27. 상속이란?

- 상속(Inheritance)은 객체 단위 코드를 '재사용'하는 방법이며, '재사용'이란 기능적 확장이나 개선을 의미함
- 상속을 이해하는데 가장 중요한 것은 바로 '관계'임
- 제작이나 사용의 대상으로 생각해야 하는 것이 하나의 클래스에서 둘 이상으로 늘어나야 하며, 그 두 클래스 사이의 '관계'를 고려해 프로그램을 작성해야 함

■ 기본문법

- 새로운 클래스를 선언할 때 특정 클래스를 상속받으려면 다음과 같이 코드를 작성

```
class 파생클래스이름 : 접근제어지시자 부모클래스이름
```

- 실제 코드 구조는 다음과 같음

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

- CMyDataEx 클래스는 CMyData 클래스를 상속받아 생성된 파생 클래스가 됨.
- CMyData 클래스는 CMyDataEx 클래스의 기본 클래스 혹은 부모 클래스가 됨.
- 기본 클래스

```
#include <iostream>
using namespace std;
// 제작자 – 초기 개발자
class CMyData
{
public: // 누구나 접근 가능
    CMyData() { cout << " CMyData() " << endl; }
    ~CMyData() {}
    int GetData() { return m_nX; }
    void SetData(int nParam) { m_nX = nParam; }
protected: // 파생 클래스 접근 가능
    void PrintData() {
        cout << " m_nX : " << GetData()  << endl;
    }
private: // 누구도 접근 불가능
    int m_nX = 0;
};
//  사용자 코드
int main()
{
    CMyData data;
    data.SetData(10);
    cout << data.GetData() << endl;
    return 0;
}
```

● 파생 클래스

```
#include <iostream>
using namespace std;
// 제작자 – 후기 개발자
class CMyDataEx : public CMyData
{
public:
    CMyDataEx()
    {
        cout << "CMyDataEx() " << endl;
    }
    void SetDataEx(int nX, int nY)
    {
        CMyData::SetData(nX);
        m_nY = nY;
    }
    int GetData() { return m_nY; }
    void PrintDataEx()
    {
        CMyData::PrintData();
        cout << " m_nY : " << GetData() << endl;
    }
    void TestFunc()
    {
        PrintData();  // CMyData::PrintData(); 로 대체 가능
        SetData(10);  // CMyData::SetData(10); 로 대체 가능
        cout << CMyData::GetData() << endl;
        PrintDataEx();
    }
private:
    int m_nY = 0;
};
//  사용자 코드
int main()
{
    CMyDataEx test;
    test.SetDataEx(20, 30);

    test.TestFunc();

    return 0;
}
```

● 파생 클래스의 인스턴스가 생성될 때 기본 클래스의 생성자도 호출됨.

● 파생 클래스는 기본 클래스의 멤버에 접근할 수 있음. 단 private 접근 제어 지시자로 선언된 클래스 멤버에는 접근할 수 없음.

● 사용자 코드에서는 파생 클래스의 인스턴스를 통해 기본 클래스 메서드를 호출 할 수 있음.

● 상속 관계에서 파생 클래스의 생성자는 먼저 호출 되지만 실행은 나중에 됨.

28. 메서드 재정의

- 메서드 재정의에서 '재정의'는 오버라이드 override로 사전적 의미는 '무시하다'
- 메서드를 재정의하면 기존의 것이 무시되기 때문
- 파생 클래스에서 기본 클래스의 메서드를 재정의하면 기존의 것은 무시되고 새로 정의된 것이 기존 것을 대체함. 기본 클래스의 메서드를 아예 없애버리는 것이 아님!

```
#include <iostream>
using namespace std;
// 제작자 – 초기 개발자
class CMyData
{
public: // 누구나 접근 가능
    CMyData() { cout << " CMyData() " << endl; }
    ~CMyData() {}
    int GetData() { return m_nX; }
    void SetData(int nParam) { m_nX = nParam; }
protected: // 파생 클래스 접근 가능
    void PrintData() {
        cout << " m_nX : " << GetData()  << endl;
    }
private: // 누구도 접근 불가능
    int m_nX = 0;
};
// 후기 개발자
class CMyDataEx : public CMyData
{
public:
    CMyDataEx() { cout << "CMyDataEx()" << endl; }
    // 파생 클래스에서 기본 클래스의 메서드를 재정의했다.
    void SetData(int nParam)
    {
        // 입력 데이터의 값을 보정하는 새로운 기능을 추가한다.
        if (nParam < 0) {
            CMyData::SetData(0);
        }
        else if (nParam > 10) {
            CMyData::SetData(10);
        }
        else {
            CMyData::SetData(nParam);
        }
    }
};
```

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

    CMyDataEx data2;
    data2.SetData(15);
    cout << data2.GetData() << endl;
    return 0;
}
```

- CMyString클래스의 SetData() 함수는 멤버변수인 m_nX에 값을 대입하는 기능만 제공하고 값의 범위를 검사하는 등의 부가적 기능은 제공하지 않음
- 시간이 지나고 멤버변수의 값을 0~10으로 제한해야 할 필요성이 생겨서 후기 제작자는 기존의 코드와 자신의 코드를 적절히 섞어 확장시키는 시도를 했고 재정의로 완성됨.

```
        if (nParam < 0) {
            CMyData::SetData(0);
        }
        else if (nParam > 10) {
            CMyData::SetData(10);
        }
        else {
            CMyData::SetData(nParam);
```

- 위의 소스를 SetData(0); SetData(10);SetData(nParam); 으로 바꿔버리면 '재귀호출'이 발생함. 즉 CMyDataEx::SetData()를 호출하게 됨.
- 파생형식에서 기본형식의 동일한 메서드를 호출하려면 반드시 소속클래스를 명시해야 함.
- CMyDataEx::SetData() 내부에서 다시 CMyData::SetData()를 호출한다는 것은 '두 코드를 섞는다'는 의미로 재정의한 이유가 기존 코드를 제거하기 위해서라기보다는 기본 메서드와 새 메서드를 한데 묶어 작동하게 하려는 의도로 이해할 수 있음.

```
        CMyDataEx b;
        b.SetData(15);
```

- 위에서 실행되는 SetData() 함수는 CMyDataEx::SetData() 함수입니다. SetData() 함수를 사용하는 방법이 기존 CMyData 클래스와 다르지 않지만 새로운 기능이 적용됨.
- 다음과 같이 호출하면 CMyData클래스의 SetData()함수를 호출할 수 있음(명시적 호출)

```
        CMyDataEx b;
        b.CMyData::SetData(15);
```

- 소속을 지정한 경우로 명시적 호출이라 함.

■ 참조 형식과 실 형식

- 앞에서 본 사용자 코드를 다음과 같이 변경하면 어떤 결과가 나올지 생각해봅시다.

```
//  사용자 코드
int main()
{
    CMyDataEx a;
    CMyData &rData = a;
    rData.SetData(15);

    cout << rData.GetData() << endl;
    return 0;
}
```

- CMyData & 형식인 rData를 선언 및 정의했는데 원본이 CMyData 클래스가 아니라 CMyDataEx 클래스임.
- CMyDataEx는 CMyData 클래스의 파생클래스 형식임.
- 파생형식을 기본형식으로 참조하는 것은 매우 자연스러운 일임.
- rData.SetData(15);처럼 실 형식과 참조 형식이 서로 다른 경우 묵시적인 호출로 어떤 메서드가 호출되는지 생각해봐야함.('참조형식'임)
- 참조형식이 참조자만 있는 것이 아니라 포인터도 있음. 다음과 같은 코드도 사용할 수 있음.

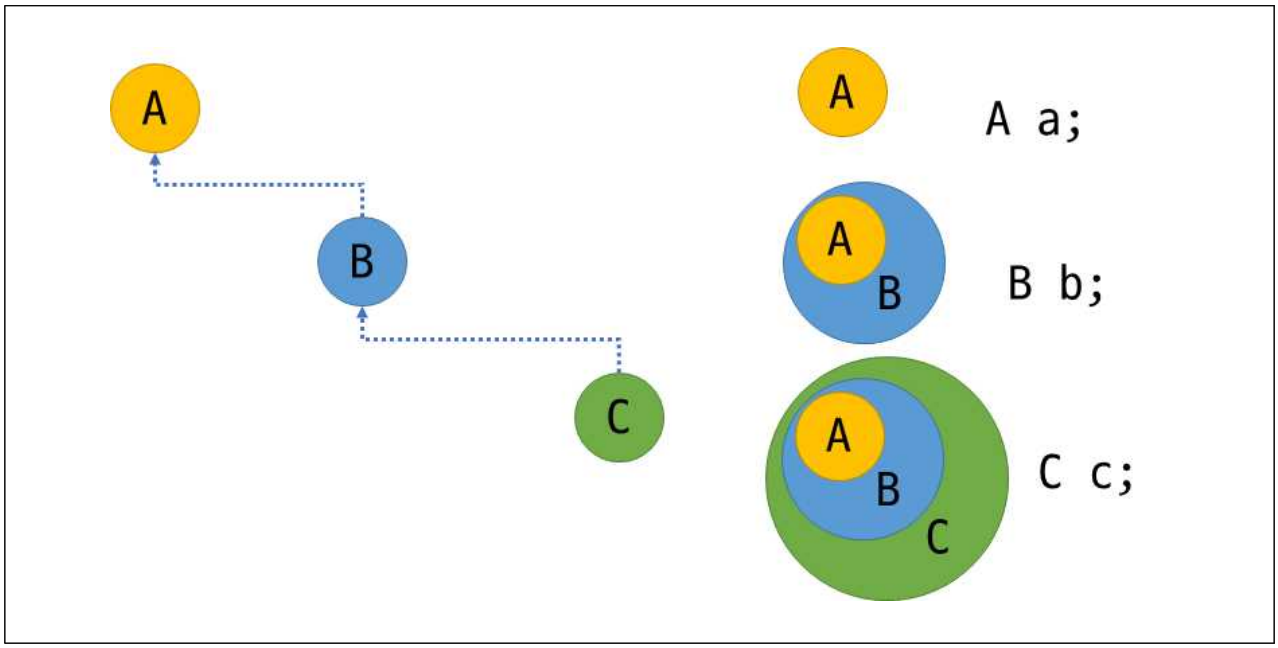
```
//  사용자 코드
int main()
{
    CMyData *pData = new CMyDataEx;
    pData->SetData(5);
    delete pData;

    return 0;
}
```

- pData->SetData(5);처럼 SetData() 메서드를 묵시적 호출했는데 참조형식이 CMyData입니다. 따라서 동적 생성된 실형식이 CMyDataEx임에도 CMyData::SetData()가 호출됨.

29. 상속에서의 생성자와 소멸자

- 상속을 통해 클래스는 점점 덩치가 커짐.
- A가 B의 기본 클래스고 B가 C의 기본 클래스인 상속관계일 때 클래스의 크기를 그림으로 표현하면 다음과 같음. 클래스의 덩치가 점점 커짐.



- A는 B의 기본이고 B는 C의 기본 클래스인 상속관계에서 생성자와 소멸자의 관계는 다음과 같음.

✓ C 클래스 인스턴스를 선언하면 생성자 호출순서는 C, B, A다.
✓ 하지만 가장 먼저 실행되는 생성자는 C가 아니라 A다.
✓ C 클래스의 인스턴스가 소멸하면 C클래스의 소멸자가 가장 먼저 호출되고 실행된다.
✓ 정리하면 “생성자는 호출과 실행순서가 역순이고 소멸자는 같다”

■ 호출 순서

- 아래 예제는 상속관계인 CMyDataA, CMyDataB, CMyDataC 세 클래스의 생성자와 소멸자 호출순서를 확인할 수 있는 예제임.
- CMyDataA는 기본 클래스가 없는 최상위 클래스고 CMyDataC는 마지막 파생 클래스임.

```
#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;
}
```

- 실행결과

```
***** Begin *****
 CMyDataA()
 CMyDataB()
 CMyDataC()
*****  End  *****
 ~CMyDataC()
 ~CMyDataB()
 ~CMyDataA()
계속하려면 아무 키나 누르십시오 . . .
```

- 생성자는 CMyDataA, CMyDataB, CMyDataC 순서로 실행되지만 생성자의 실행순서는 호출순서와 정반대임.
- 파생 클래스의 생성자가 먼저 호출되고 이어서 상위로 올라가듯 호출된다는 것임.
- 파생 클래스 생성자가 기본 클래스의 생성자를 '선택' 하고 호출한다는 것
- 생성자가 여러 형태로 다중 정의 된 경우라도 파생 클래스에서 상위 클래스 생성자는 하나만 '선택'할 수 있음. 별도로 생성자를 선택하지 않으면 디폴트 생성자가 선택됨.

```
#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;
}
```

- 이 코드의 가장 치명적인 문제는 파생 클래스에서 부모 클래스의 멤버 변수에 접근했고 부모 클래스가 관리하는 포인터를 파생클래스에서 손댄 것임.
- 파생 클래스인 CMyDataC 클래스의 소멸자에서 m_pszData가 가리키는 메모리를 이미 해제함. 소멸자는 CMyDataC가 CMyDataA보다 먼저 호출되면서 먼저 실행됨.
- 파생클래스는 부모 클래스의 멤버변수에 직접쓰기 연산하지 않는 것이 정답임. 파생 클래스 생성자에서 부모 클래스 멤버변수를 초기화 하지 않음.
- 생성자와 소멸자를 통해 상위 클래스의 멤버변수에 절대로 임의 접근하지 않기(특히 포인터!!)
- 생성자와 소멸자는 객체 자신의 초기화 및 해제만 생각하는 것이 맞음.

■ 생성자 선택

- 파생 클래스의 생성자는 자신이 호출된 후 기본 클래스의 생성자 중 어떤 것이 호출될 것인지 '선택'할 수 있음.
- '선택'하는 방법은 '생성자 초기화 목록'에 상위 클래스 생성자를 호출하듯이 기술함.

```
#include <iostream>
using namespace std;
class CTest
{
public:
    CTest() { cout << "CTest()" << endl; }
    CTest(int nParam) { cout << "CTest(int nParam)" << endl; }
    CTest(double dParam) { cout << "CTest(double dParam)" << endl; }
};

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

};

int main()
{
    CTestEx test1;
    cout << "**************" << endl;
    CTestEx test2(10);
    cout << "**************" << endl;
    CTestEx test(1.1);
    return 0;
}
```

- 실행결과

```
CTest()
CTestEx()
**************
CTest(int nParam)
CTestEx(int)
**************
CTest(double dParam)
CTestEx(double)
계속하려면 아무 키나 누르십시오 . . .
```

- 파생클래스인 CTestEx 클래스의 생성자는 세 가지로 다중 정의 되어있음.

- ✓ CTestEx();
- ✓ CTestEx(int nParam);
- ✓ CTestEx(double dParam);

- 셋 중에서 CTestEx(int)는 기본 클래스의 CTest(int)를, CTestEx(double)은 CTest(double)를 초기화 목록에서 선택함.

- C++11 표준에서 '생성자 상속'이 등장
- 파생 클래스를 만들 때 다중 정의된 상위 클래스의 생성자들을 그대로 가져오는 문법임.
- 다음 코드는 파생 클래스에서 기본 클래스의 다중 정의된 생성자들을 그대로 사용하고 있고 별다은 코드를 실행하지 않음.

```
class CTestEx : public CTest
{
public:
    CTestEx() {  }
    CTestEx(int nParam) :CTest(nParam) { }
    CTestEx(double dParam) :CTest(dParam) { }

};
```

- 위 코드는 아래 using CTest::CTest;라고 단 한줄로 정리할 수 있음.

```
class CTestEx : public CTest
{
public:
    using CTest::CTest;
};
```

- 실행결과

```
CTest()
**************
CTest(int nParam)
**************
CTest(double dParam)
계속하려면 아무 키나 누르십시오 . . .
```

- 실행결과를 보면 기본 클래스의 생성자들이 모두 호출된 것을 알수 있음.
- using CTest::CTest;라고 써주는 것으로 다음과 같은 코드가 자동으로 만들어 지는 셈임.

```
    CTestEx(int nParam) :CTest(nParam) { }
    CTestEx(double dParam) :CTest(dParam) { }
```