Chapter 6

Index

상속 : 객체 단위 코드를 재사용하는 방법, 재사용이란 기능적 확장이나 개선을 의미

재정의 : 기존의 선언 및 정의된 코드를 유지하면서도 새롭게 바꾸는 방법

메서드 재정의 : 어떤 클래스에 있는 메서드를 자유롭게 재정의해서 사용하는 방법

상속이란?

- 객체 단위의 코드를 '재사용'하는 방법 '재사용' : 기능적 확장이나 개선을 의미

" 두 클래스 사이의 '관계'를 고려해 프로그램을 작성 "

기본 문법

```
class 파생클래스이름 : 접근제어지시자 부모클래스이름
```

```
// 기본 클래스 혹은 부모 클래스
class CMyData
{
```

```
};
....
// 파생 클래스 혹은 자식 클래스
class CMyDataEx : public CMyData
{
};
```

- 파생 클래스의 인스턴스가 생성될 때 기본 클래스의 생성자도 호출
- 파생 클래스는 기본 클래스의 멤버에 접근 가능, private로 선언된 클래스 멤버는 불가능
- 사용자 코드에서는 파생 클래스의 인스턴스를 통해 기본 클래스 메서드를 호출 가능

▼ 상속 클래스 기본 코드

```
/* 상속 클래스 기본 */
#include "pch.h"
#include <iostream>
using namespace std;

// 제작자 - 초기 개발자
class CMyData
{
public:   // 누구나 접근 가능
    CMyData() { cout << "CMyData()" << endl; }
    int GetData() { return m_nData; }
    void SetData(int nParam) { m_nData = nParam; }

protected:   // 파생 클래스만 접근 가능
    void PrintData() { cout << "CMyData::PrintData()" << endl; }

private:   // 누구도 접근 불가능
    int m_nData = 0;
};

// 제작자 - 후기 개발자
class CMyDataEx : public CMyData
{
public:
    CMyDataEx() { cout << "CMyDataEx()" << endl; }
    void TestFunc()
    {
        // 기본 형식 멤버에 접근
        PrintData();
        SetData(5);
        cout << CMyData::GetData() << endl;
    }
};
```

```
// 사용자
int main()
{
    CMyDataEx data;

    // 기본 클래스(CMyData) 멤버에 접근
    data.SetData(10);
    cout << data.GetData() << endl;

    // 파생 클래스(CMyDataEx) 멤버에 접근
    data.TestFunc();

    return 0;
}
```

파생 클래스의 생성자는 먼저 호출되지만 실행은 나중

메서드 재정의

- 매서드를 재정의하면 기존의 것이 ‘무시’되기 때문

기본 문법 및 특징

▼ 메서드 재정의 기본 코드

```
/* 메서드 재정의 기본 */
#include "pch.h"
#include <iostream>
using namespace std;

// 초기 제작자
class CMyData
{
public:
    int GetData() { return m_nData; }

    void SetData(int nParam) { m_nData = nParam; }

private:
    int m_nData = 0;
};

// 후기 제작자
class CMyDataEx : public CMyData
{
public:
    // 파생 클래스에서 기본 클래스의 메서드를 재정의했다.
```

```
    void SetData(int nParam)
    {
        // 입력 데이터의 값을 보정하는 새로운 기능을 추가한다.
        if (nParam < 0)
            CMyData::SetData(0);

        if (nParam > 10)
            CMyData::SetData(10);
    }
};

// 사용자 코드
int main()
{
    // 구형에는 값을 보정하는 기능이 없다.
    CMyData a;
    a.SetData(-10);
    cout << a.GetData() << endl;

    // 신형에는 값을 보정하는 기능이 있다.
    CMyDataEx b;
    b.SetData(15);
    cout << b.GetData() << endl;

    return 0;
}
```

재정의한 이유가 기존 코드를 제거하기 위해서라기보다는 기존 메서드와 새 메서드를 한데 묶어 작동하게 하려는 의도

참조 형식과 실 형식

▼ 메서드 재정의로 원본 형식을 참조 코드

```
/* 메서드 재정의로 원본 형식을 참조 */
int main()
{
  CMyDataEx a;
  CMyData &rData = a;
  rData.SetData(15);
  cout << rData.GetData() << endl;

  return 0;
}
>>> 15
```

```
int main()
{
  CMyData *pData = new CMyDataEx;
  pData->SetData(5);
  delete pData;

  return 0;
}
```

- 참조 형식이 CMyData, CMyData::SetData( )가 호출
- delete 연산을 실행해도 CMyDataEx 클래스의 소멸자는 호출 안됨

상속에서의 생성자와 소멸자

- 생성자는 호출과 실행 순서가 역순이고 소멸자는 같다.

호출 순서

▼ 상속 관계의 생성자와 소멸자의 호출 순서 코드

```
/* 상속 관계의 생성자와 소멸자의 호출 순서 */
#include "pch.h"
#include <iostream>
using namespace std;

class CMyDataA
{
public:
    CMyDataA() {
        cout << "CMyDataA()" << endl;
    }

    ~CMyDataA() {
        cout << "~CMyDataA()" << endl;
    }
};

class CMyDataB : public CMyDataA
{
public:
    CMyDataB() {
        cout << "CMyDataB()" << endl;
    }

    ~CMyDataB() {
```

```
        cout << "~CMyDataB()" << endl;
    }
};

class CMyDataC : public CMyDataB
{
public:
    CMyDataC() {
        cout << "CMyDataC()" << endl;
    }

    ~CMyDataC() {
        cout << "~CMyDataC()" << endl;
    }
};

int main()
{
    cout << "*****Begin*****" << endl;
    CMyDataC data;
    cout << "*****End*****" << endl;

    return 0;
}
```

“생성자의 실행 순서는 호출 순서와 정반대”

▼ 상속 관계에서의 논리 오류 코드

```
/* 상속 관계에서의 논리 오류 */
#include "pch.h"
#include <iostream>
using namespace std;

class CMyDataA
{
public:
    CMyDataA() {
        cout << "CMyDataA()" << endl;
        m_pszData = new char[32];
    }

    ~CMyDataA() {
        cout << "~CMyDataA()" << endl;
        delete m_pszData;
    }

protected:
    char* m_pszData;
```

```
};

class CMyDataB : public CMyDataA
{
public:
    CMyDataB() {
        cout << "CMyDataB()" << endl;
    }

    ~CMyDataB() {
        cout << "~CMyDataB()" << endl;
    }
};

class CMyDataC : public CMyDataB
{
public:
    CMyDataC() {
        cout << "CMyDataC()" << endl;
    }

    ~CMyDataC() {
        cout << "~CMyDataC()" << endl;

        // 파생 클래스에서 부모 클래스 멤버 메모리를 해제했다.
        delete m_pszData;
    }
};

int main()
{
    cout << "*****Begin*****" << endl;
    CMyDataC data;
    cout << "*****End*****" << endl;

    return 0;
}
```

문제 : 파생 클래스에서 부모 클래스의 멤버 변수에 접근했고 부모 클래스가 관리하는 포인터를 파생 클래스가 손댄 덧

- 15행, 파생 클래스인 CMyDataC클래스의 소멸자에서 m_pszData가 가리키는 메모리를 이미 해제

> 파생 클래스는 부모 클래스의 멤버 변수에 직접 쓰기 연산하지 않는다.
>
> 파생 클래스 생성자에서 부모 클래스 멤버 변수를 초기화하지 않는다.

생성자와 소멸자는 객체 자신의 초기화 및 해제만 생각

생성자 선택

▼ 생성자 초기화 목록에서 생성자 선택 코드

```
/* 생성자 초기화 목록에서 생성자 선택 */
#include "pch.h"
#include <iostream>
using namespace std;

class CMyData
{
public:
    CMyData() { cout << "CMyData()" << endl; }
    CMyData(int nParam) { cout << "CMyData(int)" << endl; }
    CMyData(double dParam) { cout << "CMyData(double)" << endl; }
};

class CMyDataEx : public CMyData
{
public:
    CMyDataEx() { cout << "CMyDataEx()" << endl; }

    // 기본 클래스의 세 가지 생성자 중에서 int 변수를 갖는 생성자를 선택했다.
    CMyDataEx(int nParam) : CMyData(nParam)
    {
        cout << "CMyDataEx(int)" << endl;
    }

    // 기본 클래스의 디폴트 생성자를 선택했다.
    CMyDataEx(double dParam) : CMyData()
    {
        cout << "CMyDataEx(double)" << endl;
    }
};

int main()
{
    CMyDataEx a;
    cout << "**********" << endl;
    CMyDataEx b(5);
    cout << "**********" << endl;
    CMyDataEx c(3.3);

    return 0;
}
```

```
using CMyData::CMyData;   // 다중정의 코드를 간편하게 만들어줌
```

문제

1. 파생클래스에서 일반 메서드를 재정의했다, 파생 형식 인스턴스를 기본 형식에 대한 포인터로 포인팅하고 호출한다면, 기본 형식과 파생 형식 중 어느 클래스의 메소드가 호출?

   기본 형식

2. A는 B의, B는 C의 기본 클래스일 때 C 클래스의 인스턴스를 선언한다, 가장 먼저 ‘실행’되는 생성자는 어느 클래스의 생성자?

   A클래스