Chapter 5

Index

연산자 함수
산술 연산자
대입 연산자
    복합 대입 연산자
    이동 대입 연산자
배열 연산자
관계 연산자
단항 증감 연산자
문제

연산자 함수 : 연산자를 이용하듯 호출할 수 있는 메서드

연산자 다중 정의 : 필요에 따라 연산자 함수를 다중 정의하는 것

연산자 함수

- 연산자를 이용하듯 호출할 수 있는 메서드
- 간결하고 확장성이 높게 코드 작성 가능
- “절대로 논리 연산자들을 다중 정의해서는 안 된다”

산술 연산자

▼ **연산자 다중 정의와 이동 생성자 코드**

```
/* 연산자 다중 정의와 이동 생성자 */
#include "pch.h"
#include <iostream>
using namespace std;
```

```
class CMyData {
public:
    // 변환 생성자
    CMyData(int nParam) : m_nData(nParam) {
        cout << "CMyData(int)" << endl;
    }

    // 복사 생성자
    CMyData(const CMyData& rhs) : m_nData(rhs.m_nData)
    {
        cout << "CMyData(const CMyData &)" << endl;
    }

    // 이동 생성자
    CMyData(const CMyData&& rhs) : m_nData(rhs.m_nData)
    {
        cout << "CMyData(const CMyData &&)" << endl;
    }

    // 형변환
    operator int() { return m_nData; }

    // +
    CMyData operator+(const CMyData& rhs)
    {
        cout << "operator+" << endl;
        CMyData result(0);
        result.m_nData = this->m_nData + rhs.m_nData;

        return result;
    }

    // =
    CMyData& operator=(const CMyData& rhs)
    {
        cout << "operator=" << endl;
        m_nData = rhs.m_nData;

        return *this;
    }

private:
    int m_nData = 0;
};

int main()
{
    cout << "*****Begin*****" << endl;
    CMyData a(0), b(3), c(4);

    // b + c 연산을 실행하면 이름 없는 임시 객체가 만들어지며
    // a에 대입하는 것은 이 임시 객체다.
```

```
    a = b + c;
    cout << a << endl;
    cout << "*****End*****" << endl;

    return 0;
}
```

b+c는 임시객체를 만드는데 두변수의 값이 절대 달라지지 않아야 하기 때문이다.

“a = 임시 객체” 실행 → CMyData& operator=(const CMyData &rhs)라는 함수가 호출, rhs는 임시 객체의 참조, *this를 반환

“연산자 함수도 다중 정의 가능”

대입 연산자

▼ 대입 연산자 다중 정의 코드

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

// 제작자 코드
class CMyData
{
public:
    explicit CMyData(int nParam)
    {
        m_pnData = new int(nParam);
    }

    ~CMyData() { delete m_pnData; }

    operator int() { return *m_pnData; }

    // 단순 대입 연산자 다중 정의
    void operator=(const CMyData& rhs)
    {
        // 본래 가리키던 메모리를 삭제하고
        delete m_pnData;

        // 새로 할당한 메모리에 값을 저장한다.
        m_pnData = new int(*rhs.m_pnData);
    }

private:
    int* m_pnData = nullptr;
```

```
};

// 사용자 코드
int main()
{
    CMyData a(0), b(5);
    a = b;
    cout << a << endl;

    return 0;
}
```

만약 이 코드를 a=a;로 고쳤을시 delete부분 때문에 오류가 난다. → r-value가 자신이라면 대입을 수행하지 않게 한다, a = b = c를 하고 싶다면 반환 형식을 참조자로 설정하면 된다.

nullptr : NULL 포인터 (대체로 쓰는게 좋음)

복합 대입 연산자

▼ += 연산자 함수 추가 코드

```
/* += 연산자 함수 추가 */
#include "pch.h"
#include <iostream>
using namespace std;

// 제작자 코드
class CMyData
{
public:
    explicit CMyData(int nParam)
    {
        m_pnData = new int(nParam);
    }

    ~CMyData() { delete m_pnData; }

    operator int() { return *m_pnData; }

    // 단순 대입 연산자 다중 정의
    void operator=(const CMyData& rhs)
    {
        // 본래 가리키던 메모리를 삭제하고
        delete m_pnData;

        // 새로 할당한 메모리에 값을 저장한다.
        m_pnData = new int(*rhs.m_pnData);
```

```
    }

    CMyData& operator+=(const CMyData& rhs)
    {
        // 현재 값 처리
        int* pnNewData = new int(*m_pnData);

        // 누적할 값 처리
        *pnNewData += *rhs.m_pnData;

        // 기존 데이터를 삭제하고 새 메모리를 대체
        delete m_pnData;
        m_pnData = pnNewData;

        return *this;
    }
private:
    int* m_pnData = nullptr;
};

// 사용자 코드
int main()
{
    CMyData a(0), b(5), c(10);
    a += b;
    a += c;
    cout << a << endl;

    return 0;
}
```

이동 대입 연산자

▼ 이동 대입 연산자 코드

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

// 제작자 코드
class CMyData
{
public:
    explicit CMyData(int nParam)
    {
        cout << "CMyData(int)" << endl;
        m_pnData = new int(nParam);
    }
```

```
    CMyData(const CMyData& rhs)
    {
        cout << "CMyData(const CMyData &)" << endl;
        m_pnData = new int(*rhs.m_pnData);
    }

    ~CMyData() { delete m_pnData; }

    operator int() { return *m_pnData; }

    // 덧셈 연산자 다중 정의
    CMyData operator+(const CMyData& rhs)
    {
        // 호출자 함수에서 이름 없는 임시 객체가 생성된다.
        return CMyData(*m_pnData + *rhs.m_pnData);
    }

    // 단순 대입 연산자 다중 정의
    CMyData& operator=(const CMyData& rhs)
    {
        cout << "operator=" << endl;
        if (this == &rhs)
            return *this;

        delete m_pnData;
        m_pnData = new int(*rhs.m_pnData);

        return *this;
    }

    // 이동 대입 연산자 다중 정의
    CMyData& operator=(CMyData&& rhs)
    {
        cout << "operator=(Move)" << endl;

        // 얕은 복사를 수행하고 원본은 NULL로 초기화한다.
        m_pnData = rhs.m_pnData;
        rhs.m_pnData = NULL;

        return *this;
    }

private:
    int* m_pnData = nullptr;
};

// 사용자 코드
int main()
{
    CMyData a(0), b(3), c(4);
    cout << "*****Before*****" << endl;
```

```
    // 이동 대입 연산자가 실행된다!
    a = b + c;
    cout << "*****After*****" << endl;
    cout << a << endl;
    a = b;
    cout << a << endl;

    return 0;
}
```

a = b + c; 에서 임시객체를 구하고 r-value로 삼아 곧바로 단순 대입 연산을 실행한다면 이때는 이동 대입 연산자가 호출

이동시맨틱 : 이동 생성자 + 이동 대입 연산자로 구현

모두 임시 객체와 관련, 언제 어느 조건에서 호출되는지 정확히 알고 사용이 중요

배열 연산자

int& operator[ ] (int nIndex);

int operator[ ] (int nIndex) const;

일반적인 경우 첫 번째 배열 연산자 함수를 사용

상수형 메서드인 두 번째 선언은 상수형 참조를 통해서만 호출, 오로지 r-value로만 사용

▼ 배열 연산자 코드

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

// 제작자 코드
class CIntArray
{
public:
    CIntArray(int nSize)
    {
        // 전달된 개수만큼 int 자료를 담을 수 있는 메모리를 확보한다.
        m_pnData = new int[nSize];
        memset(m_pnData, 0, sizeof(int) * nSize);
```

```
    }

    ~CIntArray() { delete m_pnData; }

    // 상수형 참조인 경우의 배열 연산자
    int operator[](int nIndex) const
    {
        cout << "operator[] const" << endl;
        return m_pnData[nIndex];
    }

    // 일반적인 배열 연산자
    int& operator[](int nIndex)
    {
        cout << "operator[]" << endl;
        return m_pnData[nIndex];
    }

private:
    // 배열 메모리
    int* m_pnData;

    // 배열 요소의 개수
    int m_nSize;
};

// 사용자 코드
void TestFunc(const CIntArray& arParam)
{
    cout << "TestFunc()" << endl;

    // 상수형 참조이므로 'operator[](int nIndex) const'를 호출한다.
    cout << arParam[3] << endl;
}

int main()
{
    CIntArray arr(5);
    for (int i = 0; i < 5; i++)
        arr[i] = i * 10;

    TestFunc(arr);

    return 0;
}
```

TestFunc가 const로 호출했기 때문에 int operator .. const가 적용된다.

관계 연산자

- 상등 및 부등 연산자와 비교 연산자를 합친 것

int 클래스이름::operator==(const 클래스이름 &);

int 클래스이름::operator !=(const 클래스이름 &);

단항 증감 연산자

++, --

int operator++() : 전위식　　int operator++(int) : 후위식

▼ **단항 증감 연산자 예 코드**

```
/* 단항 증가 연산자 예 */
#include "pch.h"
#include <iostream>
using namespace std;

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

    // 형변환
    operator int() { return m_nData; }

    // 전위 증가 연산자
    int operator++()
    {
        cout << "operator++()" << endl;
        return ++m_nData;
    }

    // 후위 증가 연산자
    int operator++(int)
    {
        cout << "operator++(int)" << endl;
        int nData = m_nData;
        m_nData++;

        return nData;
    }

private:
    int m_nData = 0;
```

```
};

int main()
{
    CMyData a(10);

    // 전위 증가 연산자를 호출한다.
    cout << ++a << endl;

    // 후위 증가 연산자를 호출한다.
    cout << a++ << endl;
    cout << a << endl;

    return 0;
}
```

후위 증가식 : 미리 값을 백업후 증감후 백업한 값을 반환

문제

1. **대입 연산자 오버로딩 시 주의해야 할 점 (두가지), 이유**

a=a ; - delete ， a=b=c ; - 반환형식

2. **각종 대입 연산자들의 적절한 반환 형식**

참조자

3. **후위식 단항 증가 연산자 함수 원형**

int operator++ (int)