를 공유하고 있어서 함수 호출 방법이 달라졌다는 것을 알리고 수정을 해야 한다고 이야기 해야 하는 경우
◆ 기능은 추가해야 하는데 테스트 일정이 정해져 있어서 시간이 부족한 경우
◆ 이 작업에 대해 별도의 비용이 추가되지 않는 경우

어떤 단계에서 원칙을 무시하기로 마음을 먹으셨나요? 만약, 위 조건에도 원칙을 지키기는 것을 선택했다면, 본인의 마음은 편할지 모르겠지만 함께 일하는 개발자들의 마음도 생각하는 것이 좋습니다

결국 공부와 달리 일이라는 관점에서는 **이돈에 없는 조건이 반영되어 원칙은 지킬 수 없는 상황이 발생**할 수 있습니다. 개발자의 무지와 귀찮음 때문에 발생한 중복 코드는 분명 문제이지만, 위와 같은 상황으로 발생한 중복 코드가 정말 개발자의 잘못이라고만 이야기할 수 있을까요?

그래서 저는 개발자들이 일을 하면서 어쩔수 없이 발생하는 중복 코드는 어느정도 인정하는 편입니다. 그리고 **C 언어는 이런 중복 코드에 대한 문제를 전적으로 개발자가 해결해야 한다는 입장**이고 실제로 실력이 좋아지면 대부분 해결됩니다. 하지만 전문가의 길은 길고 멀기 때문에 아직 실력이 부족한 개발자들이 이런 문제로 고민하고 스트레스를 받는 것은 정말 안타까운 일입니다. 그래서 초, 중급 개발자들에게 위안이 될만한 다른 관점에서 중복 코드에 대한 이야기를 해보겠습니다.

2. 중복 코드를 없애는 것이 정말 최선인가?

실무자의 입장에서 볼때 소스 코드에서 중복 코드는 무조건 나쁘기 때문에 줄여야 한다는 이론적인 주장보다는 **중복 코드가 생기더라도 문제가 되지 않도록 잘 관리하면 된다고 생각**합니다.

그리고 운영 환경(OE)이나 인프라를 구축하는 프로그래밍을 하다보면 위에 제시한 조건과 달리 중복 코드에 대한 선택이 불가능한 경우도 있습니다. 예를 들어, 어떤 함수가 반복적으로 매우 빠르게 수행되어야 하는 경우, 이 **함수에 매개 변수나 조건문이 추가되면 속도 저하의 원인이 되기 때문에 기존 함수의 성능을 유지하려고 비슷한 함수를 추가해서 중복 코드가 만들어지는 경우**도 있습니다. 즉, 코드 관리보다 수행 성능이 더 중요한 경우도 있다는 뜻입니다.

결국 일을 하다보면 중복 코드는 발생할 수 밖에 없기 때문에 이런 상황으로 중복 코드가 추가되는 것에 대해 개발자가 부끄럽게 생각하거나 이론가들에게 비난받는 건 옳지 않다고 생각합니다. 그래서 저는 중복 코드에 대한 문제점을 원점에서 다시 고민해야 한다고 생각합니다.

프로그래밍 언어에서 중복 코드를 줄이라고 권장하는 이유는 프로그램의 **성능적인 문제가 아니라 소스 편집과 관련된 문제**입니다. 중복 코드가 많으면 **코드가 길어져서 소스 코드를 파악하기 불편**해지고 **기능 변경이 발생했을 때 변경해야 할 코드가 많아지기 때문**에 중복 코드를 줄이라고 하는 것입니다. 예를 들어, A 영역의 코드와 B 영역의 코드가 중복 코드라면 A 영역의 코드를 수정하게되면 같은 역할을 하는 B 영역의 코드도 함께 수정해야 하기 때문에 프로그램에 중복 코드가 있으면 좋지 않다는 뜻입니다.

그런데 이런 이유라면 중복 코드를 개발자가 직접 제거하는 방식도 가능하지만 **중복 코드를 감추어서 해결**하는 방법도 있습니다. 예를 들어, **A 영역의 코드는 그대로 유지하고 B 영역에는 'A 영역의 코드와 동일' 같은 상징적인 문구**를 적어놓습니다. 그리고 나중에 C++ 컴파일러가 소스를 해석할 때 'A 영역의 코드와 동일'이라는 문구를 만나면 A 영역의 코드를 가져와서 소스를 해석한다면 B 영역에 적어야할 중복 코드를 감추는 방법도 있다는 뜻입니다.

이렇게 중복 코드를 숨기는 방식을 사용하면 **A 영역의 코드가 수정되어도 B 영역에는 직접적인 코드 대신에 'A 영역의 코드와 동일'이라고 적혀있기 때문에 B 영역의 코드는 수정할 필요가 없습니다.** 나중에 C++ 컴파일러가 소스를 해석할 때 B 영역의 코드는 A 영역의 코드를 가져와서 컴파일하기 때문에 변경된 A 영역의 코드가 그대로 적용됩니다. 즉, 중복 코드에 대한 문제는 단순한 편집에 대한 문제이기 때문에 **중복 코드가 있어도 해당 코드가 눈에 보이지 않는다면 코드 유지 보수에 문제가 되지 않습니다.**

결국 중복 코드의 편집 문제는 개발자가 직접 제거하지 않더라도 C++ 컴파일러의 도움을 받아서 해결할 수 있습니다. 좀더 정확하게 이야기 하면 C++ 언어가 제공하는 상속 문법이 이 기능을 제공합니다. 즉, **상속은 중복 코드가 존재하지만 해당 코드가 보이지 않도록 만들어서 중복 코드의 문제를 다른 시각으로 해결한 문법**입니다. 따라서 C++ 언어를 사용하는 초, 중급 개발자가 상속 문법을 사용하면 지금까지 이야기한 중복 코드에 대한 비난을 간단하게 회피할 수 있습니다.

3. 상속 (클래스 복사 기술)

C 언어에서는 중복 코드의 단위가 함수 단위지만, **C++ 언어에서는 중복 코드의 단위가 클래스 단위**로 커지기 때문에 C++ 언어가 되면서 중복 코드는 더 심각한 문제가 되었을 것입니다. 예를 들어, 아래와 같이 멤버의 수를 기억하는 Tipsware 클래스가 있다고 가정하겠습니다.

```
#include <stdio.h>    // printf 함수를 사용하기 위해

class Tipsware
{
private:
    int m_member_count;  // 멤버의 수를 저장할 변수

public:
    Tipsware()  // 생성자
    {
        m_member_count = 0;
    }

    void SetMemberCount(int a_count)  // 멤버의 수를 저장할 때 사용할 함수
    {
        m_member_count = a_count;
    }

    int GetMemberCount()  // 멤버의 수를 얻을 때 사용할 함수
    {
        return m_member_count;
    }
};

int main()
{
    Tipsware data;
    data.SetMemberCount(31);  // 객체에 멤버의 수를 지정한다.

    printf("Member Count : %d\n", data.GetMemberCount());  // 객체에 저장된 멤버의 수를 출력한다.
    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
Member Count : 31

이 창을 닫으려면 아무 키나 누르세요...
```

위 코드에서 main 함수에 사용된 printf 함수 작업을 Tipsware 클래스에서 직접 제공했으면 좋겠다는 의견이 나왔다고 합시다. 즉, 아래와 같이 Tipsware 클래스에 ShowMemberCount 함수를 추가해서 사용하면 이 클래스를 사용하는 개발자가 더 편할 것이라는 의견입니다.

```
#include <stdio.h>    // printf 함수를 사용하기 위해

class Tipsware
{
private:
    int m_member_count;  // 멤버의 수를 저장할 변수

public:
    Tipsware()  // 생성자
    {
        m_member_count = 0;
    }

    void SetMemberCount(int a_count)  // 멤버의 수를 저장할 때 사용할 함수
    {
        m_member_count = a_count;
    }

    int GetMemberCount()  // 멤버의 수를 얻을 때 사용할 함수
    {
        return m_member_count;
    }

    void ShowMemberCount()  // 멤버의 수를 출력할 때 사용할 함수
    {
        printf("Member Count : %d\n", m_member_count);
    }
};

int main()
{
    Tipsware data;
    data.SetMemberCount(31);  // 객체에 멤버의 수를 지정한다.
    data.ShowMemberCount();  // 객체에 저장된 멤버의 수를 출력한다.
    return 0;
}
```

그런데 위 예제에서는 설명을 위해 Tipsware 클래스의 소스가 함께 있는 것처럼 적었지만, 만약 Tipsware 클래스가 라이브러리에서 제공되는 클래스라서 직접 수정이 불가능하다면 어떻게 해야 할까요?

일부 개발자들은 Tipsware 클래스의 소스 코드가 변경이 불가능하다면 그냥 예전처럼 main 함수에서 printf 함수를 직접 사용하자고 할 것입니다. 또 일부 무모한 개발자들은 편해져야 한다는 이유로 아래와 같이 Tipsware 클래스 소스를 그대로 복사해서 Tipssoft라는 클래스를 만들고 해당 클래스에 ShowMemberCount 함수를 추가해서 Tipsware 클래스 대신 Tipssoft 클래스를 사용하자고 할 것입니다.

```cpp
#include <stdio.h>    // printf 함수를 사용하기 위해

class Tipsware
{
private:
    int m_member_count;  // 멤버의 수를 저장할 변수

public:
    Tipsware()  // 생성자
    {
        m_member_count = 0;
    }

    void SetMemberCount(int a_count)  // 멤버의 수를 저장할 때 사용할 함수
    {
        m_member_count = a_count;
    }

    int GetMemberCount()  // 멤버의 수를 얻을 때 사용할 함수
    {
        return m_member_count;
    }
};

class Tipssoft
{
private:
    int m_member_count;  // 멤버의 수를 저장할 변수

public:
    Tipssoft()  // 생성자
    {
        m_member_count = 0;
    }

    void SetMemberCount(int a_count)  // 멤버의 수를 저장할 때 사용할 함수
    {
        m_member_count = a_count;
    }

    int GetMemberCount()  // 멤버의 수를 얻을 때 사용할 함수
    {
        return m_member_count;
    }

    void ShowMemberCount()  // 멤버의 수를 출력할 때 사용할 함수
    {
        printf("Member Count : %d\n", m_member_count);
    }
};

int main()
```

```
{
    Tipssoft data;
    data.SetMemberCount(31);  // 객체에 멤버의 수를 지정한다.
    data.ShowMemberCount();  // 객체에 저장된 멤버의 수를 출력한다.
    return 0;
}
```

위 코드를 보면 알겠지만 C 언어에서는 코드의 중복이란 문제가 단순히 함수 단위로 발생하지만 **C++ 언어는 코드 중복의 단위가 클래스 단위이기 때문에 중복의 범위가 넓습니다.**

물론 조금만 개념있는 개발자라면 자료형의 중첩(nested) 표현을 사용해서 이 중복 코드의 문제를 아래와 같이 조금은 해결할 수 있습니다. 즉, Tipssoft 클래스의 코드가 Tipsware 클래스로 선언된 객체를 사용하도록 변경했기 때문에 **기능 변화에 따른 중복 편집에 대한 문제점은 줄었습니다.** 하지만 소스 코드가 여전히 길고, Tipssoft 클래스에 추가된 함수들이 중복된 이름의 함수를 호출하기 때문에 코드를 이해하기는 더 어려워졌습니다. 그리고 **함수의 중복 호출로 프로그램의 수행 성능은 더 나빠지게 됩니다.**

왜냐하면 m_member_count 변수는 Tipsware 클래스의 SetMemberCount, GetMemberCount 함수로 이미 보호되어 지고 있는데 Tipssoft 클래스에서 이 함수들을 다시 Tipssoft 클래스의 SetMemberCount, GetMemberCount 함수로 중복해서 보호하기 때문에 의미가 약한 이중 보호 코드가 되어 버린 것입니다.

예를 들어, m_member_count에 31 값을 대입하려면 Tipssoft 클래스가 제공하는 SetMemberCount 함수가 호출된 다음에 다시 Tipsware 클래스가 제공하는 SetMemberCount 함수가 호출되어야지 m_member_count 변수에 31 값이 저장되기 때문에 의미없이 함수가 중복 호출되는 상황입니다. 물론 객체 지향을 극도로 따지는 이론가들은 이것이 안정 장치를 이중으로 걸기 때문에 더 효과적이라고 주장하지만 실무자 입장에서 봤을 때는 그냥 과도한 안정 장치일 뿐입니다.

```
#include <stdio.h>    // printf 함수를 사용하기 위해

class Tipsware
{
private:
    int m_member_count;  // 멤버의 수를 저장할 변수

public:
    Tipsware()  // 생성자
    {
        m_member_count = 0;
    }

    void SetMemberCount(int a_count)  // 멤버의 수를 저장할 때 사용할 함수
    {
        m_member_count = a_count;
    }

    int GetMemberCount()  // 멤버의 수를 얻을 때 사용할 함수
    {
        return m_member_count;
    }
};

class Tipssoft
{
private:
    Tipsware m_data;

public:
    Tipssoft()  // 생성자
    {
    }

    void SetMemberCount(int a_count)  // 멤버의 수를 저장할 때 사용할 함수
    {
        m_data.SetMemberCount(a_count);
    }

    int GetMemberCount()  // 멤버의 수를 얻을 때 사용할 함수
    {
        return m_data.GetMemberCount();
    }

    void ShowMemberCount()  // 멤버의 수를 출력할 때 사용할 함수
    {
        printf("Member Count : %d\n", m_data.GetMemberCount());
    }
};

int main()
{
```

```
    Tipssoft data;
    data.SetMemberCount(31);  // 객체에 멤버의 수를 지정한다.
    data.ShowMemberCount();  // 객체에 저장된 멤버의 수를 출력한다.
    return 0;
}
```

그래서 C++ 언어는 중첩 표현의 장점은 살리고 단점을 줄이는 방법으로 클래스 복사 기술인 상속 문법을 제공합니다. 이제 상속 문법을 사용해서 Tipsware 클래스와 Tipssoft 클래스의 관계를 다시 정리해 보겠습니다. 위 코드에서 Tipssoft 클래스는 Tipsware 클래스의 내용은 모두 포함하고 ShowMemberCount 함수만 추가되는 관계입니다. 따라서 Tipssoft 클래스는 Tipsware 클래스를 복사한 다음 ShowMemberCount 함수만 추가하면 됩니다.

이 표현을 C++ 문법을 사용해서 표현하면 다음과 같습니다. 아래의 코드에서 class Tipssoft : Tipsware라고 사용된 문법이 상속 문법이고 Tipsware 클래스의 내용은 Tipssoft 클래스에 그대로 복사하겠다는 뜻입니다. 즉, 중복 코드를 C++ 컴파일러가 소스를 해석할 때 추가하겠다는 뜻이기 때문에 눈에 보이는 Tipssoft 클래스 코드는 이전 코드와 비교했을 때 매우 단순해 졌습니다. 그리고 이때 Tipsware 클래스의 내용이 Tipssoft 클래스로 복사되기 때문에 Tipssoft 클래스가 Tipsware 클래스의 소스를 사용하는 것처럼 생각해서 Tipsware 클래스를 '부모 클래스'라고 이야기 하고 Tipssoft를 '자식 클래스'라고 이야기 합니다.

```
#include <stdio.h>    // printf 함수를 사용하기 위해

class Tipsware
{
private:
    int m_member_count;  // 멤버의 수를 저장할 변수

public:
    Tipsware()  // 생성자
    {
        m_member_count = 0;
    }

    void SetMemberCount(int a_count)  // 멤버의 수를 저장할 때 사용할 함수
    {
        m_member_count = a_count;
    }

    int GetMemberCount()  // 멤버의 수를 얻을 때 사용할 함수
    {
        return m_member_count;
    }
};

// Tipsware 클래스의 내용을 Tipssoft 클래스에 그대로 복사해달라는 뜻!
class Tipssoft : Tipsware
{
public:
    void ShowMemberCount()  // 멤버의 수를 출력할 때 사용할 함수
    {
        printf("Member Count : %d\n", m_member_count);
    }
};

int main()
{
    Tipssoft data;
    data.SetMemberCount(31);  // 객체에 멤버의 수를 지정한다.
    data.ShowMemberCount();  // 객체에 저장된 멤버의 수를 출력한다.
    return 0;
}
```

하지만 위 코드는 아래와 같이 오류가 발생합니다. 왜냐하면 구조체나 클래스를 복사할 때 상속 등급이 적용되기 때문입니다.

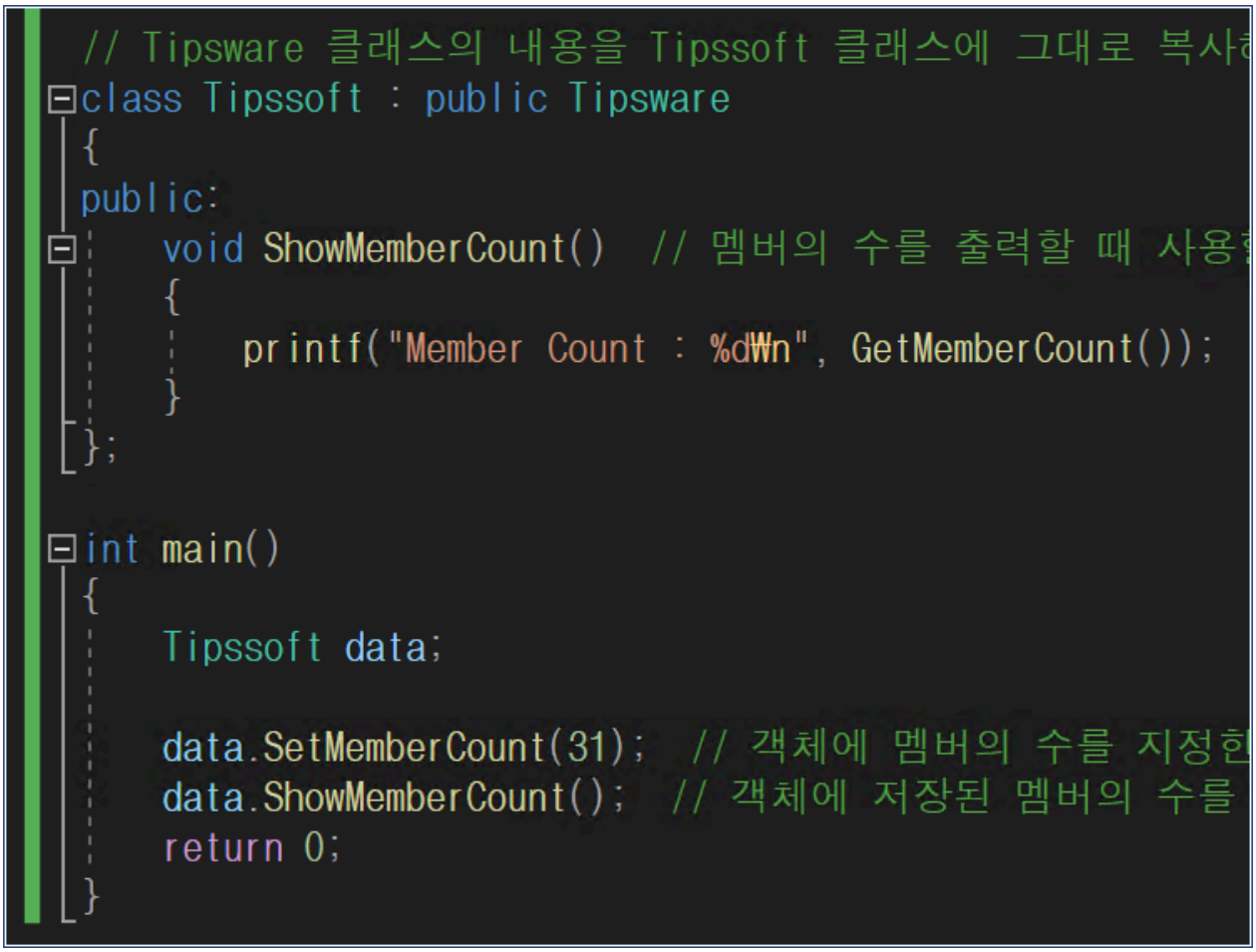
```
class Tipssoft : Tipsware
{
public:
    void ShowMemberCount()  // 멤버의 수를 출력할 때 사
    {
        printf("Member Count : %d₩n", m_member_count);
    }
};

int main()
{
    Tipssoft data;
    data.SetMemberCount(31);  // 객체에 멤버의 수를 지정
    data.ShowMemberCount();  // 객체에 저장된 멤버의 수
    return 0;
}
```

'상속 등급'은 '접근 지정자'의 단계에 영향을 주며 아래와 같이 상속 문법을 적을 때 원본과 사본 클래스의 이름 사이에 적습니다. 상속 등급은 public, protected, private을 사용할 수 있고 아래와 같이 public 상속 등급을 사용하면 Tipsware 클래스의 접근 지정자 등급을 그대로 유지하면서 복사하겠다는 뜻입니다.

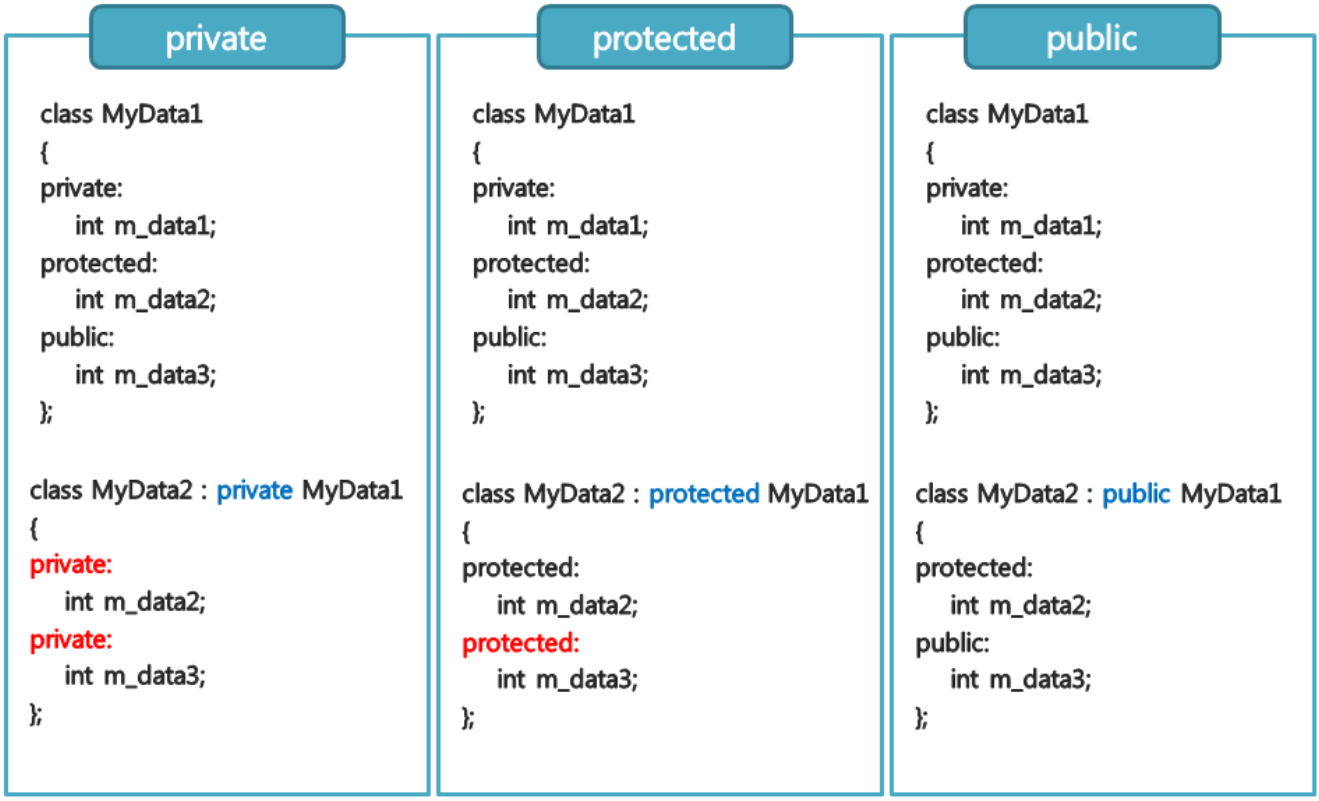
하지만 위와 같이 public 상속 등급으로 복사해도 Tipsware 클래스의 private 멤버는 직접 사용할 수 없는 상태가 됩니다. 이 상황을 오해하는 분들이 많은데 private 멤버가 복사되지 않아 사용을 못하는 것이 아니라, 복사는 되었지만 C++ 문법이 직접 사용을 허락하지 않는 것입니다. 따라서 Tipssoft 클래스에서 Tipsware 클래스에 private 접근 지정자로 선언된 m_memer_count 변수를 사용하면 오류가 발생하는 것입니다.

따라서 아래와 같이 public 상속 등급으로 복사하는 코드를 추가하더라도 Tipssoft 클래스에 사용된 m_member_count 변수는 계속 오류 상태입니다. 따라서 이 오류를 해결하기 위해서는 Tipsware 클래스의 private 접근 지정자로 선언된 변수를 직접 사용하지 말고 Tipsware 클래스에서 복사한 GetMemberCount 함수를 사용하면 됩니다. GetMemberCount 함수는 public 접근 지정자로 선언되었기 때문에 Tipssoft 클래스에서도 public 특성으로 사용이 가능합니다. 그리고 main 함수에 사용된 SetMemberCount 함수도 Tipsware 클래스에 public 접근 지정자로 선언되었기 때문에 Tipssoft 클래스를 사용하는 코드에서도 public 특성으로 사용됩니다. 따라서 main 함수의 data.SetMemberCount(); 코드에 발생했던 오류도 함께 사라지게 됩니다.



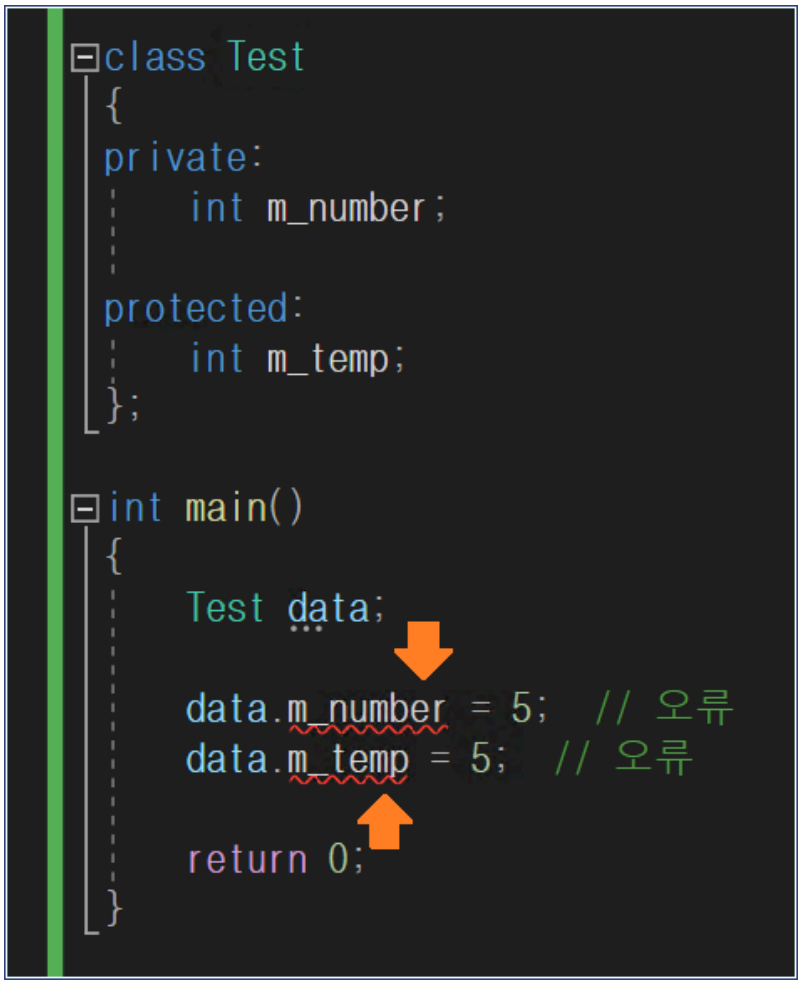
그리고 protected 상속 등급을 사용하면 Tipsware 클래스에 사용된 접근 지정자 등급이 protected이면 그대로 유지되지만 public이면 protected로 변경됩니다. 그리고 private 상속 등급을 사용하면 Tipsware 클래스에 사용된 접근 지정자 등급이 protected, public 둘다 private로 변경됩니다.

MyData1 클래스에서 MyData2 클래스의 상속할 때 상속 등급에 따라 MyData1에 사용된 접근 지정자가 MyData2 클래스에 어떻게 영향을 주는지 그림으로 그려보면 다음과 같습니다. (이 그림에서는 MyData2에 m_data2, m_data3 변수가 선언되는 것처럼 그렸지만 내부적으로 저렇게 복사된다는 의미로 적은 것이니 오해없기를 바랍니다.) 그리고 C++ 언어에서는 상속 등급을 생략하면 클래스 상속에서는 private 상속 등급이 생략된 것이고 구조체 상속에서는 public 상속 등급이 생략된 것으로 처리됩니다. 따라서 이전에 사용했던 class Tipssoft : Tipsware 코드는 class Tipssoft : private Tipsware라고 적은 것과 동일합니다.



그런데 제가 이전 강좌에서 접근 지정자를 설명하면서 protected 접근 지정자는 설명을 생략했습니다. 왜냐하면 protected 접근 지정자는 상속 관계에서만 의미가 있기 때문에 단독으로 사용되는 클래스에서는 private와 동일한 의미라서 설명을 하지 않은 것입니다. 하지만 이제 상속을 배웠으니 private와 protected 접근 지정자의 차이에 대해서 설명하겠습니다.

기본적으로 private, protected 접근 지정자로 선언된 멤버 변수는 객체 외부에서 직접 사용하는 것이 불가능합니다. 즉, 아래와 같이 private, protected 접근 지정자로 선언된 변수는 객체 외부에서 항목 지정 연산자(.)를 사용해서 값을 대입하면 둘 다 오류 처리됩니다.



그런데 상속 문법이 적용된 클래스에 대해서는 private, protected 접근 지정자의 특성에 차이가 있습니다. 아래의 그림처럼 부모 클래스인 Test에서는 멤버 함수에서 private, protected 접근 지정자로 선언된 멤버 변수 사용에 제한이 없습니다. 하지만 Test 클래스에서 상속받은 NewTest 클래스에서는 public 상속 등급을 사용해도 부모 클래스의 private 멤버는 사용할 수 없습니다. 그래서 NewShowData 함수에서 m_temp는 사용해도 오류가 없지만 m_number는 오류 처리됩니다.

따라서 Test 클래스의 m_number 변수 값을 NewTest 함수에서 사용하려면 아래와 같이 Test 클래스에 GetNumber 함수를 public 접근 지정자로 추가하고 NewShowData 함수에서 m_number 대신 GetNumber 함수를 사용하면 됩니다. 따라서 부모 클래스에 선언된 멤버 변수에 접근할 때 함수로 접근하는 것이 불편하다면 protected 접근 지정자를 사용하면 되고 원칙을 지켜 부모, 자식간에도 변수를 분리해서 사용하고 싶다면 private 접근 지정자를 사용하면 됩니다.

```
class Test
{
private:
    int m_number;

protected:
    int m_temp;

public:
    int GetNumber()
    {
        return m_number;
    }

    void OrgShowData()
    {
        printf("%d, %d\n", m_number, m_temp);
    }
};

class NewTest : public Test
{
public:
    void NewShowData()
    {
        printf("%d, %d\n", GetNumber(), m_temp);
    }
};
```

C++ 언어에서 제공하는 '객체 지향' 의미를 너무 적극적으로 받아들인 개발자들은 상속 여부와 상관없이 모든 멤버 변수를 private 접근 지정자에 선언해서 사용합니다. 저도 C++ 언어를 배워서 열심히 사용할 때 그랬던 기억이 있으니 많은 분들이 그런 과정을 거쳤으리라 생각합니다. 그리고 지금도 원칙적으로 private 접근 지정자를 사용하는 것에 반대하지는 않습니다.

하지만 자식 클래스에서 부모 클래스의 private 접근 지정자에 선언된 변수는 함수를 사용해서 접근해야 하기 때문에 당연히 함수 호출 빈도가 높아집니다. 따라서 이렇게 private 접근 지정자를 적극적으로 사용하면 변화 대처나 융통성에 관련되서는 이론적으로 높은 점수를 받을지 모르겠지만 프로그램의 수행 능력 저하라는 부작용도 발생합니다. 그렇기 때문에 무조건 private 접근 지정자를 사용하겠다는 원칙보다는 상황에 맞춰 private, protected 접근 지정자를 잘 섞어서 사용하는 것이 좋습니다. 그리고 프로그램을 다양하게 해본 개발자로써 추천한다면 protected 접근 지정자를 사용하는 것을 기본으로 하고 특수한 경우에만 private 접근 지정자를 사용하는 것이 더 좋다고 생각합니다.

그리고 상속이라는 기술이 클래스를 복사하는 개념이기 때문에 어차피 자신에게 복사된 멤버 변수를 사용하지 못하게 막고 함께 복사된 함수를 통해서 접근하는 것이 과연 올바른 방법인지도 생각해 봐야 합니다. 결국 private 접근 지정자를 너무 열심히 사용하면 중첩(nested) 표현의 분편함은 개선하기 위해 제공된 상속 표현의 장점이 줄어든게 됩니다.

프로그래밍 언어가 제공하는 문법이나 개념(이론)은 개발자가 좋은 스타일로 개발을 할 수 있도록 도와주기 위해 만들어진 것이지 문법이나 개념을 잘 지키기 위해 성능을 포기하고 불편함을 감수하면서까지 반드시 지켜야 할 필요는 없습니다. 그냥 개발자가 상황에 따라 잘 판단해서 사용하면 되는 것입니다.

따라서 위에서 소개했던 Tipssoft 클래스에서 GetMemberCount 함수를 사용하지 않고 m_member_count 변수를 직접 사용하고 싶다면 아래와 같이 Tipsware 클래스에서 m_member_count 변수를 선언할 때 protected 접근 지정자를 사용하면 됩니다.

```
#include <stdio.h>    // printf 함수를 사용하기 위해

class Tipsware
{
protected:
    int m_member_count;  // 멤버의 수를 저장할 변수

public:
    Tipsware()  // 생성자
    {
        m_member_count = 0;
    }

    void SetMemberCount(int a_count)  // 멤버의 수를 저장할 때 사용할 함수
    {
        m_member_count = a_count;
    }

    int GetMemberCount()  // 멤버의 수를 얻을 때 사용할 함수
    {
        return m_member_count;
    }
};

// Tipsware 클래스의 내용을 Tipssoft 클래스에 그대로 복사해달라는 뜻!
class Tipssoft : public Tipsware
{
public:
    void ShowMemberCount()  // 멤버의 수를 출력할 때 사용할 함수
    {
        printf("Member Count : %d\n", m_member_count);  // 부모 클래스의 protected 멤버는 직접 사용
    }
};

int main()
{
    Tipssoft data;

    data.SetMemberCount(31);  // 객체에 멤버의 수를 지정한다.
    data.ShowMemberCount();  // 객체에 저장된 멤버의 수를 출력한다.
    return 0;
}
```

4. 파생 클래스

보통의 경우 상속은 한 개의 부모 클래스를 복사해서 다든 한 개의 자식 클래스를 만드는 작업입니다. 하지만 아래와 같이 한 개의 부모 클래스(Pet)를 복사해서 여러 개의 자식 클래스(Dog, Cat, Hamster)를 만드는 것도 가능합니다. 이때, 이렇게 **클래스를 복사해서 새로운 클래스를 만드는 개념이 상속**이고 **상속을 통해 만들어진 Dog, Cat, Hamster 같은 클래스를 파생 클래스(Derived Classes)**라고 부릅니다.

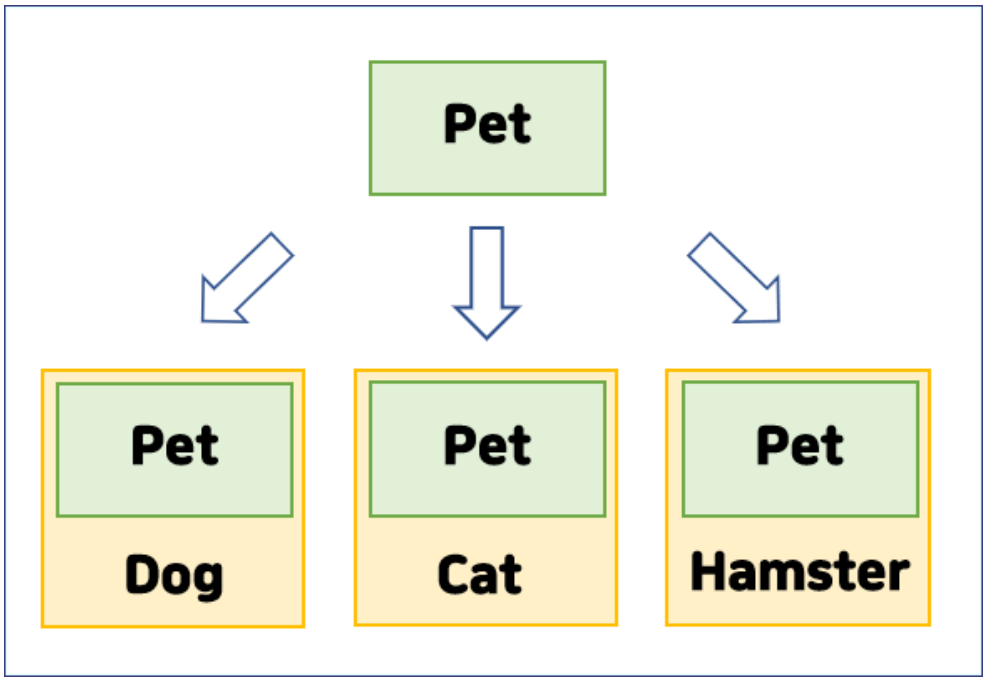
```
class Pet
{
    /* 생략 */
};

class Dog : public Pet
{
    /* 생략 */
};

class Cat : public Pet
{
    /* 생략 */
};

class Hamster : public Pet
{
    /* 생략 */
};
```

파생 클래스들이 부모 클래스 코드를 공통으로 가지고 있는 것은 사실이지만 이것은 **공유의 개념이 아니라 다 개별적으로 부모 클래스를 복사**해서 가지고 있는 개념입니다. 예를 들어, Dog, Cat, Hamster 클래스는 Pet 클래스 코드를 공유하는 것이 아니라 세 클래스 모두 Pet 클래스 코드를 동일하게 복사해서 가지고 있다는 뜻입니다. 즉, Dog, Cat, Hamster 클래스를 사용해서 객체를 만든 경우 이 세 객체는 서로의 메모리를 공유하지 않고 자신에게 할당된 개별적인 메모리를 사용합니다.



댓글 등록순 최신순

관심글 댓글 알림

Zermi 3
상속으로 코드를 숨기니까 코드가 너무 깔끔해져요!
2022.09.11. 18:26 답글쓰기

김성엽 M 작성자
중복 코드가 줄어들수록 깔끔힐 맛이 나죠 ㅎ
2022.09.11. 19:11 답글쓰기

쥐치 3
감탄..

2024.01.03. 23:44 답글쓰기

김성엽 M 작성자
재미있지? 조용하길래 걱정하고 있었는데 공부하고 있었군 ㅎㅎ 다행이다!

2024.01.03. 23:50 답글쓰기

쥐치 3
김성엽 넵 ㅎㅎ 쭉 보는 중입니다 ! 파이팅~!

2024.01.03. 23:52 답글쓰기

김성엽 M 작성자

쥐치

2024.01.03. 23:53 답글쓰기

dh221009

댓글을 남겨보세요

등록