21. 연산자 함수

- '연산자 함수'는 연산자를 이용하듯 호출할 수 있는 메서드
- 클래스를 만들 때 지원 가능한 연산자를 만들어서 넣어주면 사용자 코드가 간결해짐, 확장성 높아짐!
- 절대로 논리 연산자들을 다중정의 해서는 안된다! < 명심할 것!!(심각한 오류를 사용자에게 넘기는 꼴)

22. 산술 연산자

- int 같은 기본 자료형에서는 산술연산은 int 자료형 변수 x에 대해 x+3 같은 연산 실행하는 것
- 우리가 만든 클래스인 CTestData 의 인스턴스 data에 data+3 같은 연산은 실행하면 오류로 인식
- 연산자 다중정의로 + 의 의미를 어떻게 정의하는가에 달려있음.

```
#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& rhs)" << endl;
    }
    // 이동 생성자
    CMyData(const CMyData&& rhs) :m_nData(rhs.m_nData) {
        cout << " CMyData(const CMyData&& rhs)" << endl;
    }
    // + 덧셈 연산자
    CMyData operator+(const CMyData& rhs) {
        cout << " operator+ " << endl;
        CMyData result(0);
        result.m_nData = this->m_nData + rhs.m_nData;
        return result;
    }
    // - 뺄셈 연산자
    CMyData operator-(const CMyData& rhs) {
        cout << " operator- " << endl;
        CMyData result(0);
        result.m_nData = this->m_nData - rhs.m_nData;
        return result;
    }
    int GetData() { return m_nData; }
private:
    int m_nData = 0;
};
```

```
// 사용자 코드
int main()
{
    cout << "***** Begin ***** " << endl;
    CMyData a(2), b(3), c(4);

    CMyData d(a + b);
    cout << "d : " << d.GetData() << endl;

    CMyData e(b - c);
    cout << "e : " << e.GetData() << endl;

    return 0;
}
```

- 실행결과

```
***** Begin *****
 CMyData(int)
 CMyData(int)
 CMyData(int)
 operator+
 CMyData(int)
 CMyData(const CMyData&& rhs)    // a+b 는 이름없는 임시객체 생성
d : 5
 operator-
 CMyData(int)
 CMyData(const CMyData&& rhs)    // b-c 는 이름없는 임시객체 생성
e : -1
계속하려면 아무 키나 누르십시오 . . .
```

- *, / 같은 산술 연산자들을 다중정의 하는 방법은 +, - 연산자와 크게 다르지 않음
- 연산자 함수도 다중 정의할 수 있다는 사실!

23. 대입 연산자

- 단순 대입 연산자 함수는 Deep copy를 배울 때 이미 경험한 바 있음.
- 대입 연산자를 다중정의해야 하는 이유를 꼭 알아야 함!
- 기술적인 측면에서 추가로 고려해야 할 점 확인!

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

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

    // 단순 대입 연산자 다중정의
    void operator=(const CMyData& rhs)
    {
        // 본래 가리키던 메모리를 삭제하고
        delete m_pnData;
        // 새로 할당한 메모리에 값을 저장한다.
        m_pnData = new int(*rhs.m_pnData);
    }
    int GetData() { return *m_pnData; }
private:
    int* m_pnData = nullptr;
};
// 사용자 코드
int main()
{
    CMyData a(0), b(5);
    a = b;
    cout << a.GetData() << endl;
    return 0;
}
```

- 실행결과

```
5
계속하려면 아무 키나 누르십시오 . . .
```

- 위 코드에서 m_pnData = new int(*rhs.m_pnData);는 다음과 같이 두줄로 변경 가능

```
m_pnData = new int;
*m_pnData = *rhs.m_pnData;
```

- 사용자 코드를 다음과 같이 변경하면 심각한 논리적 오류를 경험

```
// 사용자 코드
int main()
{
    CMyData a(0), b(5);

    // 정상적인 코드라고 하기는 어려우나 문법적으로는 문제가 없다.
    a = a;
    cout << a.GetData() << endl;
    return 0;
}
```

- 실행결과

```
-572662307
계속하려면 아무 키나 누르십시오 . . .
```

- 실행결과를 보면 프로그램이 아예 죽어버린 것은 아니지만 쓰레기 값을 출력
- 사용자가 a = a; 같은 코드를 만든다 하더라도 정상적으로 작동하도록 제작해야함
- operator=()에서 데이터를 복사하기도 전에 원본을 삭제(delete m_pnData;) 했기 때문임.
- 단순 대입 연산자 내부에서 내가 나 자신에게 단순 대입 연산을 실행하는 경우에 아무런 처리 하지 않는 방법으로 대응

```
    // 단순 대입 연산자 다중정의
    void operator=(const CMyData& rhs)
    {
        // r-value가 자신이면 대입을 수행하지 않는다.
        if (this == &rhs)
            return;

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

- 대입 연산자의 반환형식이 void 가 되더라도 a = b; 와 같은 연산은 문제가 없음.
- a = b = c; 같은 연산은 바로 문제가 발생함.

```
// 사용자 코드
int main()
{
    CMyData a(0), b(5), c(10);

    // 정상적인 단순 대입 연산이다.
    a = b = c;
    cout << a.GetData() << endl;
    return 0;
}
```

```
error C2679: 이항 '=': 오른쪽 피연산자로 'void' 형식을 사용하는 연산자가 없거나 허용되는 변환이 없습니다.
```

● 대입 연산자 함수의 반환형식을 참조자로 설정해서 문제를 해결

```
    // 단순 대입 연산자 다중정의
    CMyData& operator=(const CMyData& rhs)
    {
        // r-value가 자신이면 대입을 수행하지 않는다.
        if (this == &rhs)
            return *this;

        delete m_pnData;
        m_pnData = new int(*rhs.m_pnData);
        return *this;
    }
```

■ 복합 대입연산자

● 위의 대입연산자 CMyData 클래스에 += -= 연산자 함수를 추가한 예

```
    CMyData& operator+=(const CMyData& rhs)
    {
        // 현재 값 처리
        int* pnNewData = new int(*m_pnData);
        // 누적할 값처리
        *pnNewData += *rhs.m_pnData;
        // 기존 데이터를 삭제하고 새 메모리로 대체
        delete m_pnData;
        m_pnData = pnNewData;
        return *this;
    }
    CMyData& operator-=(const CMyData& rhs)
    {
        // 현재 값 처리
        int* pnNewData = new int(*m_pnData);
        // 누적할 값처리
        *pnNewData -= *rhs.m_pnData;
        // 기존 데이터를 삭제하고 새 메모리로 대체
        delete m_pnData;
        m_pnData = pnNewData;
        return *this;
    }
```

```
// 사용자 코드
int main()
{
    CMyData a(0), b(5), c(10);
    a += b;
    b -= c;
    cout << a.GetData() << endl;
    cout << b.GetData() << endl;
    return 0;
}
```

실행결과 :

```
5
-5
```

■ 이동 대입연산자

- 연산에 의한 임시결과는 두 종류가 있음
- 첫 번째로는 a + b 같은 연산에 의한 것이고 두 번째는 함수 호출에 의한 것임.
- 임시 객체 때문에 이동생성자가 생겨났듯이 또 임시 객체 때문에 대입연산자와 별개로 이동 대입 연산자도 생겨남

```
#include <iostream>
using namespace std;
// 제작자 코드
class CMyData
{
public:
    explicit CMyData(int nParam) {
        m_pnData = new int(nParam);
    }
    CMyData(const CMyData& rhs) {
        cout << " CMyData(const CMyData& rhs)" << endl;
        m_pnData = new int(*rhs.m_pnData);
    }
    ~CMyData() { delete m_pnData; }
    // 덧셈 연산자 다중 정의
    CMyData operator+(const CMyData& rhs) {
        // 호출자 함수에서 이름 없는 임시 개게가 생성된다.
        return CMyData(*m_pnData + *rhs.m_pnData);
    }
    // 단순 대입 연산자 다중정의
    CMyData& operator=(const CMyData& rhs) {
        cout << " operator=(const CMyData& rhs) " << endl;
        if (this == &rhs)
            return *this;
        delete m_pnData;
        m_pnData = new int(*rhs.m_pnData);
        return *this;
    }
    // 이동 대입 연산자 다중정의
    CMyData& operator=(CMyData&& rhs) {
        cout << " CMyData& operator=(CMyData&& rhs) " << endl;

        // Shallow copy 를 수행하고 원본은 NULL로 초기화 한다.
        m_pnData = rhs.m_pnData;
        rhs.m_pnData = NULL;
        return *this;
    }
    int GetData() { return *m_pnData; }
private:
    int* m_pnData = nullptr;
};
```

```
// 사용자 코드
int main()
{
    CMyData a(2), b(3), c(4);
    cout << "***** Before ***** " << endl;
    //  이동 대입 연산자가 실행된다.
    a = b + c;
    cout << "***** After ***** " << endl;
    cout << " a : " << a.GetData() << endl;
    a = b;
    cout << " a : " << a.GetData() << endl;
    return 0;
}
```

- 실행결과

```
***** Before *****
 CMyData& operator=(CMyData&& rhs)
***** After *****
 a : 7
 operator=(const CMyData& rhs)
 a : 3
계속하려면 아무 키나 누르십시오 . . .
```

- main() 함수의 a = b + c; 와 a = b;를 비교해서 봐야함
- CMyData 클래스이 인스턴스인 b와 c를 더하면 당연히 CMyData::operator+() 함수가 호출되고 부산물로 임시객체가 생김
- 임시객체를 r-value 삼아 곧바로 단순 대입 연산을 실행한다면 이때는 이동대입 연산자가 호출됨.
- 이동 대입 연산자를 다중정의하지 않는다면 일반 대입 연산자 함수가 호출됨. 하지만 존재한다면 컴파일러가 이동 대입 연산자를 호출해줌
- 이동 생성자와 마찬가지로 이동 대입 연산자도 곧 사라질 임시 객체와 직접적인 관련이 있음.
- 일반 대입 연산자함수는 Deep Copy를 수행하지만 이동 대입 연산자는 Shallow Copy를 수행함.
- C++11에 새로 추가된 이동 시맨틱은 이동 생성자와 이동 대입 연산자로 구현됨.
- 언제 어느조건에서 호출되는지 정확히 알고 사용하는 것이 가장 중요함!

24. 배열 연산자

- 배열 연산자도 다중정의할 수 있음.
- 포인터 및 메모리 동적할당을 수행하면서도 사용자에게 배열을 다루는 것처럼 편리함을 제공
- 배열 연산자 함수의 매개변수는 int 자료형 변수 하나뿐인데, 이것은 배열의 인덱스로 활용됨.
- '상수형 참조로 접근'을 포함해서 두 가지 형태로 정의할 수 있음.

```
int& operator[](int nIndex);
int operator[](int nIndex) const;
```

- 첫 번째 선언에서 int&를 반환하는데 이는 l-value로 사용되는 경우를 고려해야하기 때문
- 일반적인 경우에서는 배열 연산이 l-value가 되든 r-value가 되든 첫 번째 배열 연산자 함수가 사용됨
- 상수형 메서드인 두 번째 선언은 상수형 참조로만 호출할 수 있고 오로지 r-value로만 사용됨.

```
#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[](int nIndex) const " << endl;
        return m_pnData[nIndex];
    }
    // 일반적인 배열 연산자
    int& operator[](int nIndex)
    {
        cout << " operator[](int nIndex) " << endl;
        return m_pnData[nIndex];
    }
    int GetData() { return *m_pnData; }
private:
    // 배열 메모리
    int* m_pnData = nullptr;
    // 배열 요수의 개수
    int m_nSize = 0;
};
```

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

    // 상수형 참조이므로 'operator[](int nIndex) const'를 호출한다.
    cout << " 상수형 참조 : " << endl;
    cout << arParam[3] << endl;
}
int main()
{
    CIntArray arr(5);
    for (int i = 0; i < 5; i++)
        arr[i] = i * 10;

    TestFunc(arr);

    return 0;
}
```

- 실행결과

```
operator[](int nIndex)
operator[](int nIndex)
operator[](int nIndex)
operator[](int nIndex)
operator[](int nIndex)
TestFunc(const CIntArray&)
상수형 참조 :
operator[](int nIndex) const
30
계속하려면 아무 키나 누르십시오 . . .
```

- arr[i] = i * 10;을 실행하는데, 여기서 수행하는 배열 연산자 함수는 int& operator[](int nIndex) 이고, 반환형식이 참조자 이므로 l-value가 될 수 있음.
- TestFunc() 함수에서는 매개변수 형식이 상수형 참조인 const CIntArray &arParam 이므로 int& operator[](int nIndex) 가 아니라 int operator[](int nIndex) const 가 적용됨.
- 두 배열 연산자 함수 모두 '경계 검사'를 하지 않음.
- 그러므로 요소가 5개인 배열에 대해서 –1 이나 5 이상의 인덱스를 부여한다면 분명 오류가 발생할 것임.

25. 관계 연산자

- 상등 및 부등연산자와 비교연산자를 합쳐 관계연산자라 함.
- 관계연산자들의 연산결과로 생성되는 값의 형식은 int 자료형임(참이면 1, 거짓이면 0)
- 상등 및 부등 관계 연산자 함수의 원형

```
int operator==(const 클래스이름& rhs);
int operator!=(const 클래스이름& rhs);
```

- 배열단위비교의 가장 흔한예로 문자열 비교가 있음(두 문자열이 같은지, 다른지 비교해야하는 경우)
- CMyString 클래스로 확인해봄.

26. 단항 증감 연산자

- 단항 증감 연산자는 for문에서 반복자(혹은 카운터)로 사용되는 변수에서 흔히 볼 수 있음
- 단항 증가 연산자는 ++ 이고 감소연산자는 – 임
- 이 연산자가 피 연산자 왼쪽에 붙으면 '전위식' 이고 오른쪽에 붙으면 '후위식'이 됨
- 단항 증감 연산자를 구현할 때 가장 주의해야 할 부분이 바로 전위식과 후위식임.

```
int operator++();
int operator++(int);
```

- 매개변수 없이 int operator++() 라고 추가하면 이는 전위식에 해당함.
- int 자료형 인수를 하나 받도록 int operator++(int) 라고 하면 후위식이 됨.
- 단항 증감 연산자 예

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

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

    // 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;
    }
    // 전위 감소 연산자
    int operator--()
    {
        cout << " operator--()" << endl;
        return --m_nData;
    }
    // 후위 감소 연산자
    int operator--(int)
    {
        cout << " operator--(int)" << endl;
        int nData = m_nData;
        --m_nData;
        return nData;
    }
    int GetData() const { return m_nData; }
private:
    int m_nData = 0;
};
```

```
int main()
{
    CMyData a(10), b(5);

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

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

    // 전위 감소 연산자를 호출한다.
    --b;
    cout << b.GetData() << endl;

    // 후위 감소 연산자를 호출한다.
    b--;
    cout << b.GetData() << endl;
    return 0;
}
```

- 실행결과

```
operator++()
11
operator++(int)
12
operator--()
4
operator--(int)
3
```

- 전위식 단항 증감 연산자 함수는 내부적으로 증가, 감소 시켜야 할 멤버변수를 증감시켜 반환함
- 후위식 증감 연산자 함수는 멤버 변수의 값을 증감시키기 전에 먼저 백업(int nData=m_nData;)하고 이어서 증감 연산을 실행.
- 마지막에는 증감된 값이 아닌 앞서 백업한 값을 반환(return nData) 함으로써 후위식에 의한 단항 증감을 구현