7

Chapter 7

Index

가상 함수 : virtual 예약어를 앞에 붙여서 선언한 메서드, 부모 형식과 관계 없이 파생(자식)형식에서 메서드 다시 재정의 가능

가상 클래스 : 가상 함수를 가진 클래스, 가상 클래스의 소멸자는 virtual 예약어 선언이 없더라도 자동으로 가상화

다중 상속 : 한 클래스가 두 개 이상의 클래스를 동시에 상속받는 경우, 좋은 새로운 개체 정의 가능

가상 함수

- virtual 예약어를 앞에 붙여서 선언한 메서드

기본 문법

```
virtual 반환형식 메서드이름
```

▼ 가상 함수 정의 코드

```
/* 가상 함수 정의 */
#include "pch.h"
#include <iostream>
using namespace std;

class CMyData
{
public:
    // 가상 함수로 선언 및 정의했다.
    virtual void PrintData()
    {
        cout << "CMyData: " << endl;
    }

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

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

protected:
    int m_nData = 10;
};

class CMyDataEx : public CMyData
{
public:
    // 기본 클래스의 가상 함수 멤버를 재정의했다.
    // 따라서 기존 정의는 무시된다.
    virtual void PrintData()
    {
        cout << "CMyDataEx: " << m_nData * 2 << endl;
    }
};

int main()
{
    CMyDataEx a;
    a.PrintData();
```

```
    CMyData& b = a;

    // 참조 형식에 상관없이 실 형식의 함수가 호출된다.
    b.PrintData();

    // 늘 마지막에 재정의된 함수가 호출된다!
    a.TestFunc();

    return 0;
}
```

일반 메서드: 실 형식은 중요하지 않고 참조 형식이 무엇인지에 따라 어떤 메서드가 호출되는지 결정

가상 함수: 참조 형식이 무엇이든 실 형식의 메서드를 호출

> **일반 메서드는 참조 형식을 따르고, 가상 함수는 실 형식을 따른다!**

파생 형식에서 PrintData( ) 가상 함수를 재정의한다면 현재의 19번 행 코드로 '미래'의 함수를 호출

" 가상 함수는 호출하는 것이 아니라 호출되는 것이다! "

final : 파생 클래스에서 해당 함수를 재정의 불가능

소멸자 가상화

▼ 소멸자 가상화 코드

```
/* 소멸자 가상화 */
#include "pch.h"
#include <iostream>
using namespace std;

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

```
private:
    char* m_pszData;
};

class CMyDataEx : public CMyData
{
public:
    CMyDataEx() { m_pnData = new int; }
    ~CMyDataEx()
    {
        cout << "~CMyDataEx()" << endl;
        delete m_pnData;
    }

private:
    int* m_pnData;
};

// 사용자 코드
int main()
{
    CMyData* pData = new CMyDataEx;

    // 참조 형식에 맞는 소멸자가 호출된다.
    delete pData;

    return 0;
}
```

>> delete 연산을 실행할 경우 참조 형식의 소멸자만 호출, 실 형식의 소멸자는 호출 안됨

해결방법 - 소멸자를 가상화

```
// 소멸자를 가상 함수로 선언
virtual ~CMyData()
{
  cout << "~CMyData()" << endl;
  delete m_pszData;
}
```

가상 함수 테이블(vtable)

- 함수 포인터 배열

▼ vtable 구현 코드

```
/* vtable 구현 */
#include "pch.h"
#include <iostream>
using namespace std;

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

    virtual ~CMyData() { }
    virtual void TestFunc1() { }
    virtual void TestFunc2() { }
};

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

    virtual ~CMyDataEx() { }
    virtual void TestFunc1() { }
    virtual void TestFunc2()
    {
        cout << "TestFunc2()" << endl;
    }
};

int main()
{
    CMyData* pData = new CMyDataEx;
    pData->TestFunc2();
    delete pData;

    return 0;
}
```

_ _vfptr : vtable 포인터

바인딩 : 함수나 변수의 주소가 결정되는 것 (가상 함수 = 늦은 바인딩)

▼ 이른 바인딩 코드

```
/* 이른 바인딩 */
#include "pch.h"
#include <iostream>
using namespace std;

void TestFunc(int nParam) { }

int main()
{
    TestFunc(10);

    return 0;
}
```

▼ **늦은 바인딩 코드**

```
/* 늦은 바인딩 */
#include "pch.h"
#include <iostream>
using namespace std;

void TestFunc1(int nParam) { }
void TestFunc2(int nParam) { }

int main()
{
    int nInput = 0;
    cin >> nInput;
    void(*pfTest)(int) = NULL;

    if (nInput > 10)
        pfTest = TestFunc1;

    else
        pfTest = TestFunc2;

    pfTest(10);

    return 0;
}
```

순수 가상 클래스

- 순수 가상 함수를 멤버로 가진 클래스

순수 가상 함수 : 선언은 지금 해두지만 정의는 미래에 하도록 미뤄둔 함수

```
virtual int GetData() const = 0;
```

> 순수 가상 클래스의 파생 클래스는 반드시 기본 클래스의 순수 가상 함수를 재정의해야 한다.

인터페이스 상속

- 인터페이스 : 서로 다른 두 객체가 서로 맞닿아 상호작용할 수 있는 통로나 방법

▼ 인터페이스 코드

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

// 초기 제작자의 코드
class CMyObject
{
public:
    CMyObject() { }
    virtual ~CMyObject() { }

    // 모든 파생 클래스는 이 메서드를 가졌다고 가정할 수 있다.
    virtual int GetDeviceID() = 0;

protected:
    int m_nDeviceID;
};

// 초기 제작자가 만든 함수
void PrintID(CMyObject* pObj)
{
    // 실제로 어떤 것일지는 모르지만 그래도 ID는 출력할 수 있다!
    cout << "Device ID: " << pObj->GetDeviceID() << endl;
}

// 후기 제작자 코드
class CMyTV : public CMyObject
{
public:
    CMyTV(int nID) { m_nDeviceID = nID; }
    virtual int GetDeviceID()
    {
        cout << "CMyTV::GetDeviceID()" << endl;
        return m_nDeviceID;
```

```
    }
};

class CMyPhone : public CMyObject
{
public:
    CMyPhone(int nID) { m_nDeviceID = nID; }
    virtual int GetDeviceID()
    {
        cout << "CMyPhone::GetDeviceID()" << endl;
        return m_nDeviceID;
    }
};

// 사용자 코드
int main()
{
    CMyTV a(5);
    CMyPhone b(10);

    // 실제 객체가 무엇이든 알아서 자신의 ID를 출력한다.
    ::PrintID(&a);
    ::PrintID(&b);

    return 0;
}
```

가상 함수는 추상 자료형으로 참조하더라고 언제나 실 형식의 메서드가 호출

추상 자료형의 사용 예

▼ 추상 자료형의 효율성 코드

```
/* 추상 자료형의 효율성 */
#include "pch.h"
#include <iostream>
using namespace std;

// 초기 제작자
#define DEFAULT_FARE 1000

class CPerson
{
public:
    CPerson() { }
    virtual ~CPerson() {
        cout << "virtual ~CPerson()" << endl;
    }
```

```
    // 요금 계산 인터페이스(순수 가상 함수)
    virtual void CalcFare() = 0;

    virtual unsigned int GetFare() { return m_nFare; }

protected:
    unsigned int m_nFare = 0;
};

// 초기 혹은 후기 제작자
class CBaby : public CPerson
{
public:
    // 영유아(0~7세) 요금 계산
    virtual void CalcFare() {
        m_nFare = 0;  // 0%
    }
};

class CChild : public CPerson
{
public:
    // 어린이(8~13세) 요금 계산
    virtual void CalcFare() {
        m_nFare = DEFAULT_FARE * 50 / 100;  // 50%
    }
};

class CTeen : public CPerson
{
public:
    // 청소년(14~19세) 요금 계산
    virtual void CalcFare() {
        m_nFare = DEFAULT_FARE * 75 / 100;  // 75%
    }
};

class CAdult : public CPerson
{
public:
    // 성인(20세 이상) 요금 계산
    virtual void CalcFare() {
        m_nFare = DEFAULT_FARE;              // 100%
    }
};

// 사용자 코드
int main()
{
    CPerson* arList[3] = { 0 };
    int nAge = 0;
```

```
    // 1. 자료 입력: 사용자 입력에 따라서 생성할 객체 선택
    for (auto& person : arList)
    {
        cout << "나이를 입력하세요: ";
        cin >> nAge;
        if (nAge < 8)
            person = new CBaby;

        else if (nAge < 14)
            person = new CChild;

        else if (nAge < 20)
            person = new CTeen;

        else
            person = new CAdult;

        // 생성한 객체에 맞는 요금이 자동으로 계산된다.
        person->CalcFare();
    }

    // 2. 자료 출력: 계산한 요금을 활용하는 부분
    for (auto person : arList)
        cout << person->GetFare() << "원" << endl;

    // 3. 자료 삭제 및 종료
    for (auto person : arList)
        delete person;

    return 0;
}
```

상속과 형변환

형변환 연산자	설명
const_cast<>	상수형 포인터에서 const를 제거합니다.
static_cast<>	컴파일 시 상향 혹은 하향 형변환합니다.
dynamic_cast<>	런타임 시 상향 혹은 하향 형변환합니다.
reinterpret_cast<>	C의 형변환 연산자와 흡사합니다.

static_cast

▼ static_cast 사용 예 코드

```
/* static_cast 사용 예 */
#include "pch.h"
#include <iostream>
using namespace std;

class CMyData
{
public:
    CMyData() { }
    virtual ~CMyData() { }
    void SetData(int nParam) { m_nData = nParam; }
    int GetData() { return m_nData; }

private:
    int m_nData = 0;
};

class CMyDataEx : public CMyData
{
public:
    void SetData(int nParam)
    {
        if (nParam > 10)
            nParam = 10;

        CMyData::SetData(nParam);
    }

    void PrintData()
    {
        cout << "PrintData(): " << GetData() << endl;
    }
};

int main()
{
    // 파생 형식의 객체를 기본 형식으로 포인팅합니다.
    CMyData* pData = new CMyDataEx;
    CMyDataEx* pNewData = NULL;

    // CMyData::SetData() 함수를 호출합니다.
    // 따라서 10이 넘는지 검사하지 않습니다.
    pData->SetData(15);

    // 기본 형식에 대한 포인터나 가리키는 대상은 파생 형식입니다.
    // 이 사실이 명확하므로 파생 형식에 대한 포인터로 형변환을 시도합니다.
    pNewData = static_cast<CMyDataEx*>(pData);
    pNewData->PrintData();
    delete pData;
```

```
    return 0;
}
```

잘못된 형변환 조심 - (자료형)

C++ 전용 형변환 연산자 적극 사용 권장 (C x)

dynamic_cast

- 좋지 못한 방향으로 흘러갈 때 사용 - 사용하지 않는 것을 권장

▼ dynamic_cast 사용 예 코드

```
/* dynamic_cast 사용 예 */
#include "pch.h"
#include <iostream>
using namespace std;

class CShape
{
public:
    CShape() { }
    virtual ~CShape() { }
    virtual void Draw() { cout << "CShape::Draw()" << endl; }
};

class CRectangle : public CShape
{
public:
    virtual void Draw() { cout << "CRectangle::Draw()" << endl; }
};

class CCircle : public CShape
{
public:
    virtual void Draw() { cout << "CCircle::Draw()" << endl; }
};

int main()
{
    cout << "도형 변호를 입력하세요. [1:사각형, 2:원]" << endl;
    int nInput = 0;
    cin >> nInput;

    CShape* pShape = nullptr;
    if (nInput == 1)
        pShape = new CRectangle;

    else if (nInput == 2)
```

```
        pShape = new CCircle;

    else
        pShape = new CShape;

    // 좋은 예
    pShape->Draw();

    // '매우' 나쁜 예
    // 가상 함수를 활용한다면 이런 코드를 작성할 이유가 없다!
    CRectangle* pRect = dynamic_cast<CRectangle*>(pShape);
    if (pRect != NULL)
        cout << "CRectangle::Draw()" << endl;

    else
    {
        CCircle* pCircle = dynamic_cast<CCircle*>(pShape);
        if (pCircle != NULL)
            cout << "CCircle::Draw()" << endl;
        else
            cout << "CShape::Draw()" << endl;
    }

    return 0;
}
```

> reinterpret_cast : 차라리 C스타일을 쓰는게 나음

상속과 연산자 다중 정의

▼ 파생 클래스의 연산자 다중 정의 코드

```
/* 파생 클래스의 연산자 다중 정의 */
#include "pch.h"
#include <iostream>
using namespace std;

class CMyData
{
public:
    CMyData(int nParam) :m_nData(nParam) { }

    CMyData operator+(const CMyData& rhs)
    {
        return CMyData(m_nData + rhs.m_nData);
    }
```

```
    CMyData operator=(const CMyData& rhs)
    {
        m_nData = rhs.m_nData;

        return *this;
    }

    operator int() { return m_nData; }

protected:
    int m_nData = 0;
};

class CMyDataEx : public CMyData
{
public:
    CMyDataEx(int nParam) : CMyData(nParam) { }
};

int main()
{
    CMyData a(3), b(4);
    cout << a + b << endl;

    CMyDataEx c(3), d(4), e(0);

    // CMyDataEx 클래스에 맞는 단순 대입 연산자가 없어서 컴파일 오류가 발생한다.
    e = c + d;
    cout << e << endl;

    return 0;
}
```

>>> Error

▼ Fix 코드

```
class CMyDataEx : public CMyData
{
public:
    CMyDataEx(int nParam) : CMyData(nParam) { }
    CMyDataEx operator+(const CMyDataEx& rhs)
    {
        return CMyDataEx(static_cast<int>(CMyData::operator+(rhs)));
    }
};
```

```
class CMyDataEx : public CMyData
{
public:
    CMyDataEx(int nParam) : CMyData(nParam) { }

    // 인터페이스를 맞춰주기 위한 연산자 다중 정의
    using CMyData::operator+;
    using CMyData::operator=;
};
```

다중 상속

- 한 클래스가 두 개 이상의 클래스를 동시에 상속받는 경우
- 별로 사용하지 않는 것이 좋음

다중 상속과 모호성

```
class CMyPicture : public CMyImage, public CMyShape
```

▼ 다중 상속 코드

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

class CMyImage
{
public:
    CMyImage(int nHeight, int nWidth)
        : m_nHeight(nHeight), m_nWidth(nWidth)
    {
        cout << "CMyImage(int, int)" << endl;
    }

    int GetHeight() const { return m_nHeight; }
    int GetWidth() const { return m_nWidth; }

protected:
    int m_nHeight;
    int m_nWidth;
};
```

```
class CMyShape
{
public:
    CMyShape(int nType) : m_nType(nType)
    {
        cout << "CMyShape(int)" << endl;
    }

    int GetType() const { return m_nType; }

protected:
    int m_nType;
};

// 두 클래스를 모두 상속받는다.
class CMyPicture : public CMyImage, public CMyShape
{
public:
    CMyPicture() : CMyImage(200, 120), CMyShape(1)
    {
        cout << "CMyPicture()" << endl;
    }
};

int main()
{
    CMyPicture a;
    cout << "Width: " << a.GetWidth() << endl;
    cout << "Height: " << a.GetHeight() << endl;
    cout << "Type: " << a.GetType() << endl;

    return 0;
}
```

▼ 원형이 완전히 일치하는 메서드 호출 코드

```
/* 원형이 완전히 일치하는 메서드 호출 */
#include "pch.h"
#include <iostream>
using namespace std;

class CMyImage
{
public:
    CMyImage(int nHeight, int nWidth)
        : m_nHeight(nHeight), m_nWidth(nWidth)
    {
        cout << "CMyImage(int, int)" << endl;
    }
```

```
    int GetHeight() const { return m_nHeight; }
    int GetWidth() const { return m_nWidth; }
    int GetSize() const { return 0; }

protected:
    int m_nHeight;
    int m_nWidth;
};

class CMyShape
{
public:
    CMyShape(int nType) : m_nType(nType)
    {
        cout << "CMyShape(int)" << endl;
    }

    int GetType() const { return m_nType; }
    int GetSize() const { return 0; }

protected:
    int m_nType;
};

// 두 클래스를 모두 상속받는다.
class CMyPicture : public CMyImage, public CMyShape
{
public:
    CMyPicture() : CMyImage(200, 120), CMyShape(1)
    {
        cout << "CMyPicture()" << endl;
    }
};

int main()
{
    CMyPicture a;
    cout << "Width: " << a.GetWidth() << endl;
    cout << "Height: " << a.GetHeight() << endl;
    cout << "Type: " << a.GetType() << endl;

    // GetSize() 메서드가 두 부모 클래스에 모두 존재한다.
    a.GetSize();

    return 0;
}
```

>> Error (모호함)

묵시적인 호출이 아니라 명시적인 호출로 바꿔준다.

가상 상속

▼ 가상 상속 전 코드

```
/* 가상 상속 적용 전 */
#include "pch.h"
#include <iostream>
using namespace std;

class CMyObject
{
public:
    CMyObject() { cout << "CMyObject()" << endl; }
    virtual ~CMyObject() { }
};

class CMyImage : public CMyObject
{
public:
    CMyImage() { cout << "CMyImage(int, int)" << endl; }
};

class CMyShape : public CMyObject
{
public:
    CMyShape() { cout << "CMyShape(int)" << endl; }
};

class CMyPicture : public CMyImage, public CMyShape
{
public:
    CMyPicture() { cout << "CMyPicture()" << endl; }
};

int main()
{
    CMyPicture a;
}
```

>> 최상위 기본 클래스가 두 번 호출된다.

▼ 가상 상속 후 코드

```
/* 가상 상속 적용 후 */
#include "pch.h"
```

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

class CMyObject
{
public:
    CMyObject() { cout << "CMyObject()" << endl; }
    virtual ~CMyObject() { }
};

class CMyImage : virtual public CMyObject
{
public:
    CMyImage() { cout << "CMyImage(int, int)" << endl; }
};

class CMyShape : virtual public CMyObject
{
public:
    CMyShape() { cout << "CMyShape(int)" << endl; }
};

class CMyPicture : public CMyImage, public CMyShape
{
public:
    CMyPicture() { cout << "CMyPicture()" << endl; }
};

int main()
{
    CMyPicture a;
}
```

(파생 형식 클래스에 virtual 예약어를 함께 선언)

인터페이스 다중 상속

▼ 인터페이스 다중 상속 코드

```
/* 인터페이스 다중 상속 */
#include "pch.h"

class CMyUSB
{
public:
    virtual int GetUsbVersion() = 0;
    virtual int GetTransferRate() = 0;
};
```

```
class CMySerial
{
public:
    virtual int GetSignal() = 0;
    virtual int GetRate() = 0;
};

class CMyDevice : public CMyUSB, public CMySerial
{
public:
    // USB 인터페이스
    virtual int GetUsbVersion() { return 0; }
    virtual int GetTransferRate() { return 0; }

    // 시리얼 인터페이스
    virtual int GetSignal() { return 0; }
    virtual int GetRate() { return 0; }
};

int main()
{
    CMyDevice dev;

    return 0;
}
```

CMyDevice 클래스의 인스턴스는 두 클래스가 가진 인터페이스를 모두 제공

그나마 다중 상속을 사용해도 좋은 유일한 상황

문제

1. **가상 함수를 사용하는 가장 큰 이유?**

   파생 클래스에서도 함수를 재정의할 수 있게 하기 위해서

2. **소멸자를 반드시 가상화해야 하는 경우?**

   그래야 참조 소멸자 호출 외에 실 소멸자도 호출 된다.

3. **늦은 바인딩이란?**

   함수나 변수의 주소가 처음 것이 아니라 미래의 것인 걸로 결정되는 것

4. **순수 가상 클래스의 파생 클래스에서 반드시 해야 하는 일?**

   기본 클래스의 순수 가상 함수를 재정의 해야 한다.

5. **다중 상속의 모호성을 회피하는 방법?**

묵시적으로 호출하지 말고 명시적으로
지정해준다.