04

Chapter 4

복사 생성자와 임시 객체

4장의 핵심 개념

복사 생성자, 깊은 복사와 얕은 복사, 임시 객체, 이동 시맨틱

- **복사 생성자**
  : 객체의 복사본을 생성할 때 호출되는 생성자이다.
- **깊은 복사와 얕은 복사**
  : 실제 값을 두 개로 만드는 깊은 복사와 값은 하나이나 포인터만 두 개를 생성하는 얕은 복사의 차이를 배운다.
- **임시 객체**
  : 컴파일러가 임의로 생성했다가 바로 소멸시키는 객체.
- **이동 시맨틱**
  : 임시 객체가 생성됐을 때 부하를 최소화 하기 위한 문법으로 C++11부터 지원한다.

복사 생성자

객체의 복사본을 생성(혹은 선언 및 정의)할 때 호출되는 생성자.
class-name(const class-name &rhs);

```
class CMyData
{
public:
	...
	// 복사 생성자 선언 및 정의
	CMyData(const CMyData &rhs)
		// : m_nData(rhs.m_nData)
	{
		this->m_nData = rhs.m_nData;
	...
};
int _tmain(int argc, _TCHAR* argv[])
{
	...
	// 복사 생성자가 호출되는 경우
	CMyData b(a);
	cout << b.GetData() << endl;
	...
}
```

함수 호출과 복사 생성자

이 코드 등장하는 CTest 클래스 인스턴스는 모두 몇 개 인가?

```
CTest TestFunc1() { ... }
void TestFunc(CTest param) { }
void _tmain(int argc, _TCHAR* argv[])
{
	CTest a;
	TestFunc(a);
}
```

함수 호출과 복사 생성자

이 코드에서는 쓸데없이 CTestData 객체가 두 개 이다.

```
void TestFunc(CTestData param)
{
	cout << "TestFunc()" << endl;
	// 피호출자 함수에서 매개변수 인스턴스의 값을 변경한다.
	param.SetData(20);
}
int _tmain(int argc, _TCHAR* argv[])
{
	cout << "*****Begin*****" << endl;
	CTestData a(10);
	TestFunc(a);
	// 함수 호출 후 a의 값을 출력한다.
	cout << "a: " << a.GetData() << endl;
	cout << "******End******" << endl;
	return 0;
}
```

call by value

함수 호출과 복사 생성자

단 한 글자만 추가함으로써 객체의 개수를 하나로 줄일 수 있다!

```
void TestFunc(CTestData &param)
{
	cout << "TestFunc()" << endl;
	// 피호출자 함수에서 매개변수 인스턴스의 값을 변경한다.
	param.SetData(20);
}
int _tmain(int argc, _TCHAR* argv[])
{
```

& call by ref

```
        param.SetData(20);
}
int _tmain(int argc, _TCHAR* argv[])
{
        cout << "*****Begin*****" << endl;
        CTestData a(10);
        TestFunc(a);
        // 함수 호출 후 a의 값을 출력한다.
        cout << "a: " << a.GetData() << endl;
        cout << "******End******" << endl;
        return 0;
}
```

20 출력.

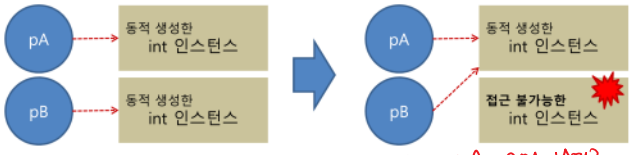
깊은 복사와 얕은 복사

아래 코드는 메모리가 해제될 때 Crash가 발생한다.

```
int _tmain(int argc, _TCHAR* argv[])
{
        // 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;
}
```

깊은 복사와 얕은 복사

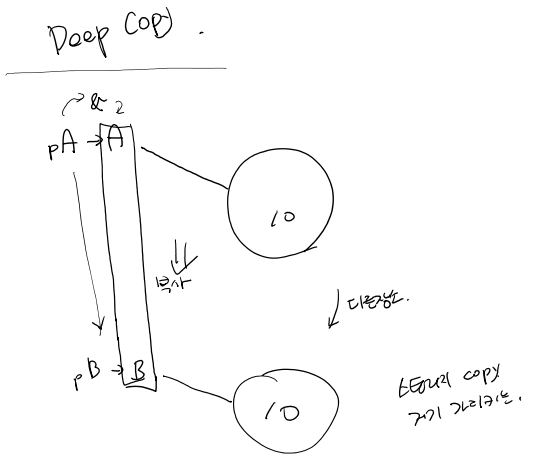
포인터에 대해 단순 대입한다고 해서 대상 메모리까지 대입되는 것이 아니다.

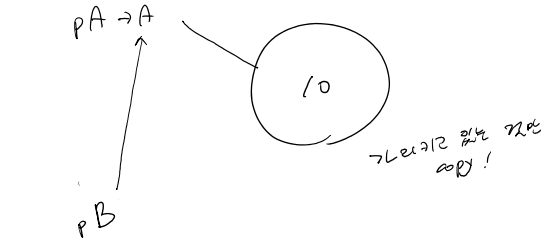


깊은 복사와 얕은 복사

깊은 복사는 포인터가 가리키는 메모리에 대한 별도의 사본을 만드는 것이다.

```
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;
        }
        ...
```





깊은 복사와 얕은 복사

단순 대입 연산자 '함수'에서도 동일한 문제가 발생한다.
즉, 깊은 복사가 필요하다면 복사 생성자 외에 대입 연산자도 함께 정의해야 !

```
int _tmain(int argc, _TCHAR* argv[])
{
        CMyData a(10);
        CMyData b(20);

        // 단순 대입을 시도하면 모든 멤버의 값을 그대로 복사한다.
        a = b;          // b(a) 와 같은 효과

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

a.operator = (b)

깊은 복사와 얕은 복사

복사 생성자와 단순 대입 연산자의 코드는 비슷한 구조를 갖는다.

```
// 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;
}
...
// 단순 대입 연산자 함수를 정의한다.
CMyData& operator=(const CMyData &rhs)
{
        *m_pnData = *rhs.m_pnData;
        // 객체 자신에 대한 참조를 반환한다.
        return *this;
}
```

레퍼런스 반환자

변환 생성자

매개변수가 한 개인 생성자이다.
이 코드는 형식인수와 실인수 형식이 다름에도 컴파일 오류가 발생하지 않는다.

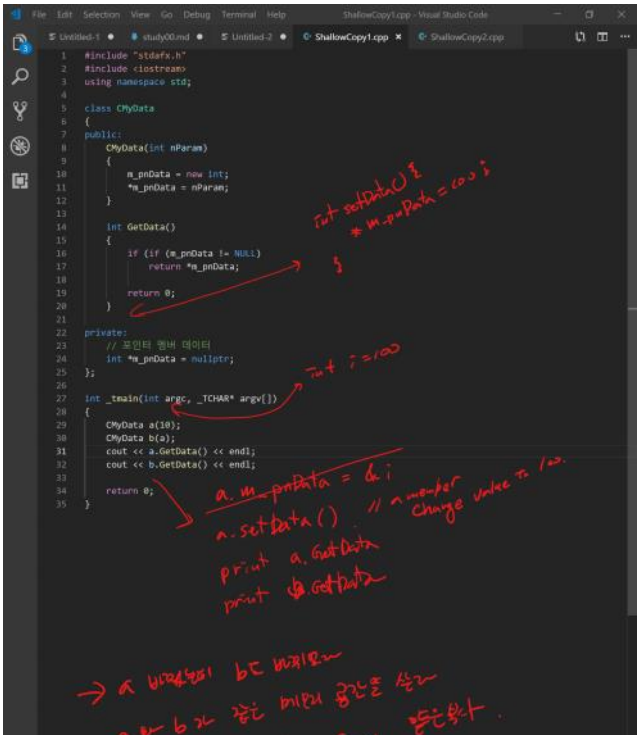
```
class CTestData
{
public:
        CTestData(int nParam) : m_nData(nParam)
        {
                cout << "CTestData(int)" << endl;
        }
        ...
};
void TestFunc(CTestData param)
{
        cout << "TestFunc(): " << param.GetData() << endl;
}
int _tmain(int argc, _TCHAR* argv[])
{
        TestFunc(5);    // 변환이 컴파일러에 의해 implicit 하게 발생한다.
        return 0;
}
```

int 5 → CTestData. 오류가 나지 않는다!

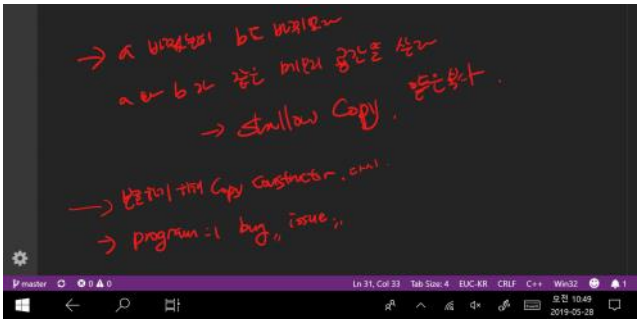
변환 생성자

매개변수가 한 개인 생성자이다.
이 코드에 등장하는 CTestData 클래스의 인스턴스 수는?

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



```
        {
                cout << "CTestData(int)" << endl;
        }
        ...
};
void TestFunc(const CTestData & param)
{
        cout << "TestFunc(): " << param.GetData() << endl;
}
int _tmain(int argc, _TCHAR* argv[])
{
        TestFunc(5);
        return 0;
}
```



변환 생성자

변환 생성자를 선언할 때는 반드시 explicit로 선언한다.

```
class CTestData
{
public:
    explicit CTestData(int nParam) : m_nData(nParam)
        {
                cout << "CTestData(int)" << endl;
        }
        ...
};
void TestFunc(const CTestData & param)
{
        cout << "TestFunc(): " << param.GetData() << endl;
}
int _tmain(int argc, _TCHAR* argv[])
{
        TestFunc(5);  // error !!! By compiler !!!
        TestFunc( CTest (5) )  // no error !!!
        return 0;
}
```

허용되는 변환

허용되는 변환 형식을 규정하면 형식간의 호환성이 생긴다.

```
class CTestData
{
public:
        explicit CTestData(int nParam) : m_nData(nParam) { }
        // CTestData 클래스는 int 자료형식으로 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;
};
int _tmain(int argc, _TCHAR* argv[])
{
        cout << a << endl;
        return 0;
}
```

이름 없는 임시 객체

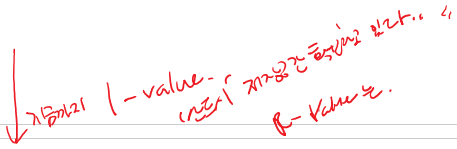
함수 반환이나 연산 과정에서 코드에 보이지 않는 인스턴스가 생겼다 사라진다.

```
// CTestData 객체를 반환하는 함수다.
CTestData TestFunc(int nParam)
{
        // CTestData 클래스 인스턴스인 a는 지역변수지역 변수다.
        // 따라서 함수가 반환되면 소멸한다.
        CTestData a(nParam, "a");
        return a;
}
int _tmain(int argc, _TCHAR* argv[])
{
        CTestData b(5, "b");
        cout << "*****Before*****" << endl;
        // 함수가 반환하면서 임시 객체가 생성됐다가 대입 연산 후 즉시 소멸한다!
        b = TestFunc(10);
        cout << "*****After*****" << endl;
        cout << b.GetData() << endl;
        return 0;
}
```

이름 없는 임시 객체

임시 객체의 생성과 소멸 규칙

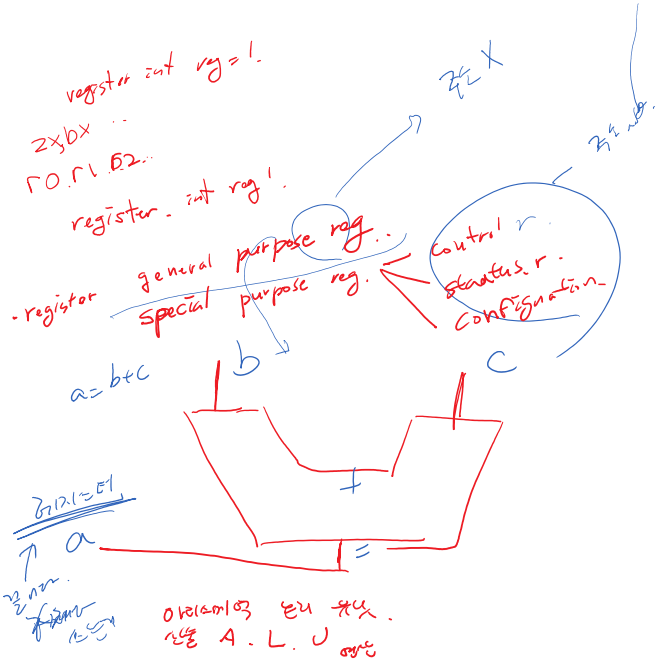
- 임시 객체는 함수 반환이나 연산 과정에서 생겨난다.
- 임시 객체는 모두 r-value이다.
- 임시 객체는 이어지는 연산에 참여직후 자동으로 소멸한다.
- 만일 이름 없는 임시 객체에 대해 참조자를 선언할 경우 참조자가 속한 scope가 닫힐 때까지 임시 객체도 살아 남는다.



r-value 참조

int &&nData = 3 + 7; 처럼 (연산의) 임시 결과에 대한 참조자 선언이다.

```
void TestFunc(int &&rParam)
{
        cout << "TestFunc(int &&)" << endl;
}
int _tmain(int argc, _TCHAR* argv[])
{
        // 3 + 4 연산 결과는 r-value이다. 절대로 l-value가 될 수 없다.
        TestFunc(3 + 4);
        return 0;
}
```



이동 시맨틱

복사 생성자와 대입 연산자에 r-value 참조를 조합해서
새로운 생성(이동 생성자) 및 대입(이동 대입 연산자)의 경우를 만들어 낸 것이다.

```
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;
        }
        int GetData() const { return m_nData; }
        void SetData(int nParam) { m_nData = nParam; }
private:
        int m_nData = 0;
};
```

이동 시맨틱

TestFunc() 함수가 반환한 임시 객체에 대한 이동 생성자가 호출된다.
곧 사라질 임시 객체에 대해 얕은 복사를 수행하여 성능을 높이는 것이 핵심이다.

```
CTestData TestFunc(int nParam)
{
        cout << "**TestFunc(): Begin***" << endl;
        CTestData a;
        a.SetData(nParam);
        cout << "**TestFunc(): End*****" << endl;
        return a;
}
int _tmain(int argc, _TCHAR* argv[])
{
        CTestData b;
        cout << "*Before***************" << endl;
        b = TestFunc(20);
        cout << "*After****************" << endl;
        CTestData c(b);
        return 0;
}
```