4

Chapter 4

복사 생성자 : 객체의 복사본을 생성할 때 호출되는 생성자

깊은 복사와 얕은 복사 : 실제 값을 두 개로 만드는 깊은 복사와, 값은 하나이나 포인터만 두 개를 생성하는 얕은 복사

임시 객체 : 컴파일러가 임의로 생성했다가 바로 소멸시키는 객체, 성능 향상을 위해 이를 다루려면 '식별자'를 부여해야 한다.

이동 시맨틱 : 복사 생성자와 대입 연산자에 r-value 참조를 조합해서 새로운 생성 및 대입의 경우를 만든 것

복사 생성자

- 객체의 복사본을 생성할 때 호출하는 생성자

▼ **복사 생성자 선언 및 정의 코드**

```
/* 복사 생성자 선언 및 정의 */
#include "pch.h"
#include <iostream>
using namespace std;

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

    // 복사 생성자 선언 및 정의
    CMyData(const CMyData &rhs)
        // : m_nData(rhs.m_nData)
    {
        this->m_nData = rhs.m_nData;
        cout << "CMyData(const CMyData &)" << endl;
    }

    int GetData(void) const { return m_nData; }
    void SetData(int nParam) { m_nData = nParam; }
```

```
private:
    int m_nData = 0;
};

int main()
{
    // 디폴트 생성자가 호출되는 경우
    CMyData a;
    a.SetData(10);

    // 복사 생성자가 호출되는 경우
    CMyData b(a);
    cout << b.GetData() << endl;

    return 0;
}
```

함수 형태로 호출할 때는 클래스가 매개변수로 사용되는 경우와 반환 형식으로 사용되는 경우로 나누어 진다.

반환 형식으로 사용되는 경우 '이름 없는 임시 객체'를 조심해야 한다.

▼ 매개변수로 사용되는 복사 생성자 코드

```
/* 매개변수로 사용된느 복사 생성자 */
#include "pch.h"
#include <iostream>
using namespace std;

class CTestData
{
public:
    CTestData(int nParam) : m_nData(nParam)
    {
        cout << "CTestData(int)" << endl;
    }

    CTestData(const CTestData& rhs) : m_nData(rhs.m_nData)
    {
        cout << "CTestData(const CTestData &)" << endl;
    }

    // 읽기 전용인 상수형 메서드
    int GetData() const { return m_nData; }

    // 멤버 변수에 쓰기를 시도하는 메서드
    void SetData(int nParam) { m_nData = nParam; }

private:
```

```
    int m_nData = 0;
};

// 매개변수가 CTestData 클래스 형식이므로 복사 생성자가 호출된다.
void TestFunc(CTestData param)
{
    cout << "TestFunc()" << endl;

    // 피호출자 함수에서 매개변수 인스턴스의 값을 변경한다.
    param.SetData(20);
}

int main()
{
    cout << "*****Begin*****" << endl;
    CTestData a(10);
    TestFunc(a);

    // 함수 호출 후 a의 값을 출력한다.
    cout << "a: " << a.GetData() << endl;

    cout << "*****End*****" << endl;

    return 0;
}
```

한 객체로 할 수 있는 일은 반드시 하나로 끝내야 한다.

방법:

- 복사 생성자를 삭제 - 하지만 오류가 뜸
- **참조자** 이용 - void TestFunc(CTestData param) → void TestFunc(CTestData &param)
    - 단점 : 사용자의 코드만 봐서는 '참조에 의한 호출'인지 아닌지 모름

    포인터는 참조자 대체 권장, **값에 의한 호출인지 참조에 의한 호출인지 구별** 필요

    함수의 실인수로 기술한 변수가 함수 호출 때문에 값이 변경될 수 있기 때문이다.

    함수의 **매개변수 형식이 클래스 형식이라면 상수형 참조**로 선언

깊은 복사와 얕은 복사

- 깊은 복사 : 복사에 의해 실제로 두 개의 값이 생성되는 것
- 얕은 복사 : 대상이 되는 값은 여전히 하나뿐인데 접근 포인터만 둘로 늘어나는 것

▼ 얕은 복사의 문제점 코드

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

int main()
{
    // 그'들'
    int* pA, * pB;

    // 한 친구의 그녀 탄생
    pA = new int;
    *pA = 10;

    // 자기 여자 친구 놔두고 친구의 친구를 마음에 담은 바보
    pB = new int;
    pB = pA;

    // 그렇게 모두 잘 지내는 것처럼 보인다.
    cout << *pA << endl;
    cout << *pB << endl;

    // 그럼 이건?
    delete pA;
    delete pB;

    return 0;
}
```

▼ 포인터가 없는 복사 생성자 사용 코드

```
/* 포인터가 없는 복사 생성자 사용 */
#include "pch.h"
#include <iostream>
using namespace std;

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

    int GetData(void) const { return m_nData; }
    void SetData(int nParam) { m_nData = nParam; }

private:
    int m_nData = 0;
```

```
};

int main()
{
    // 디폴트 생성자가 호출되는 경우
    CMyData a;
    a.SetData(10);

    // 복사 생성자가 호출되는 경우
    CMyData b(a);
    cout << b.GetData() << endl;

    return 0;
}
```

▼ 포인터가 존재했을 때의 얕은 복사 코드

```
/* 포인터가 존재했을 때의 얕은 복사 */
#include "pch.h"
#include <iostream>
using namespace std;

class CMyData
{
public:
    CMyData(int nParam)
    {
        m_pnData = new int;
        *m_pnData = nParam;
    }

    int GetData()
    {
        if (m_pnData != NULL)
            return *m_pnData;

        return 0;
    }

private:
    // 포인터 멤버 데이터
    int* m_pnData = nullptr;
};

int main()
{
    CMyData a(10);
    CMyData b(a);
    cout << a.GetData() << endl;
```

```
    cout << b.GetData() << endl;

    return 0;
}
```

이 코드는 깊은 복사로 처리하지 않았기 때문에 메모리를 해제 시 오류가 발생

▼ 해당 코드를 고친 코드

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

class CMyData
{
public:
    CMyData(int nParam)
    {
        m_pnData = new int;
        *m_pnData = nParam;
    }

    // 복사 생성자 선언 및 정의
    CMyData(const CMyData& rhs)
    {
        cout << "CMyData(const CMyData &)" << endl;

        // 메모리를 할당한다.
        m_pnData = new int;

        // 포인터가 가리키는 위치에 값을 복사한다.
        *m_pnData = *rhs.m_pnData;
    }

    // 객체가 소멸하면 동적 할당한 메모리를 해제한다.
    ~CMyData()
    {
        delete m_pnData;
    }

    int GetData()
    {
        if (m_pnData != NULL)
            return *m_pnData;

        return 0;
    }

private:
```

```
    // 포인터 멤버 데이터
    int* m_pnData = nullptr;
};

int main()
{
    CMyData a(10);
    CMyData b(a);
    cout << a.GetData() << endl;
    cout << b.GetData() << endl;

    return 0;
}
```

대입 연산자

만약 위의 코드에서 a=b;를 한다면 단순 대입으로 얕은 복사가 되므로 문제가 일어난다.

▼ 올바르게 복사 생성자를 사용하는 예제 코드

```
/* 올바르게 복사 생성자를 사용하는 예제 */
#include "pch.h"
#include <iostream>
using namespace std;

class CMyData
{
public:
    CMyData(int nParam)
    {
        m_pnData = new int;
        *m_pnData = nParam;
    }

    // 복사 생성자 선언 및 정의
    CMyData(const CMyData& rhs)
    {
        cout << "CMyData(const CMyData &)" << endl;

        // 메모리를 할당한다.
        m_pnData = new int;

        // 포인터가 가리키는 위치에 값을 복사한다.
        *m_pnData = *rhs.m_pnData;
    }

    // 객체가 소멸하면 동적 할당한 메모리를 해제한다.
    ~CMyData()
    {
```

```
        delete m_pnData;
    }

    // 단순 대입 연산자 함수를 정의한다.
    CMyData& operator=(const CMyData& rhs)
    {
        *m_pnData = *rhs.m_pnData;

        // 객체 자신에 대한 참조를 반환한다.
        return *this;
    }

    int GetData()
    {
        if (m_pnData != NULL)
            return *m_pnData;

        return 0;
    }

private:
    // 포인터 멤버 데이터
    int* m_pnData = nullptr;
};

int main()
{
    CMyData a(10);
    CMyData b(20);

    // 단순 대입을 시도하면 모든 멤버의 값을 그대로 복사한다.
    a = b;
    cout << a.GetData() << endl;

    return 0;
}
```

묵시적 변환

아무 언급없이 표현

변환 생성자

매개변수를 한 개이면 불필요한 임시 객체를 만들어냄으로써 프로그램의 효율을 갉아먹는 원인

▼ int 자료형의 변환 생성자 코드

```
/* int 자료형의 변환 생성자 */
#include "pch.h"
#include <iostream>
using namespace std;

// 제작자 코드
class CTestData
{
public:
    // 매개변수가 하나뿐인 생성자는 형변환이 가능하다.
    CTestData(int nParam) : m_nData(nParam)
    {
        cout << "CTestData(int)" << endl;
    }

    CTestData(const CTestData& rhs) : m_nData(rhs.m_nData)
    {
        cout << "CTestData(const CTestData &)" << endl;
    }
    ~CTestData()
    {
        cout << "~CTestData()" << endl;
    }

    int GetData() const { return m_nData; }
    void SetData(int nParam) { m_nData = nParam; }

private:
    int m_nData = 0;
};

// 사용자 코드
// 매개변수가 클래스 형식이며 반환 생성이 가능하다.
void TestFunc(const CTestData param)
{
    cout << "TestFunc(): " << param.GetData() << endl;
}

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

    // 새로운 CTestData 객체를 생성하고 참조를 전달한다,
    TestFunc(5);

    // 함수가 반환되면서 임시 객체는 소멸한다.
    cout << "*****End*****" << endl;
```

```
    return 0;
}
```

▼ 위의 코드의 추가 코드

```
/* int 자료형의 변환 생성자 */
#include "pch.h"
#include <iostream>
using namespace std;

// 제작자 코드
class CTestData
{
public:
    // 매개변수가 하나뿐인 생성자는 형변환이 가능하다.
    CTestData(int nParam) : m_nData(nParam)
    {
        cout << "CTestData(int)" << endl;
    }

    CTestData(const CTestData& rhs) : m_nData(rhs.m_nData)
    {
        cout << "CTestData(const CTestData &)" << endl;
    }
    ~CTestData()
    {
        cout << "~CTestData()" << endl;
    }

    int GetData() const { return m_nData; }
    void SetData(int nParam) { m_nData = nParam; }

private:
    int m_nData = 0;
};

// 사용자 코드
// 매개변수가 클래스 형식이며 반환 생성이 가능하다.
void TestFunc(const CTestData &param)
{
    cout << "TestFunc(): " << param.GetData() << endl;
}

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

    // 새로운 CTestData 객체를 생성하고 참조를 전달한다,
    TestFunc(5);
```

```
    // 함수가 반환되면서 임시 객체는 소멸한다.
    cout << "*****End*****" << endl;

    return 0;
}
```

문제 : CTestData 클래스 개체를 선언했거나 동적으로 생성하는 코드는 보이지 않습니다.

하지만 컴파일러가 '알아서' 임시 객체를 생성한 후 이 임시 객체에 대한 참조가 TestFunc( ) 함수로 전달된다. - 임시 객체를 TestFunc( ) 함수를 반환함과 동시에 소멸

"묵시적 변환 생성자를 지원하는 클래스인지 꼭 확인"

explicit : 사용자 코드에서 묵시적 변환이 일어나게끔 하여 임시객체가 생성되지 못 하도록 한다.

└ TestFunc(5) X → TestFunc(CTestDAta(5));

허용되는 변환

▼ 형변환 연산자를 통한 자료형 변환 코드

```
/* 형변환 연산자를 통한 자료형 변환 */
#include "pch.h"
#include <iostream>
using namespace std;

// 제작자 코드
class CTestData
{
public:
    explicit CTestData(int nParam) : m_nData(nParam) { }

    // CTestData 클래스는 int 자료형으로 변환될 수 있다!
    operator int(void) { return m_nData; }

    int GetData() const { return m_nData; }
    void SetData(int nParam) { m_nData = nParam; }

private:
    int m_nData = 0;
};

// 사용자 코드
int main()
{
```

```
    CTestData a(10);
    cout << a.GetData() << endl;

    // CTestData 형식에서 int 자료형으로 변환될 수 있다.
    cout << a << endl;
    cout << (int)a << endl;
    cout << static_cast<int>(a) << endl;

    return 0;
}
```

- const_cast
- dynamic_cast
- static_cast
- reinterpret_cast

임시 객체와 이동 시맨틱

이름 없는 임시 객체

인스턴스지만 '식별자'가 부여되지 않은 객체

▼ **이름 없는 임시 객체의 생성과 소멸 코드**

실행이 안됨

```
/* 이름 없는 임시 객체의 생성과 소멸 */
#include "pch.h"
#include <iostream>
using namespace std;

// 제작자 코드
class CTestData
{
public:
    CTestData(int nParam, char *pszName) : m_nData(nParam), m_pszName(pszName)
    {
        cout << "CTestData(int): " << m_pszName << endl;
    }

    ~CTestData()
    {
        cout << "~CTestData(): " << m_pszName << endl;
    }

    CTestData(const CTestData &rhs) : m_nData(rhs.m_nData), m_pszName(rhs.m_pszName)
    {
```

```
        cout << "CTestData(const CTestData &): " << m_pszName << endl;
    }

    CTestData& operator= (const CTestData &rhs)
    {
        cout << "operator=" << endl;

        // 값은 변경하지만 이름(m_pszName)은 그대로 둔다.
        m_nData = rhs.m_nData;

        return *this;
    }

    int GetData() const { return m_nData; }
    void SetData(int nParam) { m_nData = nParam; }

private:
    int m_nData = 0;

    // 변수 이름을 저장하기 위한 멤버
    char *m_pszName = nullptr;
};

// CTestData 객체를 반환하는 함수다.
CTestData TestFunc(int nParam)
{
    // CTestData 클래스 인스턴스인 a는 지역 변수다.
    // 따라서 함수가 반환되면 소멸한다.
    CTestData a(nParam, "a");

    return a;
}

int main()
{
    CTestData b(5, "b");
    cout << "*****Before*****" << endl;

    // 함수가 반환되면서 임시 객체가 생성됐다가 대입 연산 후 즉시 소멸한다!
    b = TestFunc(10);
    cout << "*****After*****" << endl;
    cout << b.GetData() << endl;

    return 0;
}
```

코드 행	코드	결과 행	결과	비고
56	CTestData b(5, ”b”);	1	CTestData(int): b	

코드 행	코드	결과 행	결과	비고
49	CTestData a(nParam, ”a”);	3	CTestData(int): a	
60	b = TestFunc(10);	4	CTestData(const CTestData &): a	이름 없는 임시 객체 복사 생성
51	return a;	5	~CTestData( ): a	TestFunc( ) 함수 반환, 이름 없는 임시 객체의 원본 객체 소멸
60	b = TestFunc(10);	6	operator=	
60	b = TestFunc(10);	7	~CTestData( ): a	이름 없는 임시 객체 소멸
62	cout << b.GetData( ) << endl;	9	10	

별명을 지정해주면 tmain( )함수 끝날 때까지로 생명주기가 늘어난다.

이름없는 임시 객체를 인식하고, 생성 및 소멸 시기를 정확하게 인식하는 것이 중요하다.

특히 소멸 시기를 인식하는 것이 중요하다.

r-value 참조

단순 대입연산자의 오른쪽 항 - &가 두 번 붙음

▼ r-value 참조의 사용 예 1 코드

```
/* r-value 참조의 사용 예 1 */
#include "pch.h"
#include <iostream>
using namespace std;

int main()
{
    int&& data = 3 + 4;
    cout << data << endl;
    data++;
    cout << data << endl;

    return 0;
}
```

r-value : 연산에 따라 생성된 임시 객체

▼ r-value 참조의 사용 예 2 코드

```
/* r-value 참조의 사용 예 2 */
#include "pch.h"
#include <iostream>
using namespace std;

void Testfunc(int& rParam)
{
    cout << "TestFunc(int &)" << endl;
}

void TestFunc(int&& rParam)
{
    cout << "TestFunc(int &&)" << endl;
}

int main()
{
    // 3 + 4 연산 결과는 r-value이다. 절대로 l-value가 될 수 없다.
    TestFunc(3 + 4);

    return 0;
}
```

▼ 다중 정의의 모호성 코드

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

// r-value 참조 형식
void TestFunc(int&& nParam)
{
    cout << "TestFunc(int &&)" << endl;
}

// r-value 참조 형식과 호출자 코드가 같다.
void TestFunc(int nParam)
{
    cout << "TestFunc(int)" << endl;
}

int main()
{
```

```
    // 모호한 호출이다. int형과 int&&형 모두 가능하다!
    TestFunc(3 + 4);

    return 0;
}
```

TestFunc(int)와 TestFunc(int &&)에도 적용되기 때문에 모호한 오류가 발생

이동 시맨틱

복사 생성자와 대입 연산자에 r-value 참조를 조합해서 새로운 생성 및 대입의 경우를 만들어낸 것

▼ 이동 생성자의 호출 시점 코드

```
/* 이동 생성자의 호출 시점 */
#include "pch.h"
#include <iostream>
using namespace std;

// 제작자 코드
class CTestData
{
public:
    CTestData() { cout << "CTestData()" << endl; }
    ~CTestData() { cout << "~CTestData()" << endl; }

    CTestData(const CTestData& rhs) : m_nData(rhs.m_nData)
    {
        cout << "CTestData(const CTestData &)" << endl;
    }

    // 이동 생성자
    CTestData(CTestData&& rhs) : m_nData(rhs.m_nData)
    {
        cout << "CTestData(const CTestData &&)" << endl;
    }
    CTestData& operator=(const CTestData&) = default;

    int GetData() const { return m_nData; }
    void SetData(int nParam) { m_nData = nParam; }

private:
    int m_nData = 0;
};

CTestData TestFunc(int nParam)
{
```

```
    cout << "**TestFunc(): Begin***" << endl;
    CTestData a;
    a.SetData(nParam);
    cout << "**TestFunc(): End*****" << endl;

    return a;
}

int main()
{
    CTestData b;
    cout << "*Before***************" << endl;
    b = TestFunc(20);
    cout << "*After****************" << endl;
    CTestData c(b);

    return 0;
}
```

b=TestFunc(20);이 실행되면 CTestData(CTestData &&)이 호출

사라질 객체이므로 깊은 복사를 수행하는 것이 아니라 얕은 복사를 수행함으로써 성능을 높이는 것

이 장의 내용은 어려우니 어려번 봐야 할 것이다.

문제

1. **함수의 매개변수가 기본 형식이 아니라 클래스라면 매개변수 형식은 뭐가 좋은가**

상수형 참조

2. **복사 생성자 및 단순 대입 연산자를 반드시 정의해야 하는 클래스**

깊은 복사

3. **이 코드의 잠재적 문제**

```
void TestFunc(const CTestData &param)
{
  ......
}

int main()
{
  TestFunc(5);
```

4. **이동 시맨틱이 등장한 원인**

이름 없는 객체의 부하를 최소화하기 위해서 이다.

```
  return 0;
}
```

객체를 선언하는 것이 없다.