18. 복사 생성자

- 복사 생성자(Copy constructor)는 말 그대로 객체의 복사본을 생성할 때 호출되는 생성자임
- 클래스를 작성할 때 복사 생성자를 생략하면 디폴트 생성자처럼 컴파일러가 알아서 만들어 줌.
- 디폴트 복사 생성자는 멤버 대 멤버의 복사를 진행한다. > shallow copy
- 필요한 경우가 아니라면 굳이 정의할 필요 없으나, 복사 생성자를 적용하지 않으면 심각한 문제가 발생하는 경우가 생김!
- 클래스 내부에서 메모리를 동적 할당 및 해제하고 이를 멤버 포인터 변수로 관리하는 경우에 복사 생성자 적용하지 않으면 심각한 문제가 발생함.(멤버변수가 힙의 메모리 공간을 참조하는 경우)
- 복사생성자 형태

```
클래스이름(const 클래스이름 &rhs);
```

```
#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() 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);

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

    return 0;
}
```

■ 함수 호출과 복사생성자

- 복사생성자가 호출되는 경우
  1) 명시적으로 객체의 복사본을 생성하는 방식으로 선언하는 경우
  2) 함수 형태로 호출되는 경우
- 함수형태로 호출할 때는 클래스가 사용되는 경우(매개변수, 반환형식)

```
CTest TestFunc1() {} // 함수형태(클래스가 반환형식으로 사용)
void TestFunc(CTest nParam) {} // 함수형태(클래스가 매개변수로 사용)
int main()
{
    CTest a(10);
    CTest b(a); // 명시적으로 객체의 복사본을 생성하는 방식

    // 함수 형태로 호출
    TestFunc(a);

    return 0;
}
```

- 매개변수로 사용되는 복사생성자

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

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

    // 복사 생성자 선언 및 정의
    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;
}
```

```
**** Begin ****
CTestData(int)
CTestData(const CTestData &)
TestFunc()
a: 10
****  End  ****
계속하려면 아무 키나 누르십시오 . . .
```

- TestFunc(a); → CTestData 객체가 두 개
- 함수가 호출될 때 매개변수인 param은 호출자의 a를 원본으로 두고 복사본이 생성
- 객체가 하나로 끝날 수 있는 것을 두 개가 생성되며 복사생성자까지 호출해야하는 부담 → 성능저하
- 한 객체로 할 수 있는 일은 반드시 하나로 끝내야 함!

● 쓸데없이 객체가 복사되는 경우를 차단하는 방법 #1(복사생성자를 삭제)

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

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

    // 복사 생성자를 아예 삭제함으로써 복사 생성을 막는다.
    CTestData(const CTestData& rhs) = delete;

    // 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;
};

void TestFunc(CTestData param)
{
    cout << "TestFunc()" << endl;

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

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

    // 사용자 코드에서 복사 생성이 불가능하다.
    CTestData a(10);
    TestFunc(a);

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

    cout << "****  End  **** " << endl;
    return 0;
}
```

```
error C2280: 'CTestData::CTestData(const CTestData &)': 삭제된 함수를 참조하려고 합니다.
```

- 쓸데없이 객체가 복사되는 경우를 차단하는 방법 #2(참조자를 이용)

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

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

    // 복사 생성자 선언 및 정의
    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 클래스의 '참조' 형식이므로 객체가 생성되지 않는다.
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;
}
```

```
**** Begin ****
CTestData(int)
TestFunc()
a: 20
****  End  ****
계속하려면 아무 키나 누르십시오 . . .
```

- 매개변수가 참조자가 되었을 때 단점???
  사용자의 코드만 봐서는 '참조에 의한 호출'인지 아닌지 도무지 알수 없음

```
CTestData a(10);
TestFunc(a);
```

- 값에 의한 호출인지 참조에 의한 호출인지 꼭 구별해서 알아야 하는 이유는 함수의 실 인수로 기술한 변수가 함수 호출 때문에 값이 변경될 수 있기 때문!!

● 함수의 매개변수 형식이 클래스 형식이라면, 일단 상수형 참조로 선언!(값 변경해야하는 경우 제외)

```
// 매개변수가 CTestData 클래스에 대한 상수형 참조다.
void TestFunc(const CTestData &param)
{
    cout << "TestFunc()" << endl;

    // 피호출자 함수에서 매개변수 인스턴스의 값을 변경할 수 없다.
    param.SetData(20);
}
```

```
error C2662: 'void CTestData::SetData(int)': 'this' 포인터를 'const CTestData'에서
'CTestData &'(으)로 변환할 수 없습니다.
```

● 매개변수가 상수형 참조이므로 함수 내부에서는 객체의 '상수형 메서드만' 호출할 수 있음.

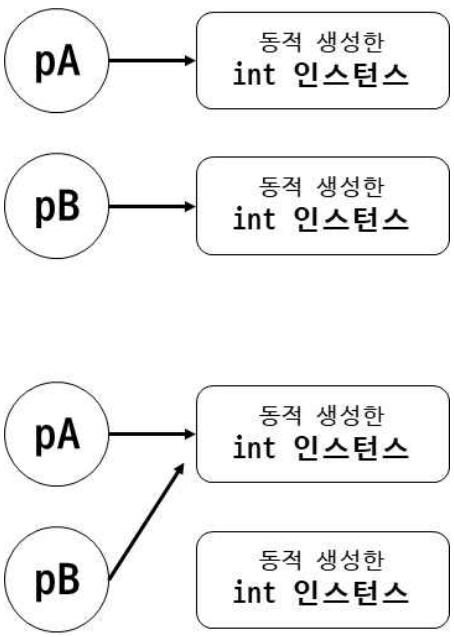
■ 깊은 복사와 얕은 복사

● 깊은 복사(Deep copy)는 복사에 의해 실제로 두 개의 값이 생성

● 얕은 복사(Shallow copy)는 대상이 되는 값은 여전히 하나뿐인데 접근 포인터만 둘로 늘어나는 것

● 얕은 복사의 문제점

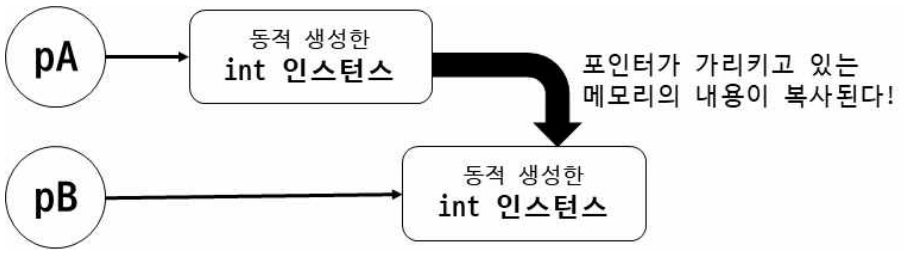
```
#include <iostream>
using namespace std;
int main()
{
    int* pA, * pB;

    pA = new int;
    *pA = 10;

    pB = new int;
    pB = pA; // 얕은 복사!!

    cout << *pA << endl;
    cout << *pB << endl;

    delete pA;
    delete pB;

    return 0;
}
```



● pA, pB 같은 대상을 가리키게 되므로 메모리를 해제하는 과정에서 오류 발생!!!

● 깊은 복사로 수정!

```
    *pB = *pA; // 깊은 복사!!
```



● pA, pB 각각 다른 대상을 가리키게 되므로 메모리를 해제하는 과정에서 오류 발생하지 않음

- 포인터가 없는 복사 생성자 사용

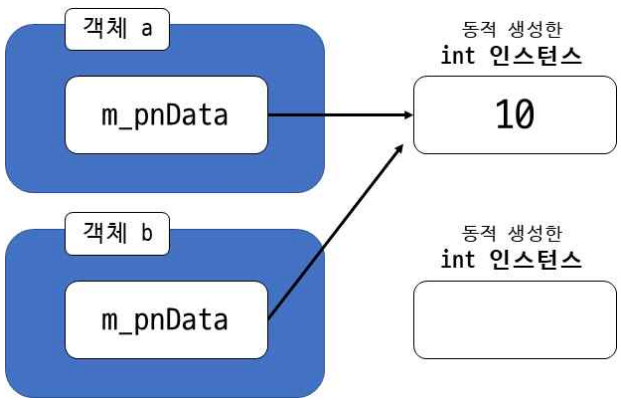
```
#include <iostream>
using namespace std;

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

    int GetData() 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);

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

    return 0;
}
```

```
CMyData()
10
계속하려면 아무 키나 누르십시오 . . .
```

- 포인터를 사용하는 복사생성자의 선언과 정의가 없으므로 CMyData b(a);에서 함수 형태로 호출된 복사 생성자는 아무런 문제 없이 실행되어 10이라는 결과를 출력.

● 포인터가 존재했을 때의 얕은 복사

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

class CMyData {
public:
    CMyData()
    {
        m_pnData = new int;
        *m_pnData = 0;
    }
    CMyData(int nParam)
    {
        m_pnData = new int;
        *m_pnData = nParam;
    }
    int GetData() const
    {
        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;
}
```




```
10
10
계속하려면 아무 키나 누르십시오 . . .
```

● 겉으로 봐서는 정상적으로 잘 작동하는 것처럼 보이지만 깊은 복사로 처리하지 않았기 때문에 심각한 오류가 발생

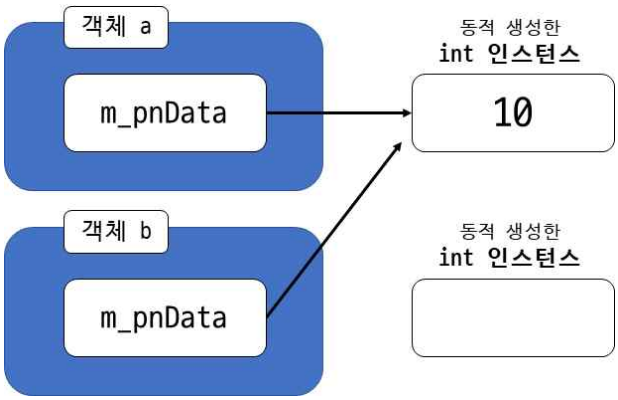
● 메모리를 해제하지 않았기 때문에 오류가 나타나지 않음.

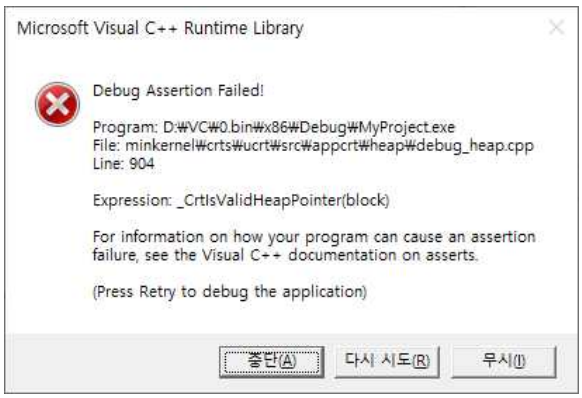
● CMyData b(a); 코드가 실행되는 순간 깊은 복사를 수행할 별도의 복사생성자가 없기 때문에 컴파일러가 만들어 넣은 복사 생성자가 작동함.(아래와 같은 역할의 디폴트 복사생성자)

```
    CMyData(const CMyData &rhs)
    {
        m_pnData = rhs.m_pnData;
    }
```

- 제작자가 메모리 누수를 의식해서 다음과 같이 소멸자를 만들어 넣어 메모리를 해제한다면 심각한 오류를 발생할 수 있음.

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

class CMyData {
public:
    CMyData()
    {
        m_pnData = new int;
        *m_pnData = 0;
    }
    CMyData(int nParam)
    {
        m_pnData = new int;
        *m_pnData = nParam;
    }
    // 객체가 소멸하면 동적 할당한 메모리를 해제한다.
    ~CMyData()
    {
        delete m_pnData;
    }
    int GetData() const
    {
        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;
}
```

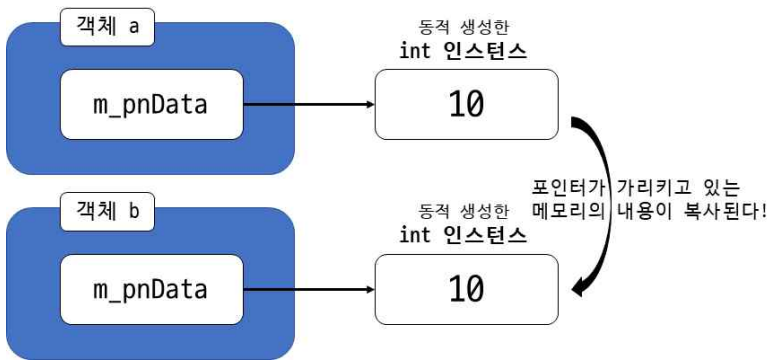




✓ 해제된 메모리를 다시 한 번 더 해제하는 오류가 발생!

● 복사생성자를 넣어 깊은 복사를 수행하도록 문제 해결

```
#include <iostream>
using namespace std;
class CMyData {
public:
    CMyData()
    {
        m_pnData = new int;
        *m_pnData = 0;
    }
    CMyData(int nParam)
    {
        cout << " CMyData(int) " << endl;
        m_pnData = new int;
        *m_pnData = nParam;
    }
    CMyData(const CMyData& rhs)
    {
        cout << " CMyData(const CMyData& rhs) " << 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() const
    {
        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;
}
```

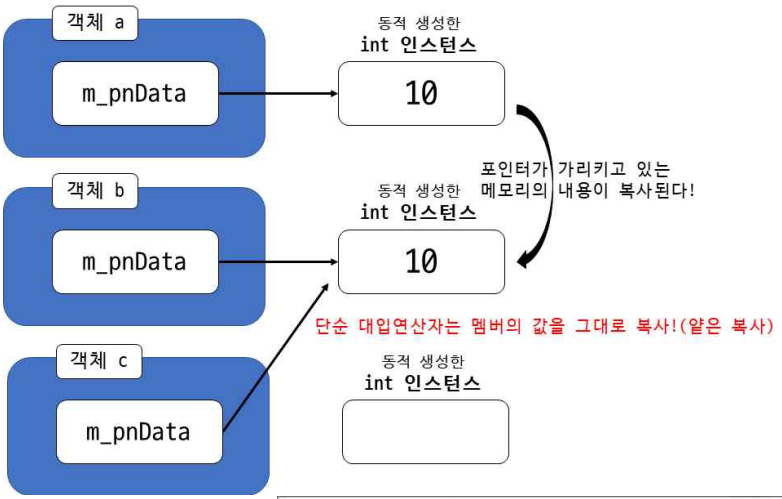


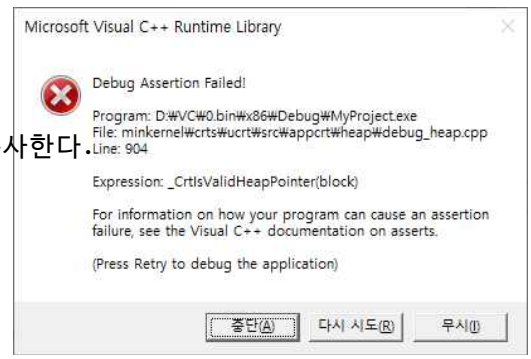

```
 CMyData(int)
 CMyData(const CMyData& rhs)
10
10
계속하려면 아무 키나 누르십시오 . . .
```

■ 대입 연산자

- 단순 대입 연산자는 오른쪽 항의 값을 왼쪽항에 넣는 연산자
- 단순 대입 연산자가 구조체나 클래스에도 기본적으로 적용된다!

```
#include <iostream>
using namespace std;
class CMyData {
public:
    CMyData() {
        cout << " CMyData() " << endl;
        m_pnData = new int;
        *m_pnData = 0;
    }
    CMyData(int nParam) {
        cout << " CMyData(int) " << endl;
        m_pnData = new int;
        *m_pnData = nParam;
    }
    CMyData(const CMyData& rhs) {
        cout << " CMyData(const CMyData& rhs) " << 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() const {
        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);
    CMyData c;

    // 단순 대입을 시도하면 모든 멤버의 값을 그대로 복사한다.
    c = b;

    cout << a.GetData() << endl;
    cout << b.GetData() << endl;
    cout << c.GetData() << endl;

    return 0;
}
```



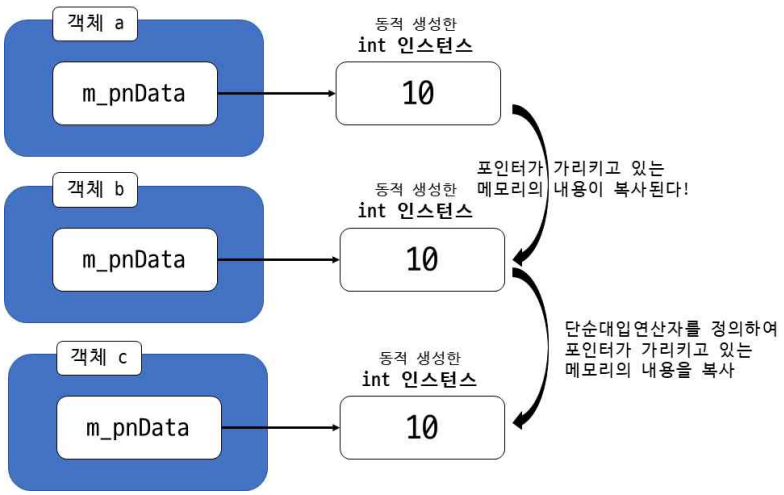


✓ c = b; 코드처럼 단순 대입을 시도하면 기본적으로 얕은 복사가 수행됨

✓ 단순 대입 연산자의 동작 방식을 수정해야 함!(연산자 다중정의)

● 단순 대입 연산자를 정의하여 깊은 복사를 수행

```
#include <iostream>
using namespace std;
class CMyData {
public:
    CMyData() {
        cout << " CMyData() " << endl;
        m_pnData = new int;
        *m_pnData = 0;
    }
    CMyData(int nParam) {
        cout << " CMyData(int) " << endl;
        m_pnData = new int;
        *m_pnData = nParam;
    }
    // 단순 대입 연산자 함수를 정의한다.
    CMyData& operator=(const CMyData& rhs) {
        *m_pnData = *rhs.m_pnData;

        // 객체 자신에 대한 참조를 반환한다.
        return *this;
    }
    CMyData(const CMyData& rhs) {
        cout << " CMyData(const CMyData& rhs) " << 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() const {
        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);
    CMyData c;

    // 단순 대입을 시도하면 모든 멤버의 값을 그대로 복사한다.
    c = b;

    cout << a.GetData() << endl;
    cout << b.GetData() << endl;
    cout << c.GetData() << endl;

    return 0;
}
```




```
CMyData(int)
CMyData(const CMyData& rhs)
CMyData()
10
10
10
계속하려면 아무 키나 누르십시오 . . .
```