C++ 강좌 09회 : 연산자 오버로딩

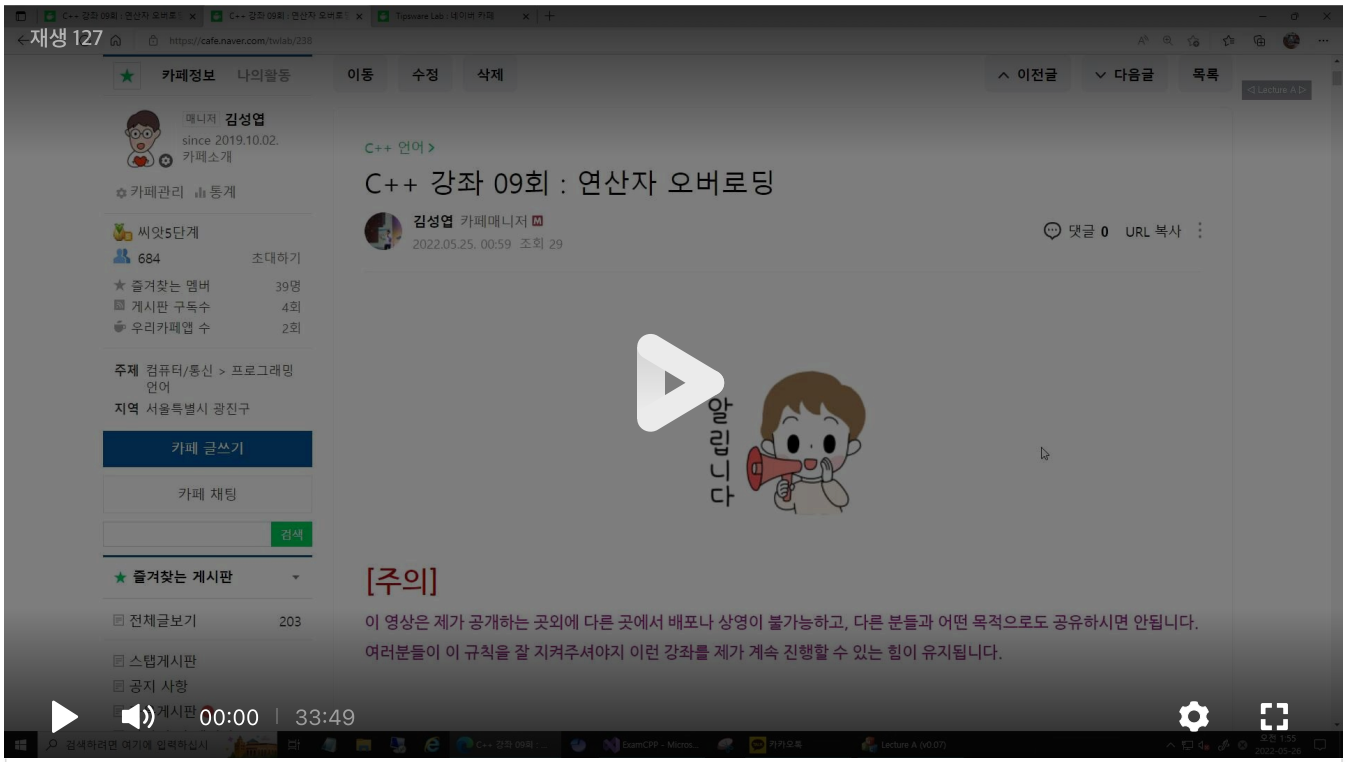
김성엽 카페매니저 + 구독 1:1 채팅
2022.05.25. 00:59 조회 556

[주의]

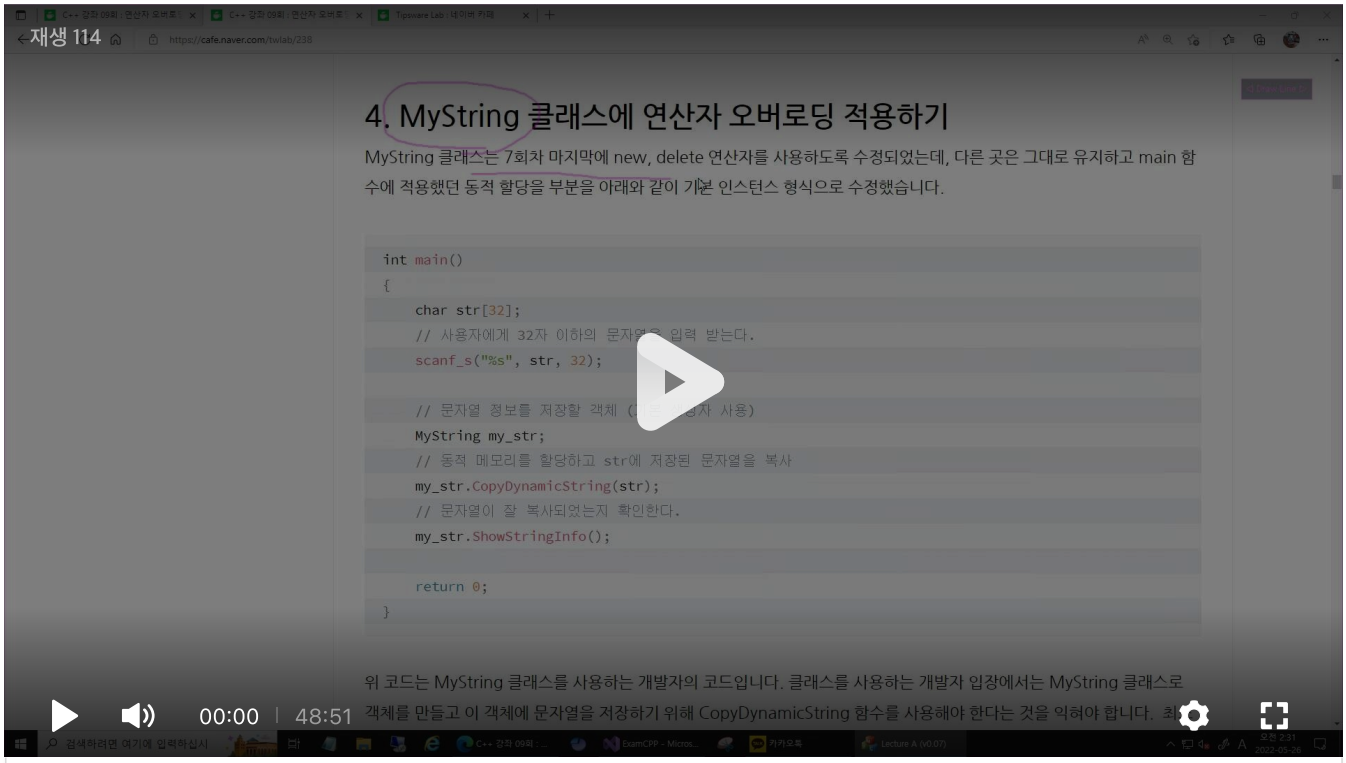
이 영상은 제가 공개하는 곳외에 다른 곳에서 배포나 상영이 불가능하고, 다른 분들과 어떤 목적으로도 공유하시면 안됩니다. 여러분들이 이 규칙은 잘 지켜주셔야지 이런 강좌를 제가 계속 진행할 수 있는 힘이 유지됩니다.

이번 강좌에서는 연산자를 함수 이름처럼 사용하는 연산자 오버로딩(operator overloading) 문법에 대해 소개하겠습니다. '오버로딩'과 관련된 기본 개념은 8회차 강좌에 소개되어 있으니 8회차 강좌를 보지 않았다면 아래에 링크한 8회차 강좌를 먼저 보고 이 강좌를 보기 바랍니다.

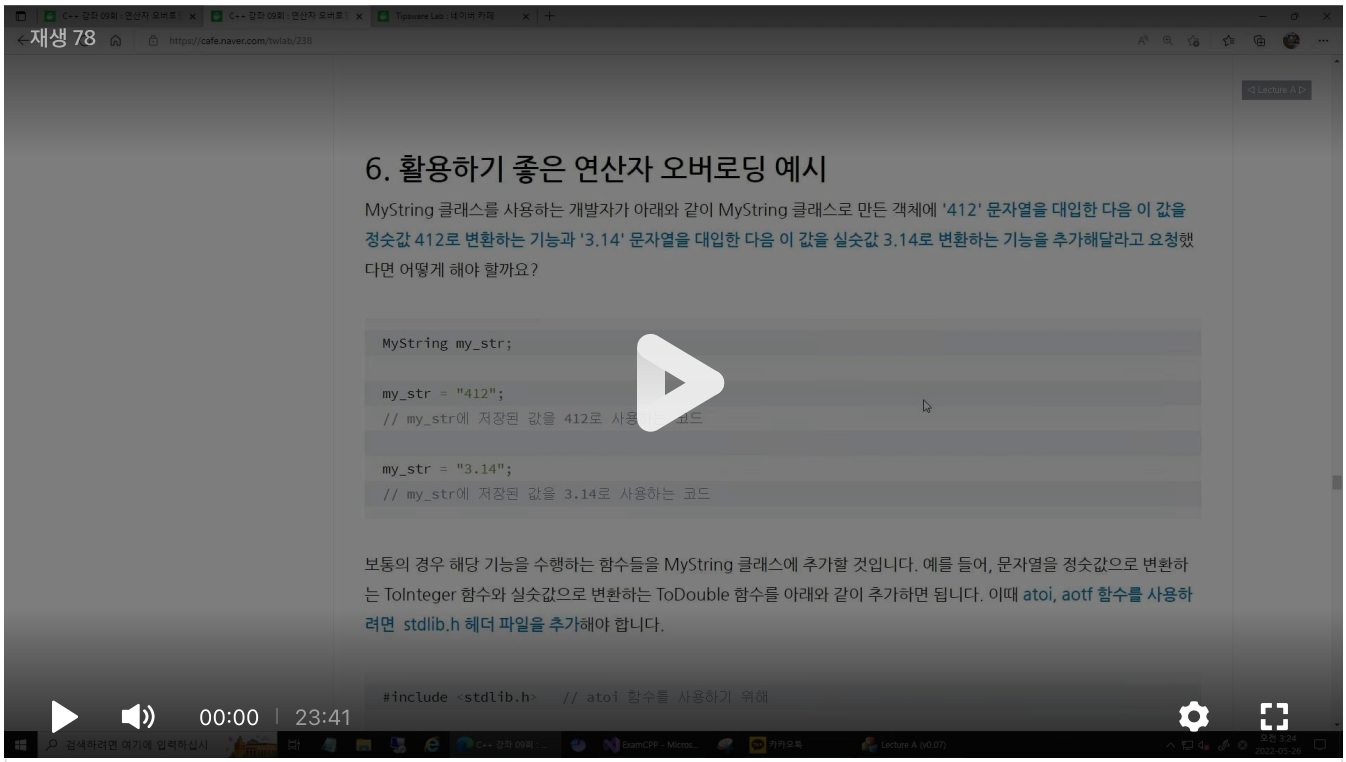
C++ 강좌 08회 : 함수 오버로딩
대한민국 모임의 시작, 네이버 카페
cafe.naver.com



C++ 강좌 9회 : 연산자 오버로딩 - 1



C++ 강좌 9회 : 연산자 오버로딩 - 2



C++ 강좌 9회 : 연산자 오버로딩 - 3

1. C 언어에서 두 가지 이상의 의미를 가진 연산자는 없었는가?

C 언어에서 연산자는 명령문을 구성할 때 동사의 역할을 합니다. 그래서 C 언어의 **명령문이 기계어로 번역될 때 연산자는 기계어 명령을 결정하는 기준**이 됩니다. 예를 들어, '+' 연산자를 사용하면 기계어로 번역될 때 ADD 명령어로 번역되고 '-' 연산자를 사용하면 기계어로 번역될 때 SUB 명령어로 번역됩니다. 따라서 C 언어에서 **연산자는 고유한 의미로 사용되는 것이 원칙**입니다.

실제로 C 언어는 연산자가 고유한 의미로 사용되게 하려고 수학에서 배웠던 연산자의 의미를 새로운 연산자를 추가해서 두 가지로 나누기도 했습니다. 예를 들어, '=' 연산 기호는 수학에서 대입하다(assign)는 의미도 있지만 동일하다(equal)는 의미도 가지고 있기 때문에 '오버로드'된 의미를 가진 연산자입니다. 따라서 **C 언어는 '=' 연산 기호를 연산자로 만들면서 의미를 한 가지로만 사용하게 하려고 '=' 연산자를 '대입하다'는 의미로만 사용하고, '동일하다'는 의미는 '==' 연산자를 추가해서 사용**했습니다.

하지만 **키보드에 제공되는 기호의 개수가 너무 적어**서 C 언어는 기본 연산자를 구성하기도 어려웠습니다. 왜냐하면 '==' 또는 '+='처럼 연산 기호를 조합해서 표현하면 가능은 했겠지만, 이런 표현이 많아지면 소스의 가독성이 떨어지고 개발자가 실수할 확률이 높아집니다. 그래서 **C 언어의 연산자는 약간의 오버로드 개념이 사용**됩니다.

예를 들어, **'*' 연산자의 경우 '곱셈 연산자'로 사용**되지만, **'번지 지정 연산자'로도 사용**이 됩니다. 아래의 예시처럼 '*' 연산자는 두 가지의 의미(역할)를 가지고 있습니다.

```
*p = 5;        // 번지 지정 연산자로 사용
num = 2 * 3;   // 곱셈 연산자로 사용
```

연산 수식에서 연산자는 '피연산자'를 요구하는데 피연산자가 한 개인 연산자를 '단항 연산자'라 하고 피연산자가 두 개인 연산자를 '이항 연산자'라 합니다. C 언어는 이런 연산자의 구성 형식을 체크해서 두 가지 의미를 가진 '*' 연산자의 의미를 구분합니다. 즉, C 언어 컴파일러는 수식에서 '*' 연산자가 한 개의 피연산자를 가진 형태이면 '번지 지정 연산자'로 번역하고 '*' 연산자가 두 개의 피연산자를 가진 형태이면 '곱셈 연산자'로 번역합니다.

> 단항 연산자 : 한 개의 연산자와 한 개의 피연산자로 구성
> 이항 연산자 : 한 개의 연산자와 두 개의 피연산자로 구성

```
*p = 5;       // * 연산자는 단항 연산자로 사용됨  :  번지 지정 연산자로 사용
num = 2 * 3;  // * 연산자는 이항 연산자로 사용됨  :  곱셈 연산자로 사용
```

결국 C 언어는 연산 기호의 부족으로 연산자에 오버로드 개념이 일부 적용되어 있습니다. 하지만 이것은 C 언어 문법이 자체적으로 사용하고 있는 것일뿐 개발자에게 오버로드 문법이 제공되지는 않았습니다.

2. 연산자 오버로딩

C 언어의 연산자는 피연산자로 정숫값, 실숫값 또는 주솟값을 사용합니다. 예를 들어, 2 + 3.14처럼 '+' 연산자는 피연산자로 정숫값 2와 실숫값 3.14를 사용합니다. a + b처럼 상수가 아닌 변수를 사용하는 경우도 있지만, 형식만 변수가 사용되었을 뿐 '+' 연산자가 사용하는 피연산자 값은 a 변수에 저장된 값과 b 변수에 저장된 값이기 때문에 상황은 동일합니다.

그리고 "abc" + 1처럼 사용한 경우에도 피연산자가 문자열인 것이 아니라 'abc' 문자열 상수가 저장된 메모리의 주솟값을 사용합니다. 따라서 '+' 연산자의 피연산자는 'abc' 문자열이 저장된 주솟값과 정숫값 1입니다. 만약, 문자열 상수를 사용하면 주솟값으로 처리되는 것에 대해 잘 모든다면 아래에 링크한 글을 참고하기 바랍니다.

결국 C 언어에서 연산자는 값을 사용해서 정해진 연산만 수행할 수 있을뿐 개발자가 다른 작업(행위)을 하도록 재정의 할 수 없습니다. 예를 들어, 피연산자가 두 개인 경우에 '+' 연산자는 두 값을 더하고 '*' 연산자는 두 값을 곱하고 '=' 연산자는 오른쪽에 있는 값을 왼쪽 변수에 대입하는 작업만 수행이 가능합니다.

하지만 C++ 언어는 연산자에 부여된 작업은 개발자가 재정의 할 수 있는데 이 작업은 '연산자 오버로딩'이라고 합니다. 연산자 오버로딩은 보통 객체에 작업하기 때문에 C++ 언어는 연산자와 함께 사용된 피연산자가 객체인 경우, 연산자의 고유 작업을 하기 전에 피연산자로 사용된 객체에 해당 연산자가 재정의 되어 있는지를 확인합니다. 그리고 객체에 해당 연산자가 재정의 되어 있다면 해당 객체가 제공하는 연산자를 사용해서 작업을 진행합니다.

예를 들어, my_str + 1 수식에서 '+' 연산자는 두 숫자를 더하는 연산자지만 피연산자로 사용된 my_str 객체에 '+' 연산자가 재정의 되어 있다면 덧셈 연산 대신에 my_str 객체의 '+' 연산자에 정의된 작업을 수행하게 됩니다.

3. 연산자 오버로딩의 예시

C++ 언어에서도 아래와 같이 '+' 연산자와 함께 사용된 피연산자가 일반 정숫값이라면 단순 덧셈 연산이 진행되어 sum 변수에 2가 저장됩니다.

```
int sum, a = 5, b = -3;
sum = a + b;    // '+' 연산자는 숫자를 더하는 연산자이다.
```

하지만 아래와 같이 MyABS 객체를 선언하고 '+' 연산자를 재정의하는 코드를 추가합니다. 연산자 오버로딩은 함수 이름을 연산자로 사용하는 것과 같은 개념이지만 단순히 함수이름을 연산자로 사용하면 +( )처럼 사용되기 때문에 문법적으로 구현이 불가능합니다. 그래서 C++ 언어는 연산자를 함수 이름으로 사용하기 위해 operator 키워드를 붙여서 operator+ ( ) 형태로 사용합니다.

그리고 연산자 오버로딩을 사용할 때는 연산자에 따라 형식이 다르기 때문에 사용할 연산자의 기본 연산 형식은 잘 고려해서 작업해야 합니다. MyABS 클래스에 사용된 '+' 연산자는 일반 덧셈처럼 두 개의 피연산자를 사용합니다. 첫 번째 피연산자는 멤버 변수(m_num)에 저장된 값을 사용할 것이라서 두 번째 피연산자로 사용할 값만 매개 변수(a_num)로 받습니다. 그리고 연산 결과를 정숫값으로 반환 할 것이기 때문에 int operator+ (int a_num) 형식으로 선언된 것입니다.

MyABS 클래스에 재정의한 '+' 연산자는 함께 사용된 피연산자가 음수면 양수로 변경해서 덧셈을 하도록 되어 있습니다. 따라서 main 함수에서 my_num + b 연산을 하게 되면 '+' 연산자는 단순 덧셈을 하기 전에 my_num 객체에 '+' 연산자가 재정의 되어 있는지 확인합니다. 그리고 my_num 객체에 '+' 연산자가 재정의 되어 있는 것이 확인되면 MyABS 클래스의 operator+ 함수가 호출되어 -2로 전달된 값을 2로 바꿔서 5와 덧셈하여 7을 반환합니다. 따라서 sum 변수에는 7이 저장되어 화면에 7이 출력됩니다.

```
#include <stdio.h>    // printf 함수를 사용하기 위해

// 절댓값 덧셈을 하기 위한 클래스
class MyABS
{
private:
    int m_num;  // 정숫값을 저장할 멤버 변수

public:
    MyABS(int a_num)  // 생성자
    {
        m_num = a_num;  // 지정한 값을 멤버 변수에 저장
    }

    int operator+ (int a_num)  // '+' 연산자 오버로딩
    {
        // a_num 변수에 저장된 값이 음수라면 양수로 변경한다.
        if (a_num < 0) a_num = -a_num;
        // 멤버 변수에 저장된 값과 전달된 값을 더해서 반환한다.
        return m_num + a_num;
    }
};

int main()
{
    MyABS my_num(5);  // 객체를 인스턴스 하면서 5를 저장한다.
    int b = -2; // 음수 값

    // my_num 객체에 '+' 연산자가 재정의 되어 있기 때문에 단순 덧셈을 하지 않고
    // my_num 객체의 '+' 연산자를 사용하여 연산을 진행하기 때문에 b 변수의 절댓값을
    // 구해서 덧셈한 결과를 반환한다. 따라서 sum에는 7이 저장된다.
    int sum = my_num + b;
    // 결과를 출력한다.
    printf("sum : %d\n", sum);
    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
sum : 7

이 창을 닫으려면 아무 키나 누르세요...
```

4. MyString 클래스에 연산자 오버로딩 적용하기

MyString 클래스는 7회차 마지막에 new, delete 연산자를 사용하도록 수정되었는데, 다른 곳은 그대로 유지하고 main 함수에 적용했던 동적 할당을 부분을 아래와 같이 기본 인스턴스 형식으로 수정했습니다.

```
int main()
{
    char str[32];
    // 사용자에게 32자 이하의 문자열을 입력 받는다.
    scanf_s("%s", str, 32);

    // 문자열 정보를 저장할 객체 (기본 생성자 사용)
    MyString my_str;
    // 동적 메모리를 할당하고 str에 저장된 문자열을 복사
    my_str.CopyDynamicString(str);
    // 문자열이 잘 복사되었는지 확인한다.
    my_str.ShowStringInfo();

    return 0;
}
```

위 코드는 MyString 클래스를 사용하는 개발자의 코드입니다. 클래스를 사용하는 개발자 입장에서는 MyString 클래스로 객체를 만들고 이 객체에 문자열을 저장하기 위해 CopyDynamicString 함수를 사용해야 한다는 것을 익혀야 합니다. 최소한 CopyDynamicString 함수 이름은 기억해야 하는데, 이런 함수 이름이 잘 외워지는 경우도 있지만 자신이 사용하는 이름 패턴과 다르면 잘 외워지지 않는 경우도 많습니다.

결국 함수 이름에 대한 스트레스는 함수를 만드는 개발자 뿐만 아니라 함수를 사용하는 쪽도 동일하게 느낀다는 뜻입니다. 그리고 CopyDynamicString 함수 이름은 CopyString로 줄여 사용해도 충분한 함수입니다. 이 함수 내부적으로 메모리를 동적으로 할당하든 정적으로 할당하든 그건 클래스 내부에서 알아서 할 문제라 사용하는 개발자가 이해 할 필요가 없기 때문입니다.

이처럼 함수의 이름은 기준에 따라 얼마든지 달라질수 있고, 개발자마다 함수의 이름을 짓는 스타일도 달라서 여러 명의 개발자가 제공하는 클래스를 함께 사용하는 개발자는 오히려 함수 이름을 정하는 개발자보다 스트레스가 더 심할수도 있습니다. 그래서 프로그래밍 언어를 만드는 학자들은 함수의 이름을 연산자로 사용하는 방법에 대해 연구하기 시작한 것입니다. 왜냐하면 단어나 문장은 사람마다 표현이 조금씩 다를 수 있지만 기호의 의미는 거의 일정하기 때문입니다.

값을 대입하는 행위는 '대입하다', '설정하다', '지정하다', 등 다양할 수 있지만 '=' 기호는 대부분의 개발자가 오른쪽에 있는 값을 왼쪽에 대입한다고 생각하기 때문입니다. 예를 들어, 함수 이름을 사용하는 my_str.CopyDynamicString(str); 같은 코드보다는 연산 기호를 사용하는 my_str = str; 형식의 코드가 사용하는 측면에서 볼때 이해하기 좋고, 함수 이름을 기억할 필요도 없기 때문에 사용하기도 편하다는 뜻입니다. 그래서 C++ 언어에서 함수의 이름은 연산자로 사용하려고 만든 기술이 연산자 오버로딩이라고 보면 됩니다.

my_str = str;은 기본적으로 사용이 불가능한 연산입니다. 왜냐하면 my_str은 객체 자체를 의미하는 대표 식별자이기 때문에 값을 직접적으로 대입하는 것이 허용되지 않습니다. 객체에 값을 대입하고 싶다면 객체에 포함된 멤버 변수를 직접 지정하거나 해당 멤버 변수에 값을 대입할 수 있게 도와주는 함수를 사용해야 합니다.

그리고 지금처럼 '=' 연산자를 사용해서 직접 대입하려면 MyString 클래스에서 멤버 변수인 mp_str의 접근 지정자를 public로 변경하고 my_str.mp_str = str; 이라고 하면 가능합니다. 하지만 str 표현은 &str[0] 표현이 생략된 형태라서 mp_str 변수에 str 배열의 시작 주소를 대입하겠다는 의미가 되어 버립니다. 즉, str 배열의 문자열을 복사하는 것이 아니라 str 배열의 주소가 직접 대입되어 의미가 완전히 달라집니다.

그래서 my_str = str; 표현을 가능하게 하려면 my_str 객체에 '=' 연산자를 이름으로 가지는 함수를 등록해야 합니다. 하지만 함수 이름으로 '=' 연산 기호만 사용하면 문법 표현이 되지 않기 때문에 아래와 같이 operator 키워드와 함께 적는 연산자 오버로딩 문법을 사용해야 합니다. char *operator= (const char *ap_str) 함수는 CopyDynamicString 함수와 작업 내용이 거의 비슷하기 때문에 CopyDynamicString 함수를 호출해서 메모리를 할당하고 문자열을 복사하게 했습니다.

```
class MyString
{
    // ... 다른 코드 생략 ...

    // 동적메모리를 할당해서 a_str에 전달된 문자열을 복사하는 함수
    int CopyDynamicString(const char a_str[])
    {
        // 이미 할당된 메모리가 있으면 제거한다.
        CleanUp();

        // 매개 변수로 전달된 문자열의 길이를 구한다.
        m_len = strlen(a_str) + 1;
        // 문자열을 저장하기 위해 동적 메모리를 할당한다.
        mp_str = new char[m_len];
        // 메모리가 정상적으로 할당되었다면 메모리에 전달된 문자열을 복사한다.
        if (mp_str != NULL) memcpy(mp_str, a_str, m_len);
        // 문자열의 길이를 반환한다.
        return m_len;
    }

    // '=' 연산자 오버로딩
    char *operator= (const char *ap_str)
    {
        // 전달된 문자열을 동적으로 메모리를 할당하여 복사한다.
        CopyDynamicString(ap_str);
        // 할당된 메모리를 반환한다.
        return mp_str;
    }

    // ... 다른 코드 생략 ...

};
```

위와 같이 MyString 클래스에 '=' 연산자를 재정의 한 함수를 추가하면 main 함수에 아래와 같이 사용했던 코드를

```
// 문자열 정보를 저장할 객체 (기본 생성자 사용)
MyString my_str;
// 동적 메모리를 할당하고 str에 저장된 문자열을 복사
my_str.CopyDynamicString(str);
```

아래와 같이 호출하는 것이 가능합니다. 하지만 operator=(str)이라고 호출하는 것이 CopyDynamicString(str)이라고 호출하는 것보다 더 편하다고 이야기 할 수 있을까요?

```
// 문자열 정보를 저장할 객체 (기본 생성자 사용)
MyString my_str;
// 동적 메모리를 할당하고 str에 저장된 문자열을 복사
my_str.operator=(str);
```

그래서 C++ 언어는 위 표현을 아래와 같이 사용할 수 있게 문법을 제공합니다. 이것은 **C++ 컴파일러가 연산자를 체크할 때 피연산자가 객체 인지 확인하고 객체라면 해당 객체에 연산자가 재정의 되어 있는지 확인해서 재정의된 연산자가 호출**되도록 해주기 때문에 가능한 것입니다. 예를 들어, my_str = str; 코드에서 컴파일러는 '=' 연산자의 피연산자인 my_str과 str을 체크하는데 str은 객체가 아니라서 무시하고 my_str은 객체이기 때문에 '=' 연산자가 재정의 되어 있는지 체크합니다. 그리고 **my_str 객체에 '=' 연산자가 재정의 되어 있다는 것이 확인되면 my_str = 코드를 my_str.operator= 으로 변경**합니다. 그리고 **'=' 연산자의 오른쪽에 있는 피연산자를 함수의 매개 변수로 넘겨줍니다.** 그래서 my_str = str; 코드는 C++ 컴파일러에 의해 my_str.operator=(str); 코드로 변경되어 처리됩니다.

```
// 문자열 정보를 저장할 객체 (기본 생성자 사용)
MyString my_str;
// 동적 메모리를 할당하고 str에 저장된 문자열을 복사
// 컴파일러가 아래의 코드를 my_str.operator=(str);로 변경한다.
my_str = str;
```

MyString 클래스를 사용하는 개발자 입장에서 my_str.CopyDynamicString(str);처럼 함수를 호출하기 위해 함수 이름을 외우는 것보다 my_str = str; 처럼 사용하는 것이 더 편할 것입니다. 따라서 초보자들에게 클래스를 제공할 때는 많이 사용될 함수은 연산자 오버로딩 작업을 해서 제공하는 것이 좋습니다.

이 작업을 실습할 수 있도록 전체 소스를 보여드리면 다음과 같습니다.

```
#include <stdio.h>    // printf, scanf_s 함수를 사용하기 위해
#include <string.h>   // strlen, memcpy 함수를 사용하기 위해

class MyString
{
private:
    int m_len;  // NULL 문자를 포함한 문자열의 길이를 저장할 변수
    char *mp_str;  // 문자열이 저장된 메모리의 주소를 기억할 변수

public:
    // 기본 생성자 : 객체를 초기화하기 위해 자동으로 호출되는 함수
    MyString()
    {
        m_len = 0;
        mp_str = NULL;
    }

    // 생성자 : 지정된 문자열을 사용해서 객체를 초기화하는 생성자 함수
    MyString(const char *ap_str)
    {
        mp_str = NULL;
        CopyDynamicString(ap_str);
    }

    // 파괴자 : 객체를 정리하기 위해 자동으로 호출되는 함수
    ~MyString()
    {
        CleanUp();
    }

    // 동적메모리를 할당해서 a_str에 전달된 문자열을 복사하는 함수
    int CopyDynamicString(const char a_str[])
    {
        // 이미 할당된 메모리가 있으면 제거한다.
        CleanUp();

        // 매개 변수로 전달된 문자열의 길이를 구한다.
        m_len = strlen(a_str) + 1;
        // 문자열을 저장하기 위해 동적 메모리를 할당한다.
        mp_str = new char[m_len];
        // 메모리가 정상적으로 할당되었다면 메모리에 전달된 문자열을 복사한다.
        if (mp_str != NULL) memcpy(mp_str, a_str, m_len);
        // 문자열의 길이를 반환한다.
        return m_len;
    }

    // '=' 연산자 오버로딩
    char *operator= (const char *ap_str)
    {
        // 전달된 문자열을 동적으로 메모리를 할당하여 복사한다.
        CopyDynamicString(ap_str);
        // 할당된 메모리를 반환한다.
```

```
        return mp_str;
    }

    // 이 객체가 관리하는 문자열 정보를 출력하는 함수
    void ShowStringInfo()
    {
        // 관리하는 문자열의 정보를 출력한다.
        if (mp_str != NULL) printf("%s : %d\n", mp_str, m_len);
        else printf("보관된 문자열이 없습니다.\n");
    }

    // 이 객체가 관리하던 정보를 정리하는 함수
    void CleanUp()
    {
        // 할당된 메모리가 있다면 해제한다.
        if (mp_str != NULL) {
            delete[] mp_str;
            mp_str = NULL;
        }
    }
};

int main()
{
    char str[32];
    // 사용자에게 32자 이하의 문자열을 입력 받는다.
    scanf_s("%s", str, 32);

    // 문자열 정보를 저장할 객체 (기본 생성자 사용)
    MyString my_str;
    // 동적 메모리를 할당하고 str에 저장된 문자열을 복사
    // my_str.CopyDynamicString(str);  // 기존 코드
    my_str = str;
    // 문자열이 잘 복사되었는지 확인한다.
    my_str.ShowStringInfo();

    return 0;
}
```

위 예제는 다음과 같이 출력됩니다.

```
tipsware
tipsware : 9

이 창을 닫으려면 아무 키나 누르세요...
```

5. MyString 클래스에 문자열은 덧붙이는 연산자 오버로딩 추가하기

문자열을 덧붙인다는 뜻은 'tips'라는 문자열이 저장되어 있을 때 해당 문자열 뒤에 'ware' 문자열을 추가해서 'tipsware' 문자열로 만드는 것을 말합니다.

연산자 오버로딩을 좀더 실습하기 위해 MyString 클래스에 저장된 문자열에 다른 문자열을 덧붙이는 함수를 추가하고 이 함수를 연산자 오버로딩으로 사용할 수 있게 해보겠습니다. 일단 MyString 클래스에 문자열을 덧붙이는 AppendString 함수를 아래와 같이 추가했습니다.

현재 mp_str에 할당된 메모리는 문자열의 길이만큼만 할당되어 있어 있기 때문에 다른 문자열을 추가할 빈 공간이 없습니다. 따라서 ap_str에 전달된 문자열을 덧붙이려면 해당 문자열의 길이만큼 추가로 메모리를 할당해야 합니다. 그래서 기존 문자열의 길이와 덧붙일 문자열의 길이를 더해서 새로운 문자열을 할당했습니다. 그리고 새로 할당된 메모리에 기존 문자열과 덧붙일 문자열을 순차적으로 복사하고 기존 문자열을 저장하던 메모리를 제거하고 새로 할당된 메모리의 주소를 mp_str에 대입하는 방법으로 구현했습니다.

```
class MyString
{
    // ... 다른 코드 생략 ...

    // 저장된 문자열에 다른 문자열을 추가하는 경우에 사용한다.
    // 예를 들어, "abc"가 저장되어 있는데 AppendString("def");라고 하면 "abcdef"가 됩니다.
    void AppendString(const char *ap_str)
    {
        // 전달된 문자열의 길이를 구하고 합쳐졌을때의 길이를 구한다.
        int len = strlen(ap_str), total_len = m_len + len;
        // 합쳐진 문자열을 저장할 메모리를 할당한다.
        char *p_str = new char[total_len];
        // NULL 문자 제외하고 처리하기 위해 1만큼 줄인다.
        m_len--;
        // 새로 할당된 메모리에 기존 문자열을 복사한다.
        memcpy(p_str, mp_str, m_len);
        // 전달된 문자열을 뒤에 추가한다.
        memcpy(p_str + m_len, ap_str, len + 1);
        // 기존 문자열을 제거한다.
        delete[] mp_str;
        // 새로 할당한 메모리를 멤버 변수에 저장한다.
        mp_str = p_str;
        // 문자열 길이도 합쳐진 길이를 저장한다.
        m_len = total_len;
    }

    // ... 다른 코드 생략 ...

};
```

그리고 위 함수를 테스트하기 위해 main 함수에 AppendString 함수를 사용하는 코드를 아래와 같이 추가했습니다. 이렇게 하면 사용자가 'tipssoft'라고 입력하면 이 문자열 뒤에 ' by tipsware'가 추가되어 my_str 객체에는 'tipssoft by

tipsware' 문자열이 저장됩니다.

```
int main()
{
    char str[32];
    // 사용자에게 32자 이하의 문자열을 입력 받는다.
    scanf_s("%s", str, 32);

    // 문자열 정보를 저장할 객체 (기본 생성자 사용)
    MyString my_str;
    // 동적 메모리를 할당하고 str에 저장된 문자열을 복사
    my_str = str;
    // 입력된 문자열 뒤에 ' by tipsware' 문자열을 추가한다.
    my_str.AppendString(" by tipsware");
    // 문자열이 잘 복사되었는지 확인한다.
    my_str.ShowStringInfo();

    return 0;
}
```

이제 my_str 객체에 저장된 문자열에 ' by tipsware' 문자열을 덧붙이려면 AppendString 함수를 사용하면 됩니다. 하지만 클래스를 사용하는 개발자 입장에서는 이 작업을 하기 위해 my_str.AppendString(" by tipsware");라고 코드를 사용하는 것보다는 my_str = my_str + " by tipsware";라는 코드가 더 편하고 자연스러운 것입니다. 그래서 my_str = my_str + " by tipsware";라는 표현이 가능하도록 '+' 연산자에 대한 연산자 오버로딩을 사용할 것입니다.

MyString 클래스에 '+' 연산자를 재정의하기 위해 아래와 같이 operator+ 함수를 추가했습니다. '+' 연산자를 재정의 한 코드도 AppendString 함수와 작업이 동일하기 때문에 AppendString 함수를 호출해서 작업을 구현했습니다.

```
class MyString
{
    // ... 다른 코드 생략 ...

    // 저장된 문자열에 다른 문자열을 추가하는 경우에 사용한다.
    // 예를 들어, "abc"가 저장되어 있는데 AppendString("def");라고 하면 "abcdef"가 됩니다.
    void AppendString(const char *ap_str)
    {
        // 전달된 문자열의 길이를 구하고 합쳐졌을때의 길이를 구한다.
        int len = strlen(ap_str), total_len = m_len + len;
        // 합쳐진 문자열을 저장할 메모리를 할당한다.
        char *p_str = new char[total_len];
        // NULL 문자 제외하고 처리하기 위해 1만큼 줄인다.
        m_len--;
        // 새로 할당된 메모리에 기존 문자열을 복사한다.
        memcpy(p_str, mp_str, m_len);
        // 전달된 문자열을 뒤에 추가한다.
        memcpy(p_str + m_len, ap_str, len + 1);
        // 기존 문자열을 제거한다.
        delete[] mp_str;
        // 새로 할당한 메모리를 멤버 변수에 저장한다.
        mp_str = p_str;
        // 문자열 길이도 합쳐진 길이를 저장한다.
        m_len = total_len;
    }

    // '+' 연산자 오버로딩 : '+' 연산자를 문자열을 합치는 연산으로 사용
    char *operator+(const char *ap_str)
    {
        // ap_str 문자열을 기존 문자열에 덧붙인다.
        AppendString(ap_str);
        // 문자열이 저장된 메모리의 주소를 반환한다.
        return mp_str;
    }

    // ... 다른 코드 생략 ...

};
```

이제 '+' 연산자를 재정의 한 코드가 MyString 클래스에 추가되었기 때문에 main 함수에서 my_str.AppendString(" by tipsware");라고 사용했던 코드를 아래와 같이 사용할 수 있습니다.

```
// 문자열 정보를 저장할 객체 (기본 생성자 사용)
MyString my_str;
// 동적 메모리를 할당하고 str에 저장된 문자열을 복사
my_str = str;
// 입력된 문자열 뒤에 ' by tipsware' 문자열을 추가한다.
// my_str.AppendString(" by tipsware");
my_str = my_str + " by tipsware";
// 문자열이 잘 복사되었는지 확인한다.
my_str.ShowStringInfo();
```

하지만 위 코드를 실행하면 아래와 같이 정상 동작하지 않습니다.

```
tipssoft
■■■■■■■■■■■■■■■■■■■■■?: 26

이 창을 닫으려면 아무 키나 누르세요...
```

위와 같이 버그가 생긴 이유는 my_str = my_str + " by tipsware"; 코드에는 '+' 연산자뿐만 아니라 '=' 연산자도 사용되기 때문입니다. 다행이도 MyString 클래스에 '=' 연산자가 연산자 오버로딩 되어 있기 때문에 문법적인 오류가 발생하지는 않지만 '=' 연산자 오버로드된 함수로 전달되는 메모리가 my_str 객체 내부에 선언된 mp_str 멤버 변수에 저장된 주소이기 때문에 버그가 발생한 것입니다. 왜냐하면 operator= 함수가 호출되면 내부적으로 CopyDynamicString 함수가 호출되고 이 함수 내부에서는 CleanUp 함수가 호출되어 mp_str이 사용하던 메모리를 해제합니다. 이렇게 되면 operator= 함수로 전달된 ap_str의 주소가 mp_str이 사용하던 주소이기 때문에 ap_str에 저장된 주소는 이미 해제된 메모리의 주소가 되어 문제가 발생한 것입니다.

따라서 CopyDynamicString 함수에 a_str로 전달된 주소와 mp_str의 주소가 같은 경우에는 작업을 무시하는 코드를 아래와 같이 추가해야 합니다.

```
// 동적메모리를 할당해서 a_str에 전달된 문자열을 복사하는 함수
int CopyDynamicString(const char a_str[])
{
    // 동일한 주소의 메모리가 전달되었다면 무시한다.
    if (a_str != mp_str) {
        // 이미 할당된 메모리가 있으면 제거한다.
        CleanUp();

        // 매개 변수로 전달된 문자열의 길이를 구한다.
        m_len = strlen(a_str) + 1;
        // 문자열을 저장하기 위해 동적 메모리를 할당한다.
        mp_str = new char[m_len];
        // 메모리가 정상적으로 할당되었다면 메모리에 전달된 문자열을 복사한다.
        if (mp_str != NULL) memcpy(mp_str, a_str, m_len);
    }
    // 문자열의 길이를 반환한다.
    return m_len;
}
```

위와 같이 코드를 추가하고 나면 이제 이 예제는 정상적으로 동작할 것입니다. 이번 예제를 실습할 수 있도록 전체 예제 소스를 공개하면 다음과 같습니다.

```
#include <stdio.h>    // printf, scanf_s 함수를 사용하기 위해
#include <string.h>   // strlen, memcpy 함수를 사용하기 위해

class MyString
{
private:
    int m_len;  // NULL 문자를 포함한 문자열의 길이를 저장할 변수
    char *mp_str;  // 문자열이 저장된 메모리의 주소를 기억할 변수

public:
    // 기본 생성자 : 객체를 초기화하기 위해 자동으로 호출되는 함수
    MyString()
    {
        m_len = 0;
        mp_str = NULL;
    }

    // 생성자 : 지정된 문자열을 사용해서 객체를 초기화하는 생성자 함수
    MyString(const char *ap_str)
    {
        mp_str = NULL;
        CopyDynamicString(ap_str);
    }

    // 파괴자 : 객체를 정리하기 위해 자동으로 호출되는 함수
    ~MyString()
    {
        CleanUp();
    }

    // 동적메모리를 할당해서 a_str에 전달된 문자열을 복사하는 함수
    int CopyDynamicString(const char a_str[])
    {
        // 동일한 주소의 메모리가 전달되었다면 무시한다.
        if (a_str != mp_str) {
            // 이미 할당된 메모리가 있으면 제거한다.
            CleanUp();

            // 매개 변수로 전달된 문자열의 길이를 구한다.
            m_len = strlen(a_str) + 1;
            // 문자열을 저장하기 위해 동적 메모리를 할당한다.
            mp_str = new char[m_len];
            // 메모리가 정상적으로 할당되었다면 메모리에 전달된 문자열을 복사한다.
            if (mp_str != NULL) memcpy(mp_str, a_str, m_len);
        }
        // 문자열의 길이를 반환한다.
        return m_len;
    }

    // '=' 연산자 오버로딩
    char *operator= (const char *ap_str)
    {
```

```
        // 전달된 문자열을 동적으로 메모리를 할당하여 복사한다.
        CopyDynamicString(ap_str);
        // 할당된 메모리를 반환한다.
        return mp_str;
    }

    // 저장된 문자열에 다른 문자열을 추가하는 경우에 사용한다.
    // 예를 들어, "abc"가 저장되어 있는데 AppendString("def");라고 하면 "abcdef"가 됩니다.
    void AppendString(const char *ap_str)
    {
        // 전달된 문자열의 길이를 구하고 합쳐졌을때의 길이를 구한다.
        int len = strlen(ap_str), total_len = m_len + len;
        // 합쳐진 문자열을 저장할 메모리를 할당한다.
        char *p_str = new char[total_len];
        // NULL 문자 제외하고 처리하기 위해 1만큼 줄인다.
        m_len--;
        // 새로 할당된 메모리에 기존 문자열을 복사한다.
        memcpy(p_str, mp_str, m_len);
        // 전달된 문자열을 뒤에 추가한다.
        memcpy(p_str + m_len, ap_str, len + 1);
        // 기존 문자열을 제거한다.
        delete[] mp_str;
        // 새로 할당한 메모리를 멤버 변수에 저장한다.
        mp_str = p_str;
        // 문자열 길이도 합쳐진 길이를 저장한다.
        m_len = total_len;
    }

    // '+' 연산자 오버로딩 : '+' 연산자를 문자열을 합치는 연산으로 사용
    char *operator+(const char *ap_str)
    {
        // ap_str 문자열을 기존 문자열에 덧붙인다.
        AppendString(ap_str);
        // 문자열이 저장된 메모리의 주소를 반환한다.
        return mp_str;
    }

    // 이 객체가 관리하는 문자열 정보를 출력하는 함수
    void ShowStringInfo()
    {
        // 관리하는 문자열의 정보를 출력한다.
        if (mp_str != NULL) printf("%s : %d\n", mp_str, m_len);
        else printf("보관된 문자열이 없습니다.\n");
    }

    // 이 객체가 관리하던 정보를 정리하는 함수
    void CleanUp()
    {
        // 할당된 메모리가 있다면 해제한다.
        if (mp_str != NULL) {
            delete[] mp_str;
            mp_str = NULL;
        }
```

```
    }
};

int main()
{
    char str[32];
    // 사용자에게 32자 이하의 문자열을 입력 받는다.
    scanf_s("%s", str, 32);

    // 문자열 정보를 저장할 객체 (기본 생성자 사용)
    MyString my_str;
    // 동적 메모리를 할당하고 str에 저장된 문자열을 복사
    my_str = str;
    // 입력된 문자열 뒤에 ' by tipsware' 문자열을 추가한다.
    // my_str.AppendString(" by tipsware");  // 기존 코드
    my_str = my_str + " by tipsware";
    // 문자열이 잘 복사되었는지 확인한다.
    my_str.ShowStringInfo();

    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
tipssoft
tipssoft by tipsware : 21

이 창을 닫으려면 아무 키나 누르세요...
```

6. 활용하기 좋은 연산자 오버로딩 예시

MyString 클래스를 사용하는 개발자가 아래와 같이 MyString 클래스로 만든 객체에 '412' 문자열은 대입한 다음 이 값은 정숫값 412로 변환하는 기능과 '3.14' 문자열은 대입한 다음 이 값은 실숫값 3.14로 변환하는 기능은 추가해달라고 요청했다면 어떻게 해야 할까요?

```
MyString my_str;

my_str = "412";
// my_str에 저장된 값을 412로 사용하는 코드

my_str = "3.14";
// my_str에 저장된 값을 3.14로 사용하는 코드
```

보통의 경우 해당 기능을 수행하는 함수들을 MyString 클래스에 추가할 것입니다. 예를 들어, 문자열을 정숫값으로 변환하는 ToInteger 함수와 실숫값으로 변환하는 ToDouble 함수를 아래와 같이 추가하면 됩니다. 이때 atoi, aotf 함수를 사용하려면 stdlib.h 헤더 파일을 추가해야 합니다.

```
#include <stdlib.h>    // atoi 함수를 사용하기 위해

class MyString
{
    // ... 다른 코드 생략 ...

    // 내부에 저장된 문자열을 정숫값을 변경해서 반환하는 함수
    int ToInteger()
    {
        // 저장된 문자열이 있다면 정숫값으로 변환해서 반환한다.
        if(mp_str != NULL) return atoi(mp_str);

        return 0;
    }

    // 내부에 저장된 문자열을 실숫값을 변경해서 반환하는 함수
    double ToDouble()
    {
        // 저장된 문자열이 있다면 실숫값으로 변환해서 반환한다.
        if (mp_str != NULL) return atof(mp_str);

        return 0;
    }
    // ... 다른 코드 생략 ...

};
```

그리고 main 함수에 아래와 같이 코드를 구성해서 코드를 테스틀 해보면 됩니다.

```
int main()
{
    MyString my_str;

    // 문자열을 대입한다.
    my_str = "412";
    // 저장된 문자열을 정숫값으로 변경한 값을 얻는다.
    int num = my_str.ToInteger();

    // 문자열을 대입한다.
    my_str = "3.14";
    // 저장된 문자열을 실숫값으로 변경한 값을 얻는다.
    double real = my_str.ToDouble();

    // 변경된 정숫값과 실숫값을 출력한다.
    printf("num : %d, real : %g\n", num, real);
    return 0;
}
```

위 코드는 아래와 같이 출력될 것입니다.

```
num : 412, real : 3.14

이 창을 닫으려면 아무 키나 누르세요...
```

하지만 위와 같이 함수를 추가해서 사용하면 MyString 클래스를 사용하는 개발자는 ToInteger, ToDouble 같은 함수를 알아야지만 사용할 수 있기 때문에 함수를 기억해야 하는 불편함이 생깁니다. 이 경우에도 MyString 클래스를 사용하는 개발자가 ToInteger, ToDouble 같은 함수를 기억할 필요없이 int num = my_str;처럼 사용할 수 있다면 더 좋을 것입니다.

이런 표현이 가능하려면 연산자 오버로딩을 사용해야 하는데 눈에 보이는 연산자가 '=' 이기 때문에 '=' 연산자 오버로딩을 시도하려고 할 것입니다. 하지만 int num = my_str;에는 사용 가능한 연산자가 하나 더 있습니다. C++ 컴파일러는 '=' 연산자의 좌, 우에 있는 피연산자의 자료형이 다르면 오른쪽에 있는 피연산자에 왼쪽 피연산자의 자료형을 강제로 형 변환하는 코드가 추가됩니다. 예를 들어, int num = my_str;은 왼쪽의 자료형이 int이고 오른쪽의 자료형이 MyString이기 때문에 자료형이 서로 다릅니다. 그래서 int num = (int)my_str;은 시도해 보는 것입니다. 즉, (int) 형변환 연산자가 암시적으로 작용되는 것입니다. 따라서 이 수식에서는 '=' 연산자 오버로딩이 아니더라도 (int) 형 변환 연산자를 사용할 수 있기 때문에 (int) 형 변환 연산자 오버로딩을 통해 작업을 할 수 있다는 뜻입니다.

그런데 (int) 연산자를 연산자 오버로딩하기 위해 operator 키워드를 붙이면 operator(int) 처럼 사용되어 operator 키워드가 함수처럼 보이게 되는 문제가 있습니다. 그래서 C++ 언어는 형 변환 연산자를 아래와 같이 두 가지로 제공합니다. 즉, 연산자 오버로드 형식 때문에 형 변환 연산자 형식을 추가로 제공하는 것입니다.

```
int a = (int)5;  // C 언어 스타일 형 변환, C++ 언어도 사용 가능
int b = int(5);  // C++ 언어에서만 사용 가능한 형 변환
```

따라서 (int) 형 변환 연산자 오버로딩은 operator(int) 가 아니라 operator int ( ) 형태로 사용됩니다. 이때 다른 연산자 오버로딩 함수와 달리 반환 자료형을 적지 않는 이유는 형 변환 연산자 자체가 반환 자료형을 변경하는 기능이라서 operator int 라고 하면 반환 자료형 자체가 int이기 때문에 별도의 반환 자료형은 적지 않는 것입니다. (int) 형 변환 연산자 오버로딩 함수를 추가하면서 (double) 형변환 연산자 오버로딩 함수도 함께 추가했습니다.

```
#include <stdlib.h>    // atoi 함수를 사용하기 위해

class MyString
{
    // ... 다른 코드 생략 ...

    // 내부에 저장된 문자열을 정숫값을 변경해서 반환하는 함수
    int ToInteger()
    {
        // 저장된 문자열이 있다면 정숫값으로 변환해서 반환한다.
        if(mp_str != NULL) return atoi(mp_str);

        return 0;
    }

    // 내부에 저장된 문자열을 실숫값을 변경해서 반환하는 함수
    double ToDouble()
    {
        // 저장된 문자열이 있다면 실숫값으로 변환해서 반환한다.
        if (mp_str != NULL) return atof(mp_str);

        return 0;
    }

    // 정수형 형 변환 연산자 오버로딩
    operator int()
    {
        return ToInteger();
    }

    // 실수형 형 변환 연산자 오버로딩
    operator double()
    {
        return ToDouble();
    }

    // ... 다른 코드 생략 ...

};
```

위와 같이 operator int(), operator double() 함수가 추가되면 main 함수에 사용된 코드는 아래와 같이 사용이 단순해 집니다. 이제 MyString 클래스를 사용하는 개발자는 이제 ToInteger, ToDouble 같은 함수의 존재를 알 필요없이 그냥 원하는 자료형을 가진 변수에 대입만 하면 자신이 원하는 결과 값을 얻을 수 있습니다.

```
int main()
{
    MyString my_str;

    // 문자열을 대입한다.
    my_str = "412";
    // 저장된 문자열을 정숫값으로 변경한 값을 얻는다.
    int num = my_str;

    // 문자열을 대입한다.
    my_str = "3.14";
    // 저장된 문자열을 실숫값으로 변경한 값을 얻는다.
    double real = my_str;

    // 변경된 정숫값과 실숫값을 출력한다.
    printf("num : %d, real : %g\n", num, real);

    return 0;
}
```

위 예제를 실습할 수 있도록 전체 코드를 보여드리면 다음과 같습니다.

```
#include <stdio.h>    // printf 함수를 사용하기 위해
#include <string.h>   // strlen, memcpy 함수를 사용하기 위해
#include <stdlib.h>   // atoi 함수를 사용하기 위해

class MyString
{
private:
    int m_len;  // NULL 문자를 포함한 문자열의 길이를 저장할 변수
    char *mp_str;  // 문자열이 저장된 메모리의 주소를 기억할 변수

public:
    // 기본 생성자 : 객체를 초기화하기 위해 자동으로 호출되는 함수
    MyString()
    {
        m_len = 0;
        mp_str = NULL;
    }

    // 생성자 : 지정된 문자열을 사용해서 객체를 초기화하는 생성자 함수
    MyString(const char *ap_str)
    {
        mp_str = NULL;
        CopyDynamicString(ap_str);
    }

    // 파괴자 : 객체를 정리하기 위해 자동으로 호출되는 함수
    ~MyString()
    {
        CleanUp();
    }

    // 동적메모리를 할당해서 a_str에 전달된 문자열을 복사하는 함수
    int CopyDynamicString(const char a_str[])
    {
        // 동일한 주소의 메모리가 전달되었다면 무시한다.
        if (a_str != mp_str) {
            // 이미 할당된 메모리가 있으면 제거한다.
            CleanUp();

            // 매개 변수로 전달된 문자열의 길이를 구한다.
            m_len = strlen(a_str) + 1;
            // 문자열을 저장하기 위해 동적 메모리를 할당한다.
            mp_str = new char[m_len];
            // 메모리가 정상적으로 할당되었다면 메모리에 전달된 문자열을 복사한다.
            if (mp_str != NULL) memcpy(mp_str, a_str, m_len);
        }
        // 문자열의 길이를 반환한다.
        return m_len;
    }

    // '=' 연산자 오버로딩
    char *operator= (const char *ap_str)
```

```
    {
        // 전달된 문자열을 동적으로 메모리를 할당하여 복사한다.
        CopyDynamicString(ap_str);
        // 할당된 메모리를 반환한다.
        return mp_str;
    }

    // 저장된 문자열에 다른 문자열을 추가하는 경우에 사용한다.
    // 예를 들어, "abc"가 저장되어 있는데 AppendString("def");라고 하면 "abcdef"가 됩니다.
    void AppendString(const char *ap_str)
    {
        // 전달된 문자열의 길이를 구하고 합쳐졌을때의 길이를 구한다.
        int len = strlen(ap_str), total_len = m_len + len;
        // 합쳐진 문자열을 저장할 메모리를 할당한다.
        char *p_str = new char[total_len];
        // NULL 문자 제외하고 처리하기 위해 1만큼 줄인다.
        m_len--;
        // 새로 할당된 메모리에 기존 문자열을 복사한다.
        memcpy(p_str, mp_str, m_len);
        // 전달된 문자열을 뒤에 추가한다.
        memcpy(p_str + m_len, ap_str, len + 1);
        // 기존 문자열을 제거한다.
        delete[] mp_str;
        // 새로 할당한 메모리를 멤버 변수에 저장한다.
        mp_str = p_str;
        // 문자열 길이도 합쳐진 길이를 저장한다.
        m_len = total_len;
    }

    // '+' 연산자 오버로딩 : '+' 연산자를 문자열을 합치는 연산으로 사용
    char *operator+(const char *ap_str)
    {
        // ap_str 문자열을 기존 문자열에 덧붙인다.
        AppendString(ap_str);
        // 문자열이 저장된 메모리의 주소를 반환한다.
        return mp_str;
    }

    // 이 객체가 관리하는 문자열 정보를 출력하는 함수
    void ShowStringInfo()
    {
        // 관리하는 문자열의 정보를 출력한다.
        if (mp_str != NULL) printf("%s : %d\n", mp_str, m_len);
        else printf("보관된 문자열이 없습니다.\n");
    }

    // 이 객체가 관리하던 정보를 정리하는 함수
    void CleanUp()
    {
        // 할당된 메모리가 있다면 해제한다.
        if (mp_str != NULL) {
            delete[] mp_str;
            mp_str = NULL;
```

```
            mp_str = NULL;
        }
    }

    // 내부에 저장된 문자열을 정숫값을 변경해서 반환하는 함수
    int ToInteger()
    {
        // 저장된 문자열이 있다면 정숫값으로 변환해서 반환한다.
        if(mp_str != NULL) return atoi(mp_str);

        return 0;
    }

    // 내부에 저장된 문자열을 실숫값을 변경해서 반환하는 함수
    double ToDouble()
    {
        // 저장된 문자열이 있다면 실숫값으로 변환해서 반환한다.
        if (mp_str != NULL) return atof(mp_str);

        return 0;
    }

    // 정수형 형 변환 연산자 오버로딩
    operator int()
    {
        return ToInteger();
    }

    // 실수형 형 변환 연산자 오버로딩
    operator double()
    {
        return ToDouble();
    }
};

int main()
{
    MyString my_str;

    // 문자열을 대입한다.
    my_str = "412";
    // 저장된 문자열을 정숫값으로 변경한 값을 얻는다.
    int num = my_str;

    // 문자열을 대입한다.
    my_str = "3.14";
    // 저장된 문자열을 실숫값으로 변경한 값을 얻는다.
    double real = my_str;

    // 변경된 정숫값과 실숫값을 출력한다.
    printf("num : %d, real : %g\n", num, real);
    return 0;
}
```

위 예제는 아래와 같이 출력됩니다.

```
num : 412, real : 3.14

이 창을 닫으려면 아무 키나 누르세요...
```

7. 생각보다 복잡한 연산자 오버로딩

5번 항목에서 '+' 연산자 오버로딩을 사용하여 문자열에 문자열을 덧붙이는 기능을 개체에 추가했습니다. 별 문제가 없는 것처럼 설명하고 넘어갔지만 해당 코드는 my_str = my_str + " by tipsware"; 같은 코드는 정상적으로 동작하지만 아래와 같이 피연산자 반대로 적으면 제대로 동작하지 않습니다. 즉, 우리가 구현한 코드는 **'+' 연산자를 정의한 객체가 '+' 연산자보다 앞에 있어야지만 가능하다는 뜻**입니다.

```
my_str = " by tipsware" + my_str;
```

따라서 연산자에 대한 다양한 표현을 다 사용하려면 C++ 언어가 제공하는 더 많은 문법을 사용해야 합니다. 하지만 개인적으로는 이런 문법의 활용도가 낮아서 시간 투자대비 효율이 좋지 않고 객체 지향 개념을 무시하는 문법도 포함되어 있습니다. 따라서 C++ 언어를 입문하는 사람들이나 중급 개발자들이 굳이 이런 문법들을 배울 필요가 없다고 생각합니다. 그래서 제가 진행하는 강좌에서는 이 부분에 대해 생략하고 넘어갑니다.

나중에 기회가 되어 저와 C++ 언어에 더 깊게 논의하는 강의가 개설되면 지금 설명을 생략한 문법들에 대해 그때 소개하도록 하겠습니다.

2022.06.02. 09:10 답글쓰기

김성엽 작성자

ㅎㅎㅎ 그래도 우리는 사용해야 하니 어쩔수 없죠 ㅎ

2022.06.02. 09:42 답글쓰기

a12345 2

안녕하십니까 선생님

연산자 오버로드 = 쓸때 클래스 안에 오퍼레이터 함수? 그거 반환형이 char * 인데 꼭 그렇게 해야하나요?

반환값이 필요없어보여서요

실제로 void형으로 해도 잘 작동하길래 왜 반환값이 char *인지 궁금해서 여쭤봅니다.

감사합니다!!

2022.06.19. 18:43 답글쓰기

김성엽 작성자

형식을 맞추려고 사용한거에요~ ㅎ 이미 내부적으로 합쳐져서 괜찮은것처럼 보이지만 만약 대입되는 쪽에 다른 변수가 사용되었다면 또 다른 문제가 생깁니다. 그리고 저런것 때문에 사실 조금더 처리해야하는데 아직 이 시점에서는 배우지 못한 문법이 많아서 저정도로 그냥 설명하고 넘어간거에요 ㅎㅎ

2022.06.19. 19:57 답글쓰기

황금잉어가물치 3

```
#include<stdio.h>
#include<string.h>

class MyString {
private:
int m_len;
char* mp_str;
public:
MyString(){
m_len = 0;
mp_str = NULL;
}

MyString(const char *ap_str{
mp_str = NULL;
CopyDynamicString(ap_str);
}

~MyString(){
CleanUp();
}

void CopyDynamicString(const char* a_str) {

if (a_str != mp_str){
CleanUp();;

m_len = (int)(strlen(a_str) + 1);
mp_str = new char[m_len];
if (mp_str != NULL){
memcpy_s(mp_str, m_len, a_str, m_len);
}
m_len = (m_len)-1;
}
}

void AppendString(const char* ap_str){
int len = (int)strlen(ap_str);
int total_len = m_len + len+1;
```

```
char* p_str = new char[total_len];

memcpy_s(p_str, m_len, mp_str, m_len);
memcpy_s(p_str+m_len,len+1,ap_str,len+1

delete[]mp_str;
memcpy_s(mp_str, total_len, p_str, total_len);
m_len = total_len-1;
}

char* operator=(const char *ap_str){
CopyDynamicString(ap_str);
return(mp_str);
}

void ShowStringInfo(){
if (mp_str != NULL){
printf_s("%s : %d\n",mp_str,m_len);
}
else {
printf_s("보관된 문자열이 없습니다.\n");
}
}

void CleanUp() {
if (mp_str != NULL){
delete []mp_str;
mp_str = NULL;
}
}
};

int main(void) {

char str[32] = { 0 };
scanf_s("%s", str, 32);

MyString my_str;
my_str = str;
my_str.AppendString(" by tipssware");
my_str.ShowStringInfo();
return(0);
}
```

2022.11.13. 18:54 답글쓰기

황금잉어가물치 3

코드를 좀 수정해서 작동시키려는데 자꾸만 에러가 납니다.
CopyDynamicString함수에서는 int대신에 void형으로 변경하면서 return문을 없앴습니다.
또한 m_len의 글자 개수를 맞추기 위해 m_len=m_len-1;을 했습니다.
AppendString 함수에서는 mp_str=p_str 주소값을 넣는 것 대신에 memcpy_s을 활용했습니다.
마지막에는 m_len=total_len-1;을 넣어 글자수를 맞추었고요. 그래도 계속 에러가 발생합니다.
확인 부탁드려요.
그리고 AppendString 함수에서 mp_str=p_str는 mp_str에 p_str주소값을 넣어 동일하게 하는 걸로 이해하면 되는지요?
operator=가 있어 헷갈립니다. 확인부탁드립니다.

2022.11.13. 18:59 답글쓰기

김성엽 M 작성자

황금잉어가물치 memcpy_s 함수를 사용하는 코드에서 바로 윗줄에 mp_str 메모리를 해제하는 코드를 사용하셨네요. 메모리를 해제하고 해제된 주소를 사용해서 메모리를 삭제하려고 하니 문제가 발생할 것입니다.

2022.11.13. 19:05 답글쓰기

김성엽 M 작성자

황금잉어가물치 그리고 memcpy 함수는 이미 사용할 크기를 지정하는 함수라서 memcpy_s 함수로 대체하는 것에 의미가 별로 없습니다. 따라서 굳이 memcpy 함수를 memcpy_s로 사용할 필요가 없습니다.

2022.11.13. 19:06 답글쓰기

김성엽 M 작성자

황금잉어가물치 그리고 공부하실때는 소스를 원하는데로 바꾸는 연습보다는 제가 소개한 소스를 그대로 받아들이는 연습을 먼저해서 코드에 익숙해진다음에 변경하는 연습을 하는 것이 좋습니다. 그래야지 시간을 더 효율적으로 사용할 수 있습니다.

2022.11.13. 19:09 답글쓰기

김성엽 M 작성자

황금잉어가물치 그리고 댓글로 소스를 넣으면 들여쓰기가 안되어서 코드를 보는 것이 생각보다 어려우니, 문제가 되는 부분만 보여주는 것이 좋습니다.

2022.11.13. 19:09 답글쓰기

황금잉어가물치 3

김성엽 아 맞네요. 메모리 해제하면 안되군요. memcpy는 할당된 메모리로 복사하니 이부분을 놓쳤습니다.
알려주셔서 감사합니다.

2022.11.13. 20:12 답글쓰기

카일 3

연산자 오버로딩에 대한 마음가짐이 바뀌는 시간이었습니다. 강의 듣고 나서 감동이 떠나질 않네요. 사용하니 이렇게 깔끔해 지는군요. 감사합니다 ~

2022.11.25. 08:17 답글쓰기

김성엽 M 작성자

ㅎㅎㅎ 언어의 발전이라는 것이 이런거죠. 결국 컴파일러가 얼마나 지원해주느냐의 문제입니다 ㅎ

2022.11.25. 11:15 답글쓰기

카일 3

김성엽 아직 컴파일러를 생각할 수준이 아니어서 생각도 않고 있었는데요. 말씀 들으니 컴파일러 중요성을 생각하게 됩니다.

2022.11.25. 11:53 답글쓰기

김성엽 M 작성자

카일 점점 공부하시다보면 '아~ 이사람들도 이렇게 밖에 할수 없었구나!', 하는 공감하는 단계까지 올라가실거에요 ㅎ

2022.11.25. 11:56 답글쓰기

카일 3

김성엽 넵. 감사합니다~ㅎ

2022.11.25. 11:57 답글쓰기

임레이 3

연산자 오버로딩에서 첫 번째 피연산자는 무조건 객체인가요? int sum = my_num + b;를 b + my_num으로 바꾸니까 컴파일 에러가 나는데 이유를 잘 모르겠습니다.
그리고 매개변수로 두 개 이상의 값을 사용하면 안되던데, 이건 +연산자의 고유 특성(피연산자 2개를 더하는)이라서 그런 건가요?

2023.03.05. 08:18 답글쓰기

김성엽 M 작성자

반대로 하려면 그 형태에 맞게 연산자 정의를 다시 해야 합니다. 그래서 그게 불편하기 때문에 friend 키워드 사용해서 양쪽으로 다 사용할 수 있는 표현을 정의할수 있습니다. 하지만 friend 키워드가 편리함은 주지만 C++ 기본 의도와는 맞지 않기 때문에 개인적으로는 그렇게 사용하는걸 반대하기 때문에 제 강의해서는 생략한 것입니다.

2023.03.05. 12:18 답글쓰기

김성엽 M 작성자

특수 연산자를 제외하고는 연산자 대부분이 단항 연산자나 이항 연산자로 구성되어 있습니다. 이건 연산자의 기본 특성입니다. 예를 들어, 3개의 정숫 값을 더하려면 2 + 3 + 4처럼 연산자를 두 번 사용해서 해결하기 때문에 연산자에 두 개 보다 많은 피연산자를 사용할 필요가 없다고 생각하는 것입니다.

2023.03.05. 12:21 답글쓰기

임레이 3

김성엽 답변 감사합니다!

2023.03.07. 09:07 답글쓰기

정원서 2

선생님, 혹시 AppendString에서 delete[] mp_str;을 하고 mp_str=p_str을 하면 기존에 있던 동적할당 받은 멤버변수가 사라지고,
멤버변수가 새로 동적할당한 주소를 전달받게 되는데, 그러면 이후에 다시 delete[] mp_str을 하면 새로 할당받은 메모리가 삭제되나요>?
또한 연산자 오버로딩에서 char*를 반환하였는데, main에서는 my_str=str; 했을때 my_str은 자료형이 클래스이고 str은 char*인데
my_str안에있는 char*멤버 변수에 알아서 같은 자료형을 찾아서 대입을 시키는 것인가요?

2023.03.08. 04:19 답글쓰기

김성엽 작성자

네~ 새로할당된 주소를 다시 delete 하면 새로 할당된 주소가 해제됩니다.
그리고 반환형은 함수를 만든 사람의 의지이기 때문에 해당 자료형으로 반환하고 받는 쪽과 비교해서 형변환이 가능하면 형변환되어 대입됩니다. 자동으로 찾아서 반환하는 개념은 아닙니다.

2023.03.08. 13:38 답글쓰기

김성엽 작성자

연산자 오버로딩은 대입되는쪽의 자료형에 맞춰서 연산자 오버로딩이 적용된 함수의 매개 변수와 일치하는 함수가 호출될뿐 해당 함수의 반환형은 호출 기준에 관여하지 않습니다.

2023.03.08. 13:39 답글쓰기

정원서 2

김성엽 감사합니다!

2023.03.08. 13:40 답글쓰기

임레이 3

캡처본 밑줄 친 부분을 주석 처리 하면, 실행이 정상적으로 되지 않습니다. 코드 흐름을 따라가면서 분석해 봤을 때 my_str이 초기화 되지 않았기에 my_str의 멤버 변수인 mp_str이 NULL인 상태인데, mp_str이 NULL인 상태라도 어차피 +연산자(AppendString 함수)를 통해 mp_str이 p_str로 할당되기 때문에 정상적으로 작동해야 하는 것 아닌가? 라는 의문이 듭니다.

2023.03.10. 17:43 답글쓰기

김성엽 작성자

해당 코드는 mp_str이 NULL이 되는 상황을 고려한 코드가 아니라서 값을 대입하지 않고 + 연산자 또는 AppendString 함수가 호출되면 NULL 값을 가진 mp_str을 delete 하려고 하기 때문에 문제가 발생하게 됩니다.

2023.03.10. 22:44 답글쓰기

임레이 3

김성엽 헉 책에서 배운 동적 할당 내용에 있는 오류 부분이네요.. 동적 할당 다시 복습해야겠습니다ㅠㅠ 감사합니다!

2023.03.10. 22:48 답글쓰기

김성엽 작성자

임레이 강의하면서 예외처리를 다하면 좋겠지만 일일이 예외처리를 다하게 되면 예제 소스가 너무 길어지다보니, 강의에 맞게 준비하다보니 이렇게 되었네요 ㅎㅎ, 이해해 주시길 바랍니다.

2023.03.10. 23:14 답글쓰기

임레이 3

김성엽 넵 양질의 강의 너무 잘 보고 있습니다

2023.03.11. 08:56 답글쓰기

임레이 3

new 연산자를 통해 동적 할당된 포인터 p에 대해, 연산자 오버로딩을 쓰고 싶은데 operator를 생략하면 컴파일 오류가 뜨더라구요 이건 어쩔 수 없는 부분인가요? 너무 자잘한 것들에만 신경 쓰면서 질문하는 것 같네요.. ㅎㅎ

```
int main() {

    MyString my_str;
    MyString* p = new MyString;

    char str[32];

    printf("문자열을 입력하세요: ");
    scanf_s("%s", str, 31);

    my_str = str;
    my_str.ShowStringInfo();

    p->operator=(str);
    p->ShowStringInfo();

    return 0;
}
```

2023.03.11. 09:52 답글쓰기

김성엽 M 작성자

포인터인 경우에는 *p = str; 이라고 사용해야 합니다.

2023.03.11. 13:21 답글쓰기

임레이 3

김성엽 넵 감사합니다

2023.03.11. 14:19 답글쓰기

모리또 2

강의 잘 듣고 있습니다 선생님
이러한 기능이 어떤 필요로 인해 생겼는지, 초심자에게 어려운 것이 왜 어렵게 느껴지는지 등을 말씀해 주셔셔 더욱 좋은 것 같습니다.
c++을 그래도 한 바퀴 돌아 기초는 안다고 생각했었는데 전혀 그렇지 않다는 걸 강의를 볼수록 체감하고 있습니다ㅠ
쉬워질 날이 올 때까지 열심히 반복해 보겠습니다!

2024.01.26. 20:30 답글쓰기

모리또 2

연산자 오버로딩도 이걸 내가 쓸 때가 있을까 하는 생각이 들었었는데 초보자여서 그런 생각이 들었던 것 같습니다
시야가 넓어지니 너무나 배울 게 많았네요

2024.01.26. 20:48 답글쓰기

김성엽 M 작성자

C++ 언어은 공부할수록 더 많이 보이고 더 많이 들릴거에요 ㅎㅎ 최소한 3번은 보세요 :)
파이팅입니다!

2024.01.26. 20:50 답글쓰기

김성엽 M 작성자

모리또 다른 사람을 위한 클래스를 만들기 시작하면 의외로 많이 사용하게 됩니다 ㅎㅎ
그러니 잘 연마해두세요. 결국 상급자로 가면 사용하게 됩니다 :)

2024.01.26. 21:04 답글쓰기

dh221009

댓글을 남겨보세요

등록