20. 임시객체와 이동시맨틱

■ 이름 없는 임시객체

- 변환 생성자가 묵시적으로 호출되는 것을 explicit 예약어로 막으려는 이유는 사용자 코드에서 보이지 않는 객체가 생성되고 소멸되는 것을 막기 위함.
- 더 은밀한 임시객체도 있음 – 이름이 없음, 대부분 생성 직후 곧바로 소멸하여 생성을 막을수 없음 내부 연산이 최소화되도록 코드를 줄이는 방법으로 대응해야 함!
- 함수의 반환 형식이 클래스일 때 이름 없는 임시객체가 발생함!
- 이름 없는 임시객체는 존재하는 인스턴스지만 식별자가 부여되지 않은 객체!

```
#include <iostream>
using namespace std;
// 제작자 코드
class CTestData
{
public:
    CTestData() :m_nData(0) {
        cout << " CTestData() " << endl;
    }
    CTestData(int num) :m_nData(num) {
        cout << " CTestData(int) " << endl;
    }
    CTestData(int num, const char* pszName):m_nData(num) {
        m_pszName = (char*)pszName;
        cout << " CTestData(int, char* ) " << m_pszName << endl;
    }
    CTestData(const CTestData& rhs):m_nData(rhs.m_nData), m_pszName(rhs.m_pszName) {
        cout << " CTestData(const CTestData& rhs) " << m_pszName << endl;
    }
    CTestData& operator=(const CTestData& rhs) {
        m_nData = rhs.m_nData;
        m_pszName = rhs.m_pszName;
        cout << " operator=(const CTestData& rhs) " << m_pszName << endl;
        return *this;
    }
    ~CTestData() { cout << " ~CTestData() " << m_pszName << endl; }

    void SetData(int nParam) { m_nData = nParam; }
    int GetData() { return m_nData; }
private:
    int m_nData = 0;
    // 변수 이름을 저장하기 위한 멤버
    char* m_pszName = nullptr;
};
```

```
// 사용자 코드

// CTestData 객체를 반환하는 함수다.
CTestData TestFunc(int num)
{
    cout << "--------- CTestData t() 전 " << endl;
    // CTestData 클래스 인스턴스인 t는 지역변수다.
    // 따라서 함수가 반환되면 소멸한다.
    CTestData t(num, "T");
    cout << "--------- CTestData t() 후 " << endl;
    return t;
}
int main()
{
    cout << "--------- main() Begin " << endl;
    CTestData a(30, "A");
    CTestData b(a);
    CTestData c;
    cout << "--------- TestFunc() 호출 전 " << endl;
    // 함수가 반환되면서 임시객체가 생성됐다가 대입 연산 후 즉시 소멸한다!
    c = TestFunc(50);
    cout << "--------- TestFunc() 호출 후 " << endl;
    cout << "--------- main() End " << endl;
    return 0;
}
```

- 실행결과

```
--------- main() Begin
 CTestData(int, char* ) A
 CTestData(const CTestData& rhs) A
 CTestData()
--------- TestFunc() 호출 전
--------- CTestData t() 전
 CTestData(int, char* ) T
--------- CTestData t() 후
 CTestData(const CTestData& rhs) T // 이름 없는 임시객체 생성(복사 생성)
 ~CTestData() T
 operator=(const CTestData& rhs) T
 ~CTestData() T                    // 이름 없는 임시객체 삭제
--------- TestFunc() 호출 후
--------- main() End
 ~CTestData() T
 ~CTestData() A
 ~CTestData() A
```

- c = TestFunc(50); 코드 한줄 실행하는 동안 TestFunc 함수를 반환하면서 임시객체가 생성되고 대입 연산이 끝나면서 소멸함.
- 이름 없는 임시객체의 원본은[CTestData t(num, "T");] 임시객체의 복사 생성이 끝난 후 소멸
- cout << TestFunc(40).GetData() << endl; 라는 코드를 작성하는 것이 가능함.
  함수가 반환한 임시객체(인스턴스)에 멤버 접근 연산을 실행하는 것임.

■ r-value 참조

- r-value는 단순 대입 연산자의 오른쪽 항
- 변수가 올 수도 있지만 3이나 4 같은 리터럴 상수가 올 수도 있음.
- 숫자 3에 4를 대입할 수 없듯이 변수가 아닌 대상에 참조를 선언하는 것은 본래 허용되지 않음
- C++11 표준이 정해지면서 r-value 에 대한 참조자가 새롭게 제공
- r-value 참조자는 & 가 두 번 사용
- int 자료형에 대한 r-value 참조형식은 int&& 임.

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

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

    return 0;
}
```

- data는 int 자료형에 대한 r-value 참조자임. 초기값이 3+4인데 이를 계산해서 7로 결정
- int&& data = 7와 다르지 않음.
- 변수가 아니라 상수에 대한 참조인데 단항 연상을 수행할 수가 있음.(int data로 선언한 것과 비교)
- r-value 는 '연산에 따라 생성된 임시 객체' 라는 것

```
#include <iostream>
using namespace std;
void TestFunc(int& nParam)
{
    cout << "TestFunc(int&)" << endl;
}
void TestFunc(int&& nParam)
{
    cout << "TestFunc(int&&)" << endl;
}
int main()
{
    int x = 3;

    TestFunc(x);  // 변수 참조 l-value
    TestFunc(4);  // 상수 참조 r-value
    TestFunc(x+5); // 연산 결곽는 r-value이다. 절대로 l-value가 될수 없다.
    TestFunc(5 + 6);// 연산 결곽는 r-value이다. 절대로 l-value가 될수 없다.

    return 0;
}
```

● 매개변수 형식에 따른 실인수 예

매개변수 형식	실인수 예	비고
TestFunc(int)	int x = 3; TestFunc(x); TestFunc(4); TestFunc(x + 5); TestFunc(5 + 6);	
TestFunc(int&)	int x = 3; TestFunc(x);	
TestFunc(int&&)	int x = 3; TestFunc(4); TestFunc(x + 5); TestFunc(5 + 6);	TestFunc(x)는 불가능!

● 다중정의의 모호성

```
#include <iostream>
using namespace std;
// r-value 참조형식
void TestFunc(int&& nParam)
{
    cout << "TestFunc(int&&)" << endl;
}
// r-value 참조형식과 호출자 코드가 같다.
void TestFunc(int nParam)
{
    cout << "TestFunc(int)" << endl;
}
int main()
{
    int x = 3;

    TestFunc(x);
    TestFunc(4);     // 모호한 호출이다. int형과 int&& 형 모두 가능하다!
    TestFunc(x + 5); // 모호한 호출이다. int형과 int&& 형 모두 가능하다!
    TestFunc(5 + 6); // 모호한 호출이다. int형과 int&& 형 모두 가능하다!

    return 0;
}
```

```
error C2668: 'TestFunc': 오버로드된 함수에 대한 호출이 모호합니다.
```

● r-value 참조가 꼭 필요한 순간은 int 같은 기본 자료형이 아니라 '클래스에 적용될 때'!!

■ 이동 시맨틱

- C++11에 새로 추가된 이동 시맨틱(Move semantics)은 이동 생성자와 이동 대입 연산자로 구현됨.
- 이동 시맨틱이 생겨난 이유는 '이름 없는 임시객체' 때문
- 임시객체가 생성되더라도 부하가 최소화 될 수 있도록 구조를 변경하기 위함
- 복사 생성자와 대입 연산자에 r-value 참조를 조합해서 새로운 생성 및 대입의 경우를 만들어 낸 것!

```
#include <iostream>
using namespace std;
// 제작자 코드
class CTestData
{
public:
    CTestData() :m_nData(0) {
        cout << " CTestData() " << endl;
    }
    CTestData(int num) :m_nData(num) {
        cout << " CTestData(int) " << endl;
    }
    CTestData(int num, const char* pszName):m_nData(num) {
        m_pszName = (char*)pszName;
        cout << " CTestData(int, char* ) " << m_pszName << endl;
    }
    // 이동 생성자
    CTestData(CTestData &&rhs):m_nData(rhs.m_nData), m_pszName(rhs.m_pszName) {
        cout << " CTestData(CTestData &&rhs) " << m_pszName << endl;
    }
    CTestData(const CTestData& rhs):m_nData(rhs.m_nData), m_pszName(rhs.m_pszName) {
        cout << " CTestData(const CTestData& rhs) " << m_pszName << endl;
    }
    CTestData& operator=(const CTestData& rhs) {
        m_nData = rhs.m_nData;
        m_pszName = rhs.m_pszName;
        cout << " operator=(const CTestData& rhs) " << m_pszName << endl;
        return *this;
    }
    ~CTestData() { cout << " ~CTestData() " << m_pszName << endl; }

    void SetData(int nParam) { m_nData = nParam; }
    int GetData() { return m_nData; }
private:
    int m_nData = 0;
    // 변수 이름을 저장하기 위한 멤버
    char* m_pszName = nullptr;
};
```

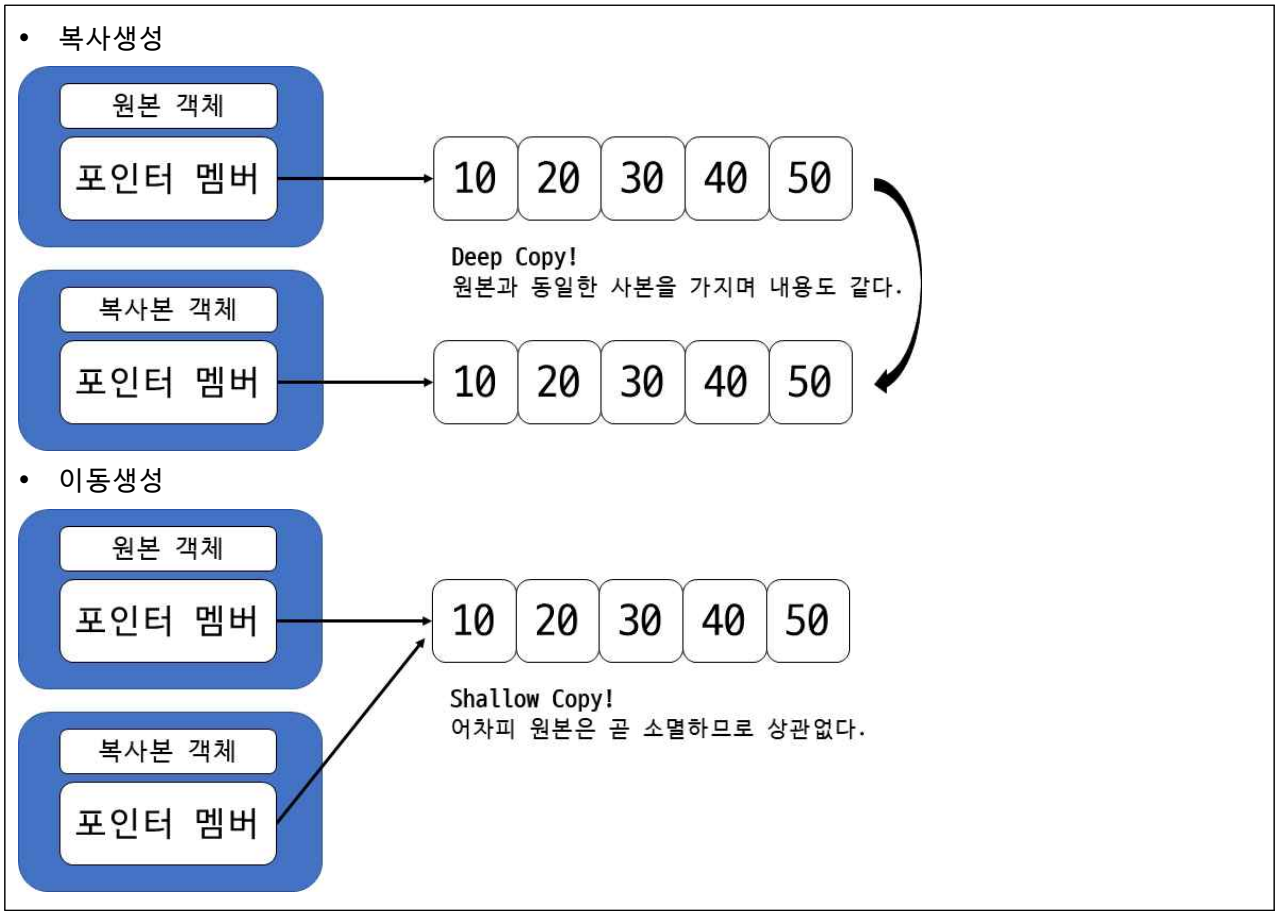
```
// 사용자 코드
// CTestData 객체를 반환하는 함수다.
CTestData TestFunc(int num)
{
    cout << "------------------------- TestFunc() Start " << endl;
    CTestData t(num, "T");
    cout << "------------------------- TestFunc() End " << endl;
    return t;
}
int main()
{
    cout << "------------------------- main() Begin " << endl;
    CTestData c;
    cout << "------------------------- TestFunc() 호출 전 " << endl;
    c = TestFunc(50);
    cout << "------------------------- TestFunc() 호출 후 " << endl;
    cout << "------------------------- main() End " << endl;
    return 0;
}
```

- 실행결과

```
------------------------- main() Begin
 CTestData()
------------------------- TestFunc() 호출 전
------------------------- TestFunc() Start
 CTestData(int, char* ) T
------------------------- TestFunc() End
 CTestData(CTestData &&rhs) T      // 이름 없는 임시객체 생성(이동 생성)
 ~CTestData() T
 operator=(const CTestData& rhs) T
 ~CTestData() T                    // 이름 없는 임시객체 소멸
------------------------- TestFunc() 호출 후
------------------------- main() End
 ~CTestData() T
계속하려면 아무 키나 누르십시오 . . .
```

- 임시 객체가 만들어질 때 복사생성이 아닌 다른 방법으로 생성이 된 것
- Shallow Copy가 문제가 될 때(멤버변수의 메모리 동적할당, 해제시) 복사생성자를 정의해서 Deep Copy로 해결하는 부분을 학습했음.
- 임시객체는 어차피 사라질 객체이므로 Deep copy를 수행하는 것이 아니라(복사생성자가 아니라) Shallow copy를 수행함으로써 성능을 높이는 것.

- 복사생성과 이동생성



- 일반적인 복사생성의 경우에는 두 배의 메모리 용량이 필요함.
- 메모리를 복사하는 연산을 반드시 실행해야 함(문자열의 경우 strcpy() 함수 실행 필수!)
- 이동 생성자를 이용해서 메모리를 복사하지 않는다면 사라질 객체로부터 알맹이는 빼내고 껍데기만 버리는 것처럼 코드를 만들 수 있음.
- 메모리를 절반 사용하는 데다 반복해서 메모리 복사를 할 필요가 없음.
- 우리가 작성한 클래스가 내부적으로 덩치 큰 영상 파일을 로드하고 인스턴스마다 메모리를 20~30MB 정도 사용하고 있다면 이동 생성자가 있고 없고에 따라 사용하는 메모리의 크기 및 연산 속도는 체감할 수 있을 정도로 달라짐.