



## MyString 클래스 기본 설계하기

이제 본격적으로 C++에서 제공하는 `string` 클래스를 설계해 보자. 클래스를 새로 설계하는 이유는 `string` 클래스와 같은 기능을 하면서 문자 배열의 단점을 극복하기 위함이다. 그래서 새로 구현한 `string` 클래스는 문자열 배열에 없는 기능도 추가하고 문자열 배열에서 사용하던 기능까지도 재정의해 줄 것이다. 그럼, 먼저 극복해야 할 문자열의 단점과 해결 방법부터 살펴보자.

### 문자 배열의 단점

문자 배열은 정적 메모리 할당을 하기 때문에 문자열의 길이에 상관없이 고정된 메모리를 사용하므로 메모리가 낭비된다. 반대로 할당된 메모리를 지나서 문자열을 저장하게 될 수도 있으므로 위험하다.

### MyString 클래스로 극복

우리가 설계할 `MyString` 클래스는 필요한 만큼의 메모리만 스스로 알아서 할당할 수 있도록 설계한다. 클래스가 생성될 때 초깃값에 의해 알맞은 메모리가 할당되도록 생성자를 구현한다. 생성되는 클래스마다 서로 다른 크기의 메모리를 할당받도록 하려면 동적 메모리 할당을 해야 한다.

### MyString 클래스에 필요한 멤버변수

동적 메모리 할당을 하는 `MyString` 클래스를 구현하려면 문자열의 길이를 저장할 정수형의 멤버변수(`int m_nLen`)와 동적 메모리 할당한 힙 영역을 가리킬 문자 포인터형의 멤버변수(`char *m_pStr`)가 필요하다.

```
int m_nLen;
char *m_pStr;
```

## MyString 클래스에 필요한 생성자 정의하기

어떠한 초깃값으로도 MyString 객체를 생성할 수 있도록 하려면 다음과 같은 2가지 종류의 생성자를 구현해야 한다.

[표 17-1] MyString 클래스의 생성자

생성자	설명
MyString(const char * const str)	문자열 상수나 문자열 배열을 초깃값으로 하는 생성자
MyString()	기본 생성자로 널 값을 갖도록 한다

### ■ 문자열 상수나 문자열 배열을 초깃값으로 하는 생성자

다음과 같이 객체를 생성할 때 초깃값으로 문자열 상수를 지정하는 경우에 호출되는 생성자다.

```
MyString strA("Apple");
```

문자열 상수를 매개변수로 받아 객체를 생성해야 하므로 다음과 같이 매개변수가 char \* 형태여야 한다. 매개변수로 넘겨진 값은 생성자 호출 후에 값이 변경되지 않도록 하려고 예약어 const를 붙였다. 매개변수로 넘겨진 문자열을 new 연산자로 새롭게 할당받은 기억공간에 저장해야 한다.

```
MyString::MyString(const char * const str)
{
    m_nLen = strlen(str)+1;    // ----- ❶
    m_pStr = new char[m_nLen]; // ----- ❷
    strcpy(m_pStr, str);      // ----- ❸
}
```

- ❶ 우선 메모리를 할당할 기억공간의 크기를 알아내야 한다. 생성자의 매개변수인 str을 strlen 함수를 사용해서 문자열의 길이를 구한 후 여기에 더하기 1을 한 값을, 문자열의 길이를 저장하는 변수 m\_nLen에 저장한다. 1을 더한 이유는 strlen 함수는 널 문자 전까지의 문자열의 길이를 재는데, 문자열을 저장할 공간은 널 문자가 저장될 공간까지 고려해서 메모리 할당을 해야 하기 때문이다.
- ❷ 할당받은 힙 영역의 주소값을 char \*형 변수인 m\_pStr에 저장한다.
- ❸ strcpy 함수를 사용해서 힙 영역에 새롭게 할당받은 메모리(m\_pStr)에 매개변수로 넘겨준 문자열(str)을 복사한다.

## ■ 기본 생성자

매개변수가 있는 생성자를 프로그래머가 정의하면 컴파일러로부터 제공받던 기본 생성자를 더 이상 제공받지 못하게 된다.

```
MyString strB;
```

그러므로 MyString 객체를 생성할 때 초깃값을 주지 않으려면 프로그래머가 기본 생성자를 반드시 정의해 주어야 한다. 널 문자를 저장하려고 문자열의 길이를 저장하는 변수 m\_nLen에는 1을 대입한다. new 연산자로 1바이트의 메모리를 할당한 후에 m\_pStr에 할당받은 기억공간의 주소를 저장한다. 할당된 기억공간에 널 문자를 저장한다.

```
MyString::MyString()
{
    m_nLen=1;
    m_pStr=new char[m_nLen];
    strcpy(m_pStr, "");
}
```

## MyString 클래스 소멸자 정의하기

new에 의해 힙 영역에 잡힌 기억공간은 반드시 delete에 의해 메모리가 해제되어야 한다. 메모리 할당만 하고 해제를 하지 않으면 메모리 누수 현상에 의해서 접근 불가능한 메모리가 생겨 다른 자료를 저장할 수 없게 된다. 그리고 점차 사용할 수 있는 메모리 공간이 줄어서 조금만 큰 프로그램을 실행하면 금방 메모리가 부족해 메모리 할당을 할 수 없는 예외사항이 발생하게 된다.

MyString 클래스를 설계하는 과정에서도 생성자에서 new 연산자로 동적 메모리 할당을 하기 때문에 반드시 소멸자에서 할당된 메모리를 해제해 주어야 한다. 다음과 같이 하면 m\_pStr이 가리키는 힙 영역을 delete 연산자로 메모리 해제하고 문자열의 길이는 0으로, m\_pStr에는 NULL 포인터를 저장한다.

```
MyString::~MyString()
{
    delete []m_pStr;
    m_nLen = 0 ;
    m_pStr = NULL;
}
```

### MyString 객체를 출력할 삽입 연산자(<<) 오버로딩하기

cout 객체를 선언한 ostream 클래스에는 문자형 포인터(char \*)를 출력하도록 << 연산자가 오버로딩되어 있다. 그러므로 문자열 상수, 문자열 배열, 문자열 포인터를 cout 객체의 << 연산자를 통해서 출력할 수 있다. 하지만 우리가 새로 설계한 MyString 클래스에 대한 정보는 ostream 클래스에 명시되어 있지 않다. 그러므로 MyString 클래스를 설계하면서 << 연산자에 대한 오버로딩도 함께 해주어야 MyString 클래스를 화면에 출력할 수 있다. 출력을 담당하는 삽입 연산자인 <<는 공식화되어 있기 때문에 이전에 학습한 연산자 오버로딩에서 Complex 객체를 << 연산자의 오버로딩과 동일한 형태로 정의한다.

```
ostream & operator<<(ostream & os, MyString & temp)
{
    cout<<temp.m_pStr;
    return os;
}
```

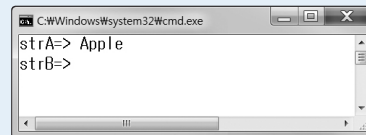
매개변수는 2개인데, 첫 번째 매개변수는 ostream &형, 두 번째 매개변수는 MyString &형이다. 출력 후에 << 연산자를 연속해서 사용하려면 함수의 반환값도 ostream &이어야 한다. 이 때 << 연산자는 MyString 클래스의 프렌드 함수로 선언한다. 왜냐하면 << 연산자는 MyString 클래스의 멤버함수가 아니고 일반 함수지만 MyString 클래스의 private 멤버를 함수 내부에서 사용하기 때문이다.

```
friend ostream & operator<<(ostream &os, MyString & temp);
```

[예제 17-3]은 사용자가 정의한 생성자, 소멸자, 출력을 위해 << 연산자만 이용해서 MyString 클래스를 설계한 첫 번째 프로그램이다.

#### 예제 17-3 MyString 클래스 기본 설계하기(17\_03.cpp)

```
01 #include<iostream>
02 #include<string>
03 using namespace std;
04
05 class MyString
06 {
07 private :
08     int m_nLen;
```



```

09  char *m_pStr;
10  public :
11      MyString(const char * const str);
12      MyString();
13      ~MyString();
14      friend ostream & operator<<(ostream & os, MyString & temp);
15  };
16
17  MyString::MyString(const char * const str)
18  {
19      m_nLen = strlen(str)+1;
20      m_pStr = new char[m_nLen];
21      strcpy(m_pStr, str);
22  }
23
24  MyString::MyString()
25  {
26      m_nLen=1;
27      m_pStr=new char[m_nLen];
28      strcpy(m_pStr, "");
29  }
30
31  MyString::~MyString()
32  {
33      delete []m_pStr;
34      m_nLen = 0 ;
35      m_pStr = NULL;
36  }
37
38  ostream & operator<<(ostream & os, MyString & temp)
39  {
40      cout<<temp.m_pStr;
41      return os;
42  }
43
44  void main()
45  {
46      MyString strA("Apple");
47      MyString strB;

```

```

48
49  cout<<"strA=> "<< strA<<endl;
50  cout<<"strB=> "<< strB<<endl;
51
52  // MyString strC(strA);
53  // cout << "strC=> "<< strC << endl;
54  }

```

**08행** MyString 클래스의 멤버변수는 2개인데, 하나는 문자열의 길이를 저장하기 위한 멤버변수로 정수형 m\_nLen이다.

**09행** 다른 하나는 문자열을 저장할 힙 영역을 가리키는 멤버변수로, 문자형 포인터 m\_pStr이다.

**17행** 문자열을 매개변수로 하는 생성자다. 46행처럼 문자열 상수나 문자 배열을 초깃값으로 설정해서 객체를 생성할 때 호출되는 생성자다. 매개변수로 전달된 문자열의 길이만큼 메모리를 할당한다.

**19행** 생성자의 매개변수인 str을 strlen 함수를 사용해서 문자열의 길이를 구한 후 여기에 1을 더해서 멤버변수 m\_nLen에 저장한다.

**20행** new 연산자를 통해 매개변수로 넘겨준 문자열을 저장할 수 있을 만큼의 기억공간을 할당받는다. 할당받은 힙 영역의 주소값을 cahr \*형 변수인 m\_pStr에 저장한다.

**21행** 힙 영역에 할당받은 메모리에 매개변수로 넘겨준 문자열(str)을 strcpy 함수를 사용해서 저장한다.

**24행** 매개변수가 없는 기본 생성자를 정의한다. 생성자를 하나도 정의하지 않으면 컴파일러가 기본 생성자를 제공해 주지만 일단 프로그래머가 생성자를 구현하면 컴파일러가 제공하는 기본 생성자를 사용할 수 없으므로 44행처럼 객체를 생성할 경우에 대비해서 기본 생성자도 일일이 작성해야 한다.

**33행** 기본 생성자는 new 연산자를 통해 동적으로 할당받은 기억공간을 delete로 해제한다.

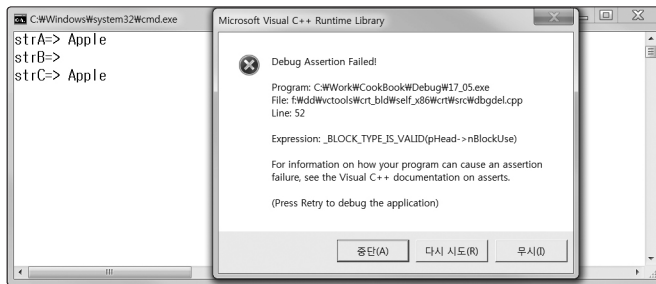


## 복사 생성자

[예제 17-3]에서 52행~53행의 주석을 해제하여 실행해 보자.

```
52 MyString strC(strA);
53 cout<<"strC=> "<< strC<<endl;
```

객체 strC를 새롭게 생성하면서 이미 선언된 객체 strA를 초깃값으로 주었다. 프로그램이 잘 수행되다가 마지막 프로그램을 종료하기 직전에 다음과 같은 에러가 발생한다.



어떠한 문제가 있어서 이러한 예외사항이 발생하는 걸까? 그 이유부터 살펴본 후에 문제를 해결해 보자.

### ① 기본 복사 생성자와 얇은 복사

기본 자료형인 int형 변수에 대해서 다음과 같이 변수 b는 생성되면서 변수 a를 초깃값으로 지정해 줄 수 있다. 그러면 새로 생성되는 변수 b는 변수 a의 값인 5를 초깃값으로 갖게 된다. 동일한 클래스로 선언된 2개의 객체들 사이에서 앞에서와 같은 정수형 변수 사이의 관계가 설정될 수 있다.

```
int a=5;
int b=a;
```

예를 들어 Complex 클래스로 객체를 생성할 경우 다음과 같이 프로그램을 작성할 수 있다. 객체 one은 초깃값으로 real, image 멤버변수에 100, 100을 갖는다. two는 one을 초깃값으로 주어 객체 two가 생성될 때 one의 두 멤버변수 값을 그대로 복사한다.

```
Complex one(100,100);
Complex two(one);
```

모든 객체는 객체 생성 시 초기화를 하려고 생성자가 호출된다. 객체 one을 생성할 때 호출되는 생성자는 (정수형으로 선언된) 매개변수 2개짜리 생성자고, 객체 two가 생성될 때는 자신과 동일한 클래스로 선언된 객체 one을 초깃값으로 주었기 때문에 기본 복사 생성자가 호출된다.

기본 복사 생성자는 매개변수 없는 기본 생성자처럼 C++ 컴파일러에 의해 제공되는 생성자인데, 객체 생성시 동일한 객체를 초깃값으로 줄 때 자동으로 호출된다. 이렇게 컴파일러에 의해서 제공되는 생성자는 클래스의 멤버변수들끼리 일대일 대응으로 값이 복사된다.

```
Complex two(one);
```

위와 같이 객체 생성을 할 경우 내부적으로 다음과 같은 동작이 일어나는데, 이렇게 컴파일러에 의해서 제공되는 기본 복사 생성자에 의해서 멤버변수 값이 복사되는 것을 ‘얕은 복사(shallow copy)’라고 한다.

```
two.real = one.real;
two.image = one.image;
```

## ② 복사 생성자의 오버로딩과 깊은 복사

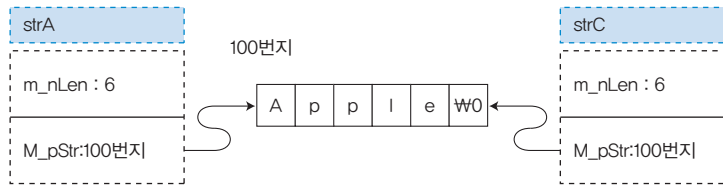
컴파일러에 의해 제공되는 복사 생성자가 특정 클래스에 대해서는 문제를 일으킨다. 기본 복사 생성자를 사용하면 안 되는 대표적인 예가 바로 생성자에서 동적 메모리 할당을 하는 클래스의 경우다.

```
MyString strA("Apple");
```

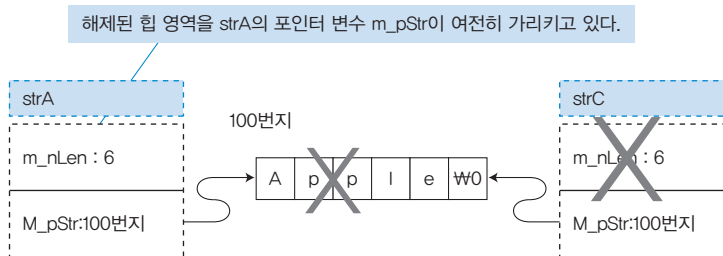


객체 strA가 생성될 때 힙 영역에 동적 메모리를 할당하고 그 기억공간을 멤버변수 m\_pStr이 가리키고 있다. 객체 strA를 초깃값으로 하여 strC를 생성하게 되면 기본 복사 생성자에 의해서 얇은 복사가 일어나서 멤버변수끼리 값이 복사되어 자료값을 저장하고 있는 힙 영역을 두 객체가 가리키는 구조가 된다.

```
MyString strC(strA);
```



strA와 strB가 서로 동일한 힙 영역을 가리키고 있다가 두 객체가 소멸될 때 소멸자가 각각 호출되면서 이미 메모리 해제된 힙 영역을 다시 메모리 해제하려고 하는 문제가 발생한다.



나중에 선언된 객체가 먼저 소멸되므로 객체 strC가 먼저 소멸된다. 객체 strC가 소멸되면서 이미 힙 영역의 자료를 저장하는 공간은 사라졌다. 하지만 객체 strA의 m\_pStr이 여전히 힙 영역을 가리키고 있다가 strA가 소멸될 때 다시 한 번 메모리 해제를 시도하게 된다.

이러한 문제를 해결하려면 복사 생성자가 단순히 멤버변수만 복사(얇은 복사)하는 것이 아니라 새로 생성되는 객체가 별개의 메모리를 힙 영역에 할당받도록 프로그래머가 직접 정의해야 한다. 우선 C++ 컴파일러에 의해 제공되는 기본 생성자의 기본 형태를 살펴보자.

```
클래스명(const 클래스명 &객체명);
```

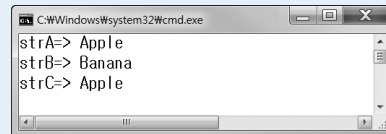
복사 생성자도 생성자이므로 함수명이 클래스명과 동일해야 하고, 복사 생성자 호출 시 이미 존재하는 객체를 실 매개변수로 전달해주므로 이를 형식 매개변수가 참조 변수 형태로 받아서 처리한다.

참조란 이미 선언된 객체에 별칭을 붙여주는 것이므로 따로 메모리 할당을 하지 않는다. 만일 복사 생성자의 형식 매개변수를 값에 의한 호출 방식으로 지정한다면 따로 메모리 할당을 하면서 얇은 복사를 하게 되므로 반드시 참조에 의한 호출 방식으로 지정해야 한다. 또한 생성자를 호출한 후에 원래 존재하던 객체의 값이 변경되어서는 안 되기 때문에 객체명 앞에 예약어 `const`를 덧붙인다. `const`는 상수라는 의미를 내포하고 있는 지정어로 별칭이 가리키는 객체의 값을 가져다 사용만 할 뿐, 값을 변경하지 못하게 하기 위해 사용한다. 그럼 직접 복사 생성자를 만들어 보자.

기본 복사 생성자는 얇은 복사를 하므로 깊은 복사를 하게 하려고 복사 생성자를 정의하는 프로그램이 [예제 17-4]다.

#### 예제 17-4 깊은 복사를 위한 복사 생성자 정의하기(17\_04.cpp)

```
01 #include<iostream>
02 #include<string>
03 using namespace std;
04
05 class MyString
06 {
07 private :
08     int m_nLen;
09     char *m_pStr;
10 public :
11     MyString();
12     MyString(const char * const str);
13     ~MyString();
14     friend ostream & operator<<(ostream & os, MyString & temp);
15     MyString(const MyString& str);
16 };
17
```



```
18 MyString::MyString(const MyString & src)
19 {
20     m_nLen=src.m_nLen;
21     m_pStr=new char[m_nLen];
22     strcpy(m_pStr, src.m_pStr);
23 }
24
25 MyString::MyString(const char * const str)
26 {
27     m_nLen = strlen(str)+1;
28     m_pStr = new char[m_nLen];
29     strcpy(m_pStr, str);
30 }
31
32 MyString::MyString()
33 {
34     m_nLen=1;
35     m_pStr=new char[m_nLen];
36     strcpy(m_pStr, "");
37 }
38
39 MyString::~MyString()
40 {
41     delete []m_pStr;
42     m_nLen = 0 ;
43     m_pStr = NULL;
44 }
45
46 ostream & operator<<(ostream & os, MyString & temp)
47 {
48     cout<<temp.m_pStr;
49     return os;
50 }
51
52 void main()
53 {
54     MyString strA("Apple");
55     MyString strB("Banana");
56
```

```

57  cout<<"strA=> "<< strA<<endl;
58  cout<<"strB=> "<< strB<<endl;
59
60  MyString strC(strA);
61  cout<<"strC=> "<< strC<<endl;
62
63  // strB=strA;
64  // cout<<"strB=> "<< strB<<endl;
65  }

```

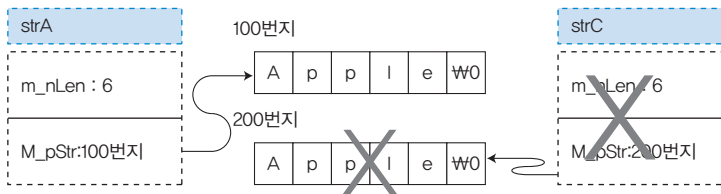
15행 클래스 MyString에 복사 생성자의 원형을 정의했다.

60행 객체 strA를 초기값으로 하여 strC를 생성하게 되면 18행~23행에서 정의한 복사 생성자가 호출된다.



이번에는 21행에서 힙 영역에 배열을 위한 기억공간을 따로 할당한다. 22행에서 따로 할당된 기억공간에 문자열 상수를 저장함으로써 깊은 복사를 할 수 있도록 했다.

strA와 strC의 포인터 변수 m\_pStr이 서로 다른 배열을 가리키므로 객체 strA가 소멸될 때 해제하는 힙 영역과 객체 strC가 소멸될 때 해제하는 힙 영역이 서로 다르므로 에러가 발생하지 않는다.



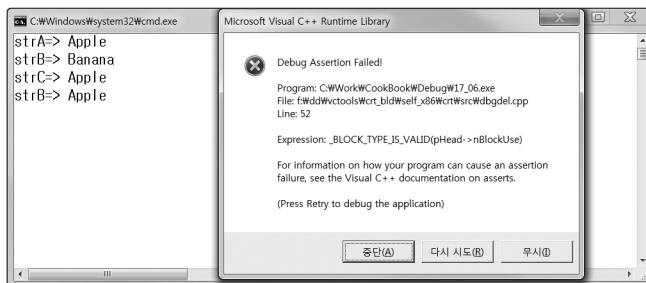


## 대입 연산자 오버로딩

[예제 17-4]의 63행~64행의 주석을 풀고 실행해 보자.

```
63 strB=strA;
64 cout<<"strB=> "<< strB<<endl;
```

이미 생성된 객체 strA의 값을 strB에 대입했다. 프로그램이 잘 수행되는 듯 하다가 마지막 프로그램을 종료하기 바로 전에 다음과 같은 에러가 발생한다. 그것도 실행하는 중에 에러(예외사항)가 발생한다.



어떠한 문제가 있기에 이러한 예외사항이 발생하는 걸까? 그 이유부터 살펴본 후에 문제를 해결해 보자.

### ① 기본 대입 연산자와 얇은 복사

int형 변수에 대해서 다음과 같이 변수 b의 값을 변수 a에 대입해 줄 수 있다. 그러면 변수 b에 변수 a의 값인 5가 저장된다.

```
int a=5;
int b=10;
b=a;
```

동일한 클래스로 선언된 2개의 객체들 사이에서 위와 같은 정수형 변수 사이의 관계가 설정될 수 있다. 예를 들어 Complex 클래스로 객체를 생성할 경우 다음과 같이 프로그램을 작성할 수 있다. 객체 one은 초깃값으로 real, image 멤버변수에 100, 100을 갖는다. two에 one을 대입하면 객체 two에 객체 one의 두 멤버변수 값을 그대로 복사한다.

```
Complex one(100,100);
Complex two(50, 50);
two=one;
```

이렇게 객체끼리 값을 대입하는 것은 서로 다른 클래스로 선언된 객체끼리는 불가능하고 동일한 클래스로 선언된 객체끼리만 가능한데, 이는 C++ 컴파일러가 기본 대입 연산자를 제공해 주기 때문이다. 이렇게 컴파일러에 의해서 제공되는 대입 연산자는 클래스의 멤버 변수들끼리 일대일 대응으로 값이 복사된다. 내부적으로 일어나는 동작은 다음과 같은데, 이렇게 컴파일러에 의해서 제공되는 기본 대입 연산자에 의해서 멤버변수 값이 복사되는 것을 ‘얕은 복사(shallow copy)’라고 한다.

```
two.real = one.real;
two.image = one.image;
```

## ② 대입 연산자 오버로딩과 깊은 복사

컴파일러에 의해 제공되는 대입 연산자가 특정 클래스에 대해서는 문제를 일으킨다. 기본 대입 연산자를 사용하면 안 되는 대표적인 예가 바로 생성자에서 동적 메모리 할당을 하는 클래스의 경우다.

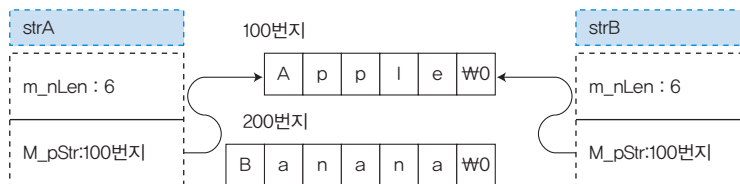
```
MyString strA("Apple");
MyString strB("Banana");
```

객체 strA와 객체 strB가 생성될 때 힙 영역에 동적 메모리 할당을 하고 그 기억공간을 멤버변수 m\_pStr이 가리키고 있다. 객체 strA는 “Apple”을 저장하려고 힙 영역에 6바이트 메모리 할당을 했고, 객체 strB는 “Banana”를 저장할 수 있도록 힙 영역에 따로 7바이트 메모리 할당을 했다.

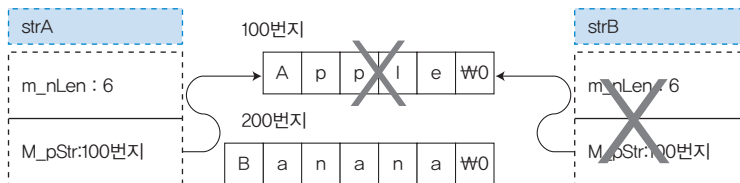


다음 예처럼 strA를 strB에 저장한다고 해 보자.

```
strB=strA;
```

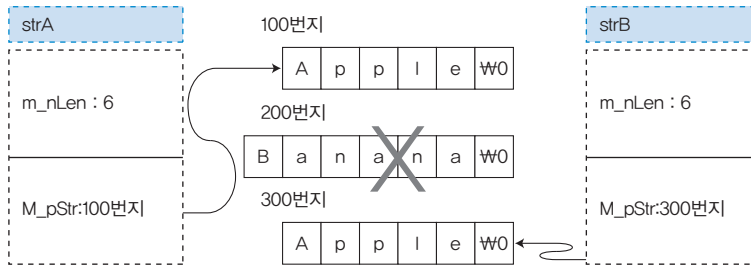


MyString 클래스의 멤버값 2개만이 복사되는 얇은 복사로 인해 strA가 가리키는 힙 역을 strB가 가리키게 된다. 이렇게 두 객체가 동일한 힙 영역을 가리키게 되면, 객체가 소멸될 때 소멸자에 의해서 소멸된 기억공간을 다시 소멸하려고 시도하기 때문에 디버깅 에러가 발생한다.



strB는 strA의 내용만 복사하지 않고, strA와 동일한 형태의 기억공간을 별도로 할당받는 깊은 복사가 이루어져야 별 문제없이 두 객체 사이에 대입 연산자를 사용할 수 있게 된다.

다음은 객체값을 대입 연산자로 치환했을 때 바람직한 형태를 그린 그림이다.



MyString 클래스에서 대입 연산자가 문제를 일으키지 않으려면 포인터 변수 `m_pStr`이 대입 연산자 오른쪽 피연산자의 `m_pStr` 영역을 가리키는 것이 아니고, 다음과 같이 동작해야 한다.


- ❶ 왼쪽 피연산자가 가리키는 힙 영역을 메모리 해제한다.
- ❷ 오른쪽 피연산자로 사용된 객체와 동일한 크기의 메모리를 별개의 기억공간으로 따로 할당받는다.
- ❸ 그 곳에 왼쪽 피연산자가 가지고 있는 자료값을 복사해야 한다. 위에 언급한 사항을 내용으로 하는 대입 연산자는 다음과 같이 정의한다.

```
MyString& MyString::operator=(const MyString &temp)
{
    if(this == &temp)
        return *this;
    delete [] m_pStr;
    m_nLen = temp.m_nLen;
    m_pStr = new char[m_nLen];
    strcpy(m_pStr, temp.m_pStr);
    return *this;
}
```

동적 메모리를 할당하는 MyString 객체는 = 연산자가 멤버변수 값만 그대로 대입하는 기본 대입 연산자를 사용해서는 안 되고 복사 생성자를 재정의했듯이 = 연산자도 재정의해야 한다. = 연산자를 기준으로 오른쪽에 있는 객체의 크기와 동일한 크기로 왼쪽에 있는 객체를 메모리에 새로 할당하여 오른쪽 객체에 저장된 문자열을 복사해야 한다. = 연산자도 << 연산자처럼 연속적으로 사용해야 하므로 반환값이 MyString &이어야 한다.



```
strC = strB = strA;
```



대입 연산자는 strA에 저장된 값을 strB에 저장하고 대입 연산한 결과값을 다시 strC에 저장한다. = 연산자를 MyString 클래스의 멤버함수로 구현했다면 컴파일러에 의해 호출되는 형태는 다음과 같다.

```
strC.operator=(strB.operator=(strA))
```

strB 객체와 strA 객체를 실 매개변수로 하여 operator= 연산자 함수를 호출한 결과값을 다시 operator= 연산자의 매개변수로 사용해야 하므로 operator= 연산자 함수의 결과값은 MyString&이어야 한다. 그리고 = 연산자는 이항 연산자이므로 피연산자 2개가 있어야 하는데 = 연산자를 MyString 클래스의 멤버함수로 구현하게 되면 피연산자 2개 중 왼쪽 피연산자는 this에 의해 관리되므로 매개변수는 한 개만 기술하면 된다.

```
MyString& MyString::operator=(const MyString & temp)
{
    ...
}
```

= 연산자 함수의 원형 정의는 대략 형태가 나왔다. 이번에는 함수를 어떻게 구현해야 할지 살펴보자. 연산자를 기준으로 왼쪽 객체는 이전에 할당받은 힙 영역을 해제하고 오른쪽 객체와 동일한 크기의 메모리를 다시 할당받아야 한다.

```
strA=strA;
```

하지만 만일 위와 같이 대입 연산자가 사용되었다면 동일한 객체를 대입하게 되어 내부적으로 아무런 일도 일어나지 않는다. 그러므로 대입 연산자 오른쪽 피연산자와 왼쪽 피연산자가 동일 객체인지 살펴보고 동일 객체라면 아무런 작업도 하지 않고 함수를 종료한다. 대입 연산자 왼쪽 객체의 주소값이 this에 의해서 관리되므로 이 주소와 대입 연산자 오른쪽 객체가 매개변수로 전달되고 매개변수의 주소값이 동일한 객체라면 두 객체가 같다고 보고 대입 연산자 왼쪽 객체를 결과값으로 반환하고 여기서 함수를 종료시킨다. = 연산자 함수의 반환값이 MyString이므로 왼쪽 피연산자에 해당되는 객체값을 결과값으로 반환해야 하므로 return문 다음에 \*this를 기술했다.

```
if(this == &temp)
    return *this;
```

동일한 객체를 대입하는 것이 아니라면 this로 참조되는 객체의 메모리를 해제하고 매개 변수에 의해 관리되는 객체의 m\_nLen 멤버변수의 크기로 메모리를 재할당받아야 한다.

```
delete [] m_pStr;
m_nLen = temp.m_nLen;
m_pStr = new char[m_nLen];
```

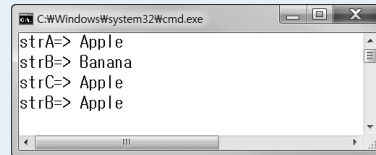
메모리 할당 후에는 대입 연산자 오른쪽 객체가 저장하고 있는 문자열 상수를 대입 연산자 왼쪽 객체에 복사한 후 왼쪽 피연산자가 동일 객체인지 살펴보고 동일 객체라면 아무런 작업도 하지 않고 함수를 종료한다. 연산자 함수의 반환값이 MyString이므로 왼쪽 피연산자에 해당되는 객체값을 결과값으로 반환하므로 return문 다음에 \*this를 기술했다.

```
strcpy(m_pStr, temp.m_pStr);
return *this;
```

기본 대입 연산자는 얇은 복사를 하므로 깊은 복사를 하려고 대입 연산자를 오버로딩하는 프로그램이 [예제 17-5]다.

#### 예제 17-5 깊은 복사를 위한 대입 연산자 오버로딩하기(17\_05.cpp)

```
01 #include<iostream>
02 #include<string>
03 using namespace std;
04
05 class MyString
06 {
07 private :
08     int m_nLen;
09     char *m_pStr;
10 public :
11     MyString();
12     MyString(const char * const str);
13     ~MyString();
14     friend ostream & operator<<(ostream & os, MyString & temp);
15     MyString(const MyString& str);
16     MyString& operator=(const MyString &temp);
```



```
17  };
18
19  MyString& MyString::operator=(const MyString &temp)
20  {
21      if(this == &temp)
22          return *this;
23
24      delete [] m_pStr;
25      m_nLen = temp.m_nLen;
26      m_pStr = new char[m_nLen];
27      strcpy(m_pStr, temp.m_pStr);
28      return *this;
29  }
30  // 17_06.cpp의 내용과 중복
31  MyString::MyString(const MyString &src)
32  {
33      m_nLen=src.m_nLen;
34      m_pStr=new char[m_nLen];
35      strcpy(m_pStr, src.m_pStr);
36  }
37
38  MyString::MyString(const char * const str)
39  {
40      m_nLen = strlen(str)+1;
41      m_pStr = new char[m_nLen];
42      strcpy(m_pStr, str);
43  }
44
45  MyString::MyString()
46  {
47      m_nLen=1;
48      m_pStr=new char[m_nLen];
49      strcpy(m_pStr, "");
50  }
51
52  MyString::~MyString()
53  {
54      delete []m_pStr;
55      m_nLen = 0 ;
```

```

56  m_pStr = NULL;
57  }
58
59  ostream & operator<<(ostream & os, MyString & temp)
60  {
61      cout<<temp.m_pStr;
62      return os;
63  }
64
65  void main()
66  {
67      MyString strA("Apple");
68      MyString strB("Banana");
69
70      cout<<"strA=> "<< strA<<endl;
71      cout<<"strB=> "<< strB<<endl;
72
73      MyString strC(strA);
74      cout<<"strC=> "<< strC<<endl;
75
76      strB=strA;
77      cout<<"strB=> "<< strB<<endl;
78
79      // strA="Apple";
80      // strB="Banana";
81      // strC=strA+strB;
82      // cout<<"strA=> "<< strA<<endl;
83      // cout<<"strB=> "<< strB<<endl;
84      // cout<<"strC=> "<< strC<<endl;
85
86      // cout<<"strC[2]=>"<<strC[2]<<endl;
87  }

```

21행~22행 원본과 대상이 동일한 객체라면 자기 자신을 반환한다.

24행 왼쪽 피연산자가 가리키는 힙 영역을 메모리 해제한다.

26행 오른쪽 피연산자와 동일한 크기의 메모리를 따로 할당받는다.

27행 따로 할당받은 메모리에 왼쪽 피연산자가 가지고 있는 문자열을 복사해야 한다.