



[이것이 C++ 이다]

Chapter 5  
연산자 다중 정의

## 5장의 핵심 개념

### 연산자 함수, 연산자 다중 정의

- 연산자 함수

: 연산자를 이용하듯 호출할 수 있는 메서드. 사용자 코드에 보이는 연산자(예를 들어 '+' 연산자)가 실제로는 함수이고 사용자가 직접 그 의미를 구현하는 문법이다.

- 연산자 다중 정의

: 필요에 따라 연산자 함수를 다중 정의하는 것.

## 산술 연산자

### 반환형 operator 연산자 ( 매개변수 );

```
class CMyData
{
public:
    ...
    // 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;
    }
}
```

*Handwritten notes:*  
- Under 'return result;': 'return' with an arrow pointing to 'result' and 'return' written below.  
- To the right: 'return' written twice, once above 'result' and once below 'result'.

## 산술 연산자

반환형 operator 연산자 ( 매개변수 );

```
// =
CMyData& operator=(const CMyData &rhs)
{
    cout << "operator=" << endl;
    m_nData = rhs.m_nData;
    return *this;
}
private:
    int m_nData = 0;
};
```

## 산술 연산자

연산자 함수가 제공되는 클래스는 높은 추상성을 제공하며 쉽게 사용할 수 있다.

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

$a = b + c$

$a = b.operator(c)$

$a.operator = (b.operator + c)$

## 대입 연산자

대입 연산자는 복사 생성자처럼 깊은 복사, 얕은 복사 문제가 있다.

```
class CMyData
{
public:
    ...
    void operator=(const CMyData &rhs)
    {
        // 본래 가리키던 메모리를 삭제하고
        delete m_pnData;
        // 새로 할당된 메모리에 값을 저장한다.
        m_pnData = new int(*rhs.m_pnData);
    }
private:
    int *m_pnData = nullptr;
};

int _tmain(int argc, _TCHAR* argv[])
{
    CMyData a(0), b(5);
    a = b;
    cout << a << endl;
    return 0;
}
```



## 대입 연산자

$a = a;$  와 같은 코드나  $a = b = c;$  같은 코드도 고려해야 한다.

```
class CMyData
{
public:
    ...
    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;
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    CMyData a(0), b(3), c(4);
    a = b = c;
    return 0;
}
```

## 이동 연산자 (이동 시맨틱)

임시 객체가 r-value인 단순 대입 연산을 고려해야 한다.

```
class CMyData
{
public:
    ...
    CMyData& operator=(CMyData &&rhs)
    {
        cout << "operator = (Move)" << endl;
        // 얇은 복사를 수행하고 원본은 NULL로 초기화한다.
        m_pnData = rhs.m_pnData;
        rhs.m_pnData = NULL;
        return *this;
    }
    ...
};
int _tmain(int argc, _TCHAR* argv[])
{
    CMyData a(0), b(3), c(4);
    a = b + c;
    return 0;
}
```

## 배열 연산자

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

```
class CIntArray
{
public:
    ...
    // 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];
    }
    ...
};
```

## 배열 연산자

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

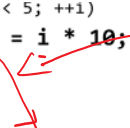
```
// 사용자 코드
void TestFunc(const CIntArray &arParams)
{
    cout << "TestFunc()" << endl;
    // 상수형 참조이므로 'operator[] (int nIndex) const'를 호출한다.
    cout << arParams[3] << endl;
}

int _tmain(int argc, _TCHAR* argv[])
{
    CIntArray arr(5);
    for (int i = 0; i < 5; ++i)
        arr[i] = i * 10;
    TestFunc(arr);
    return 0;
}
```

```

for (int i = 0; i < 5; ++i)
    arr[i] = i * 10;
TestFunc(arr);
return 0;
}

```



## 관계, 단항 증/감 연산자

다음과 같은 형태로 정의 될 수 있다.

- int 클래스이름::operator==(const 클래스이름 &rhs);
- int 클래스이름::operator!=(const 클래스이름 &rhs);
- int operator++() //전위식
- int operator++(int) //후위식

## 단항 증/감 연산자

후위 연산은 현재 값을 따로 백업 해둔 후 반환한다.

```

class CMyData
{
public:
    ...
    // 전위 증가 연산자
    int operator++()
    {
        return ++m_nData;
    }
    // 후위 증가 연산자
    int operator++(int)
    {
        int nData = m_nData;
        m_nData++;
        return nData;
    }
private:
    int m_nData = 0;
};

```