30. 가상함수

- 가상함수 virtual function 은 virtual 예약어를 앞에 붙여서 선언한 메서드임.
- 파생 형식에서 메서드를 재정의하면 과거의 정의가 완전히 무시(Override) 된다는 특징.

■ 기본문법

- 가상함수를 선언하는 형태는 다음과 같음

```
virtual 반환형식 메서드이름
```

- 가상함수 정의 예제

```
#include <iostream>
using namespace std;
class CTest
{
public:
    // 가상함수로 선언 및 정의했다.
    virtual void PrintData()
    {
        cout << "CTest: " << m_nData << endl;
    }

    void TestFunc() {
        cout << "*** TestFunc() ***" << endl;

        // 실 형식의 함수가 호출된다!
        PrintData();
        cout << "******************" << endl;
    }
protected:
    int m_nData = 10;
};
class CTestEx : public CTest
{
public:
    // 기본 클래스의 가상 함수 멤버를 재정의했다.
    // 따라서 기존 정의는 무시된다.
    virtual void PrintData()
    {
        cout << "CTestEx: " << m_nData*2 << endl;
    }
};

int main()
{
    CTestEx a;
    a.PrintData();
    CTest& b = a;
    // 참조형식에 상관없이 실 형식의 함수가 호출된다.
    b.PrintData();
    // 늘 마지막에 재정의 된 함수가 호출된다.
    a.TestFunc();
}
```

- 위에서 CTest::PrintData() 함수를 가상함수로 선언했는데, 이 함수는 내부 멤버 데이터인 m_nData의 값을 출력함.
- CTestEx 클래스에서 PrintData() 함수를 재정의함.
- a.PrintData(); 코드가 실행되면 CTestEx::PrintData() 가 호출되는 것은 일반메서드 재정의한 것과 크게 다르지 않음.
- CTest& b = a;에서 실형식은 CTestEx이며 참조형식은 기본 클래스인 CTest에 속한 참조자 b가 선언됨. 그리고 b.PrintData();에서 b를 통해 PrintData()를 호출함.
- 일반 메서드의 경우 실형식은 중요하지 않고 참조형식이 무엇인지에 따라 어떤 메서드가 호출되는지 결정이 남.
- 가상함수는 일반메서드와 달리 참조형식이 무엇이든 실 형식의 메서드를 호출함.
- 일반 메서드는 참조형식을 따르고, 가상함수는 실 형식을 따른다!

```
void TestFunc() {
    cout << "*** TestFunc() ***" << endl;

    // 실 형식의 함수가 호출된다!
    PrintData();
    cout << "******************" << endl;
}
```

- 위의 TestFunc() 함수는 기본클래스인 CTest 의 멤버임. 함수 내부의 PrintData()는 기본적으로 CTest 클래스의 PrintData() 함수를 말하는 것임.
- 문제는 PrintData() 함수가 '가상함수' 라는 사실임.
- TestFunc() 함수에서 호출하는 PrintData() 함수는 미래에 재정의된 함수일 수 있음.
- 만일 파생형식에서 PrintData() 가상함수를 재정의한다면 TestFunc() 함수에서 PrintData() 호출하는 코드로 '미래'의 함수를 호출하는 것임.
- 가상 함수는 호출하는 것이 아니라 호출되는 것이다!
- C++로 프로그램을 개발할 때 모든 것을 새롭게 다 만드는 것이 아니라 외부 라이브러리나 그 집합체라 할 수 있는 '프레임워크'(Framework)를 가져다 사용하는 경우가 많음.
- 이 경우 주의해야 할 것은 '가상 함수를 대하는 태도'임.
- 특정 가상함수를 재정의 했을 때 해당 함수가 적절한 시기에 '자동으로' 호출된다는 사실을 미리 알고 거기에 맞게 코드를 작성하기 어려움.
- 프레임워크를 사용할 때 혹은 배울 때 주의해야 할 것중 하나는 어떤 클래스가 어떤 가상함수를 가졌고 언제 호출되는지 정확히 알아야 함.
- 언젠가 제작자 입장에서 코드를 설계해야 하는 때에 특정 가상함수가 미래에 재정의되는 것을 막아야 할 수도 있음. 그럴 경우 아래처럼 자신이 재정의한 가상함 수 뒤에 final 예약어를 붙임.

```
virtual void PrintData() final;
```

- 이렇게 해두면 컴파일 오류가 발생하기 때문에 파생 클래스에서 해당 함수를 재정의 할 수 없음.

■ 소멸자 가상화

- CTest클래스가 CTestEx 클래스의 기본형식이라면 CTest에 대한 포인터나 참조자로 CTestEx 클래스 인스터스를 참조할 수 있음. 이 경우 참조 형식과 실 형식이 달라짐.

```
CTest *p = new CTestEx;
```

- 위 코드처럼 상위 클래스로 하위 파생 클래스를 참조할 때 상위 클래스 형식을 '추상 자료형'(Abstract Data Type)이라고 함.
- 추상 자료형을 이용해 동적 생성한 객체를 참조할 경우 심각한 메모리 누수 오류가 발생할 수 있음.
- 원인은 파생 형식의 소멸자가 호출되지 않아서임.

● 소멸자 가상화 예제

```
#include <iostream>
using namespace std;

// 제작자 코드
class CTest
{
public:
    CTest() { m_pszData = new char[32]; }
    ~CTest() {
        cout << "~CTest() " << endl;
        delete[] m_pszData;
    }
private:
    char* m_pszData;
};

class CTestEx : public CTest
{
public:
    CTestEx() { m_pnData = new int[10]; }
    ~CTestEx() {
        cout << "~CTestEx() " << endl;
        delete[] m_pnData;
    }
private:
    int* m_pnData;
};

// 사용자 코드
int main()
{
    CTest* pData = new CTestEx;

    // 참조형식에 맞는 소멸자가 호출된다.
    delete pData;
    return 0;
}
```

● 실행결과

```
~CTest()
계속하려면 아무 키나 누르십시오 . . .
```

● CTest* pData = new CTestEx;에서 동적 생성한 CTestEx 인스턴스를 기본형식인 CTest에 대한 포인터로 참조함. 문제 되지 않음.

● delete pData;에서 delete 연산을 실행할 경우 참조형식의 소멸자만 호출되고 실 형식의 소멸자가 호출되지 않는 심각한 내부적 문제가 발생함.

- 이 문제를 해결하는 가장 쉬운 방법은 '소멸자를 가상화'하는 것임.
- 다음 코드처럼 기본 클래스의 소멸자를 가상함수로 선언하면 됨.

```
// 소멸자를 가상함수로 선언
virtual ~CTest() {
    cout << "~CTest() " << endl;
    delete[] m_pszData;
}
```

- 실행결과

```
~CTestEx()
~CTest()
계속하려면 아무 키나 누르십시오 . . .
```

- 소멸자를 가상화하고 실행결과를 확인하면 파생클래스의 소멸자까지 제대로 호출된 것을 확인할 수 있음.