CString

C 프로그래밍에서 가장 일반적인 동작은 문자열에 대한 조작입니다. 문자열을 복사한다거나 문자열에서 특정 문자열이나 문자를 찾는 동작입니다. 그런데 C나 C++에는 문자열이란 자료형이 기본 자료형으로 제공 되지 않습니다 그래서 문자열에 대한 조작은 문자의 배열을 사용해야 합니다. 문자의 배열을 사용하기 때문에 위에서 이야기한 문자열에 대한 조작은 항상 복잡하고 귀찮은 작업을 동반하게 됩니다. 또 프로그램 작성자는 문자열을 최대 길이를 기준으로 문자열 버퍼를 확보해야 하므로 불필요하게 공간이 낭비 될 수도 있습니다. 표준 ANSI라이브러리에 문자열을 조작하는 많은 함수를 제공하고 있지만 그리 편리한 것은 아입니다. 특히 문자열을 조작하기 전에 보관하기 위한 버퍼의 할당과 '₩0'으로 끝나야 하는 문자열 특성을 관리하는 것은 귀찮은 작업이기도 하고 또한 버그를 만들 가능성이 많습니다. MFC 프로그램에서도 일반적인 C나 C++프로그램에서와 같이 많은 문자열 처리 작업을 해야 하며, 역시 마찬가지의 문제점을 가지고 있습니다. 그래서 MFC에서는 이와 같은 문자열을 처리 하기 쉽도록 하는 클래스를 가지고 있는데 이것이 CString 이라는 클래스입니다.

MFC가 제공하는 CString이라는 클래스는 MFC로 프로그래밍 작업을 하는데 있어서 가장 많이 사용되는 클래스 중 하나입니다. 그러나 MFC 프로그래머들은 CString이 외부에서 보기에는 너무나 단순하고 쉬운 모양을 가지고 있어서 이에 대한 사용을 너무 쉽게 생각하는 경향이 있습니다. 그러나 CString 클래스 안에는 아주 복잡하고 최적화 된 원리가 숨어 있습니다. 그러므로 CString 클래스의 원리를 바로 알고 잘 사용하면 아주 빠른 속도의 문자열 처리 프로그램을 할 수 있지만 바르지 못하게 사용하는 실수를 한다면 아주 찾기 힘들고 치명적인 버그를 만들 수도 있습니다.

MFC에서 제공하는 CString이라는 클래스는 MFC의 거의 모든 클래스가 일반적으로 상속받는 CObject에서 상속 받지 않는 몇몇 안 되는 클래스 입니다. CString 이 일반적인 MFC 클래스와는 다르게 CObject를 베이스 클래스로 하지 않는 이유는 Microsoft 의 문서에 정확하게 나와 있지 않지만 아주 가볍고 빠른 클래스를 목적으로 하기 위함이라고 생각됩니다. CString이 가벼운 클래스라고는 하지만 그 안에는 복잡한 원리가 숨어 있습니다. 그리고 CString은 문자열의 조작을 목적으로 하므로 문자열의 조작에 필요한 문자열 버퍼와 문자열 비교, 문자열 찾기 등의 기능을 멤버함수로 제공하고 있습니다. 특히 CString이라는 것은 클래스이고 실제로 사용되는 것은 문자열 자체 이므로 CString클래스를 실제 문자열로 추상화 시켜주는 오버로딩 된 연산자 함수를 CString 클래스는 제공합니다. 이외에도 ANSI C 라이브러리인 printf같은 문자 포맷을 만들어 주는 함수와 컴파일러 스위치만 변경 해 줌으로서 UNICODE를 지원하는 문자열 처리도 쉽게 구현할 수 있습니다.

먼저 CString 클래스의 선언을 살펴보겠습니다. CString의 선언은 MFC의 include 디렉토리에 있는 afx.h 라는 헤더 파일에 있습니다. CString 은 LPTSTR m_pchData 이라는 맴버변수와 많은 멤버함수를 가지고 있습니다. 만약 sizeof(CString)을 한다면 4 라는 크기를 얻게 됩니다. 멤버 함수에 가상함수(virtual)가 없습니다면 멤버함수의 갯수는 클래스의 크기에 영향을 전혀 미치지 않습니다. 따라서 m_pchData라는 멤버변수 만이 크기에 계산됩니다. 만약 CString이나 CString의 배열을 만들고 사용하기를 원한다면 특별히 메모리의 효율을 위해 힙(heap) 할당(new)을 사용하는 경우가 있는데 전혀 그렇게 할 필요가 없습니다. 오히려 힙 할당을 사용하는 것은 추가적인 포인터(pointer)에 대한 기억 장소와 연산코드가 필요하므로 불리합니다.

CString에 있는 유일한 멤버 변수인 m_pchData은 우리가 목적으로 하는 문자열에 대한 저장 버퍼로서의 역할을 하는 변수입니다. 만약 CString클래스에 “Hellow”라는 문자열을 저장한다면 바로 m_pchData라는 멤버 변수는 “Hellow”를 가리키는 포인터로 사용됩니다. 그리고 CString은 우리가 원할 때 m_pchData를 반환하여 우리가 “Hellow”라는 문자열을 조작할 수 있게 합니다. 여기까지 본다면 CString은 너무나 단순한 클래스입니다. 그러나 CString은 메모리 관리의 효율과 인자전달의 효율 등 여러 가지 목적으로 추가 적인 논리를 구현하고 있습니다.

CString이 구현한 가장 단순한 논리부터 하나씩 알아보도록 하겠습니다.

CString strData (“Hellow”);

CString strData1 = strData;

만약 위와 같은 코드를 작성한다면, 이것은 CString의 CString(LPCSTR lpsz) 생성자 함수와 const CString& operator=(const CString& stringSrc)라는 연산자 오버로딩(다중정의) 함수를 각각 호출하게 됩니다. 그리고 이 생성자 함수나 연산자 오버로딩 함수에서 해야 할 일은 일반적으로 생각한다면 다음 코드와 같이 너무나 단순합니다.

```
1:    CString::CString(LPCTSTR lpsz)
2:    {
3:       m_pchData = new char[strlen(lpsz)+1];
4:
5:       strcpy(m_pchData, lpsz);
6:    }
7:
```

```
8:    const CString& CString::operator=(const CString& stringSrc)
9:    {
10:       m_pchData = new char[::strlen(stringSrc.m_pchData)+1];
11:
12:       strcpy(m_pchData, stringSrc.m_pchData);
13:
14:       return *this;
15:   }
```

위의 코드로 구현된 CString도 전혀 논리적으로 문제점을 가지고 있지 않습니다. 그러나 이것은 논리적으로 문제점을 가지고 있지는 않지만 CString이 반복적으로 사용되는데 있어서 몇 가지 불합리한 점을 가지고 있습니다. 불합리한 점이란 바로 "Hellow"라는 문자열에 대한 버퍼가 strData와 strData1에 두 번 존재하게 된다는 것입니다. 물론 strData2 = strData1이라고 또 다시 사용한다면 이것은 세 번 존재하게 됩니다. 똑같은 내용을 여러 버퍼에 반복해서 저장한다면 메모리 관리 측면에서 아주 불합리한 것이라 할 수 있습니다. 그리고 새로운 할당과 복사(strcpy)는 시간이 많이 걸리는 논리입니다. 그래서 MFC의 CString은 같은 내용을 저장하는 CString을 위하여 공유 버퍼 개념을 사용합니다. 이 공유 버퍼 개념을 구현하기 위에서 본 const CString& CString::operator=(const CString& stringSrc) 함수는 새로운 버퍼를 할당 받고 복사하기 보다는 넘겨받은 stringSrc의 m_pchData멤버 포인터 값을 자신의 m_pchData에게 복사하면 됩니다. 이렇게 하면 strData2가 가지고 있는 m_pchData 값은 strData가 가지고 있는 값과 같은 값이 되고 같은 버퍼를 사용하게 됩니다. 새로운 버퍼가 생기지 않아 메모리도 절약되고, 할당과 문자열 복사라는 추가적인 논리가 실행 되지 않아 실행 속도 또한 빠릅니다. 구현된 모양을 본다면 다음과 같이 될 것입니다.

```
1:    const CString& CString::operator=(const CString& stringSrc)
2:    {
3:       m_pchData = stringSrc.m_pchData;
4:
5:       return *this;
6:    }
```

그러나 위에 함수로 구현된 CString은 몇가지 논리적 문제점(버그)를 가지고 있습니다. CString 내부에서 본다면 m_pchData가 자기 자신만이 소유하는 버퍼인지 다른 CString에 의해서 공유되는 버퍼인지 전혀 알 수가 없습니다. 이것이 무슨 문제가 될 것인가 생각하는 사람도 있겠으니 이것은 아주 치명적입니다. 위의 예에서 생성한 strData와 strData1이 스택상에서 없어 질 때 컴파일러는 strData와 strData1의 소멸자(destructor)를 호출 할 것입

니다. 소멸자 속에서 m_pchData에 대한 처리는 아주 곤란한 상황에 놓이게 됩니다. 소멸자 내부에서 m_pchData가 공유 될 수도 있다는 가정 때문에 m_pchData를 함부로 삭제 할 수가 없습니다. 그렇다고 해서 아무도 m_pchData에 대해서 삭제하지 않는다면 m_pchData가 가지고 있던 버퍼는 메모리를 누출시키는 결과를 가져옵니다. 참으로 곤란한 상황입니다. 이 CString의 소멸자에서 나타나는 논리적 문제점 외에 또 하나 예를 들어 보겠습니다. CString 에 보면 const CString& operator+ =(LPCTSTR lpsz)라는 연산자 오버로딩 함수가 있는데 이것은 현재 CString이 가지고 있는 문자열 버퍼의 끝에 문자열을 추가하는 역할을 합니다. 만약 위의 strData1이라는 것을 대상으로

strData1 + = “World”;

을 수행한다면 strData1의 내용이 변경되어야 합니다. 그러나 strData1이 가지고 있는 버퍼는 바로 strData가 가지고 있는 버퍼이고 프로그램 작성자는 오직 strData1에만 “World”라는 문자열이 추가 된다고 생각하고 있을 것입니다. 이것은 아주 찾아 내기 어려운 버그를 프로그램 작성자에게 만들어 줍니다.

이제 단순 공유 구조가 가진 문제점을 알았습니다. 실제 CString의 구현은 위에서 말한 문제점 때문에 단순한 공유구조를 가지고 있지 않습니다. 공유되는 m_pchData에 대한 문제점을 해결하기 위해서는 먼저 추가적인 정보가 필요로 합니다. 이 추가적인 정보란 바로 몇 개의 CString에 의해서 m_pchData 버퍼가 공유되고 있는가 하는 정보입니다. 이런 정보를 가 추가된다면 위에서 살펴본 단순 공유버퍼의 문제점을 해결할 수가 입니다. 이 참조카운터 정보는 long형 크기이면 충분할 것입니다. long는 최대 2147483648까지 사용할 수 있고, 이 말은 m_pchData가 2147483648개의 CString에 의해 참조 될 수 있다는 뜻이기 때문입니다. 그럼 이 참조카운터에 대한 정보를 어디에 두는 것이 좋을까요? 참조카운터 또한 m_pchData와 마찬가지로 CString에 의해 공유되어야 합니다. 그러므로 참조카운터 정보도 공유되고 있는 m_pchData라는 버퍼 어디에 인가 두는 것이 좋을 것입니다. 하지만 이 참조카운터 정보를 m_pchData 버퍼의 마지막에 둔다면 또 다른 복잡한 문제가 발생합니다. CString은 + = 같은 연산자 오버로딩을 지원하며 이것은 m_pchData 버퍼의 크기가 동적으로 변할 수도 있다는 뜻입니다. 그러므로 참조카운터 변수를 m_pchData 버퍼의 뒤쪽 어딘가에 두는 것은 좋은 생각이 아닙니다. 그래서 실제 CString은 참조 변수를 m_pchData라는 버퍼의 뒤쪽이 아니라 앞쪽에 두고 있습니다. m_pchData 버퍼의 앞쪽에 참조카운터 변수를 추가정보로 보관 하지만 이 참조카운터 변수는 사용자에게 보여질 필요가 없으므로 항상 m_pchData는 문자열을 가리키도록 설정되어 있습니다.

MFC의 CString 에서는 위에서 이야기한 참조카운터 외에 MFC 의 실행 성능을 향상 시키기 위한 정보를 몇 가지 더 가지고 있습니다. MFC에서는 이 정보를 CStringData라는 구

조체로 만들어 놓았습니다. 그리고 이 CStringData라는 정보구조체는 m_pchData앞쪽에 저장해 두고, 공유되는 모든 CString에서 참조할 수 있도록 해 놓았습니다. CStringData가 클래스가 아니라 구조체라는 데 유의하기 바랍니다. MFC 프로그램에서 사용되는 몇몇 구조체는 구조체 임에도 불구하고 'C'라는 클래스 이름 규칙을 따르고 있습니다. CStringData구조체를 살펴보겠습니다.

```
1:    struct CStringData
2:    {
3:       long nRefs;       // reference count
4:       int nDataLength;
5:       int nAllocLength;
6:    };
```

nRefs라는 멤버변수는 몇 개의 CString이 m_pchData 버퍼를 공유하는 것인가를 카운트하고 있습니다. nDataLength라는 멤버변수는 m_pchData 버퍼가 가지고 있는 문자열의 크기를 나타냅니다. 그리고 nAllocLength는 m_pchData 버퍼에 할당된 실제 크기보다 1이 작은 값을 가지고 있습니다. nDataLength와 nAllocLength는 일반적으로 같은 값을 가지지만 꼭 그런 것은 아닙니다. 실행 성능의 향상을 위해 nAllocLength가 최소한 nDataLength보다 같거나 큰 값을 가지고 있습니다. m_pchData 버퍼와 CStringData구조체의 상호 관계를 나타내면 다음과 같습니다.

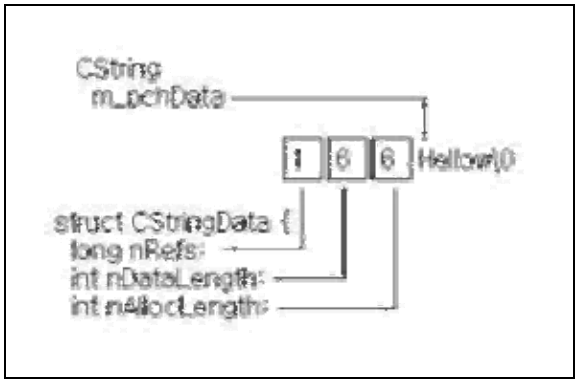

그림 1

실제 CString 은 같은 CString에 대해서만 공유버퍼 논리를 사용합니다. 따라서 LPCTSTR 과 같은 문자열을 받아 들이는 멤버함수는 새롭게 버퍼를 할당합니다. 이렇게 새롭게 버퍼를 할당하기 위해서 내부적으로 사용하는 AllocBuffer라는 protected 멤버 함수를 가지고 있으며, 이 함수를 이용하여 버퍼를 할당하고, 몇 가지 초기화 작업을 실행합니다.

```
1:    void CString::AllocBuffer(int nLen)
2:    {
3:       ASSERT(nLen >= 0);
4:       ASSERT(nLen <= INT_MAX-1);      // max size (enough room for 1 extra)
5:
6:       if (nLen == 0)
7:                Init();
8:       else
9:       {
10:               CStringData* pData= (CStringData*)new BYTE[sizeof(CStringData) +
11:
                      (nLen+1)*sizeof(TCHAR)];
12:
13:               pData->nRefs = 1;
14:               pData->data()[nLen] = '₩0';
15:               pData->nDataLength = nLen;
16:               pData->nAllocLength = nLen;
17:
18:               m_pchData = pData->data();
19:      }
20:   }
```

AllocBuffer 함수는 문자열을 위하여 할당 받을 크기(nLen)가 0 이라면 Init 라는 함수를 호출합니다. Init는 m_pchData 에 대하여 유효한 값(NULL이 아닌 값)을 대입해 주지만 빈 문자열을 가진 주소를 대입합니다. 후에 Init함수에 대해서 조금 더 상세히 기술 하겠습니다. 대부분의 경우처럼 할당 받을 크기(nLen)가 0이 아니라면 할당 받을 크기에 CStringData구조체 크기의 버퍼를 추가로 할당 합니다. 그리고 nLen+1의 크기로 1바이트 더 크게 할당하는 것은 문자열이 '₩0'로 끝나는 특성을 위해서 입니다. 위의 AllocBuff에서 pData는 할당 받은 버퍼의 맨 앞을 가리키고 있습니다. 이곳은 바로 공유버퍼(m_pchData)의 추가 정보를 가지고 있는 곳입니다. 그래서 pData 를 이용하여 참조카운터를 최초(1)라는 의미로 설정하고, 문자열의 크기(nDataLength)와 할당 받은 크기(nAllocLength)를 대입합니다. 마지막으로 m_pchData에 pData->data()에서 넘겨 받은 주소를 설정하는데 이 pData->data()라는 함수는 pData+1을 실행하는 함수 입니다. pData가 CStringData구조체 포인터 므로 pData+1은 바로 공유버퍼를 위한 정보 구조체 바로 다음의 실제 문자열 버퍼를 가리키게 됩니다.

지금까지 알아본 CString의 공유버퍼 논리를 바탕으로 실제 조작에서 CString의 값이 어떻게 변하고 동작하는지 알아보겠습니다. 프로그램 작성자가 CString을 하나 정의하고 그것에 “Hellow”라는 문자열을 대입합니다.

CString strData = “Hellow”;

위의 코드는 CString의 인스턴스를 생성하고(스택상에 CString의 메모리 공간을 확보) CString의 디폴트 생성자(인자가 없는 생성자)를 호출합니다. CSring에 있는 디폴트 생성자는 void Init()라는 내부 protected함수를 호출하고 함수의 제어를 반환합니다. 반환된 제어는 const CString& operator=(LPCSTR lpsz)라는 연산자 오버로딩 함수를 호출하게 됩니다. 이 연산자 오버로딩 함수는 void AssignCopy(int nSrcLen, LPCTSTR lpszSrcData)라는 내부 protected함수를 호출하여 “Hellow”라는 문자열에 대한 길이를 계산하고, 위에서 살펴본 AllocBuffer를 이용하여 “Hellow”라는 문자열의 크기와 공유정보를 위한 영역을 확보한 후 m_pchData와 공유정보 구조체를 초기화 합니다. 그런 다음 m_pchData를 상대로 “Hellow”를 memcopy 합니다. 여기까지의 과정을 거치고 난후의 CString의 모양입니다.

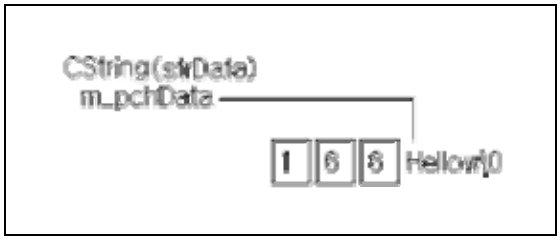

그림 2

위의 CString에서 1 이라는 것은 “Hellow”라는 것이 하나에 의해서 공유되고 있음을 가리키고 있습니다. 이 상황에서 새로운 CString 객체를 선언하고 strData를 대입한다고 하겠습니다.

CString strData1 = strData;

이 코드는 const CString& CString::operator=(const CString& stringSrc)라는 연산자 오버로딩 함수를 호출하게 됩니다. 이 연산자 오버로딩 함수에서 stringSrc는 strData를 가리키는 참조형 변수입니다. 이 연산자 오버로딩 함수는 일단 전달된 stringSrc변수가 이미 현재 자기 자신과 같은 주소를 가리키고 있는지, 즉 이미 공유되고 있는지 확인한 후 아직 공유되고 있지 않다면 단순히 stringSrc의 m_pchData를 현재 m_pchData로 복사한 후 m_pchData의 앞에 있는 CStringData의 nRefs의 값을 1 증가 시켜 둡니다.

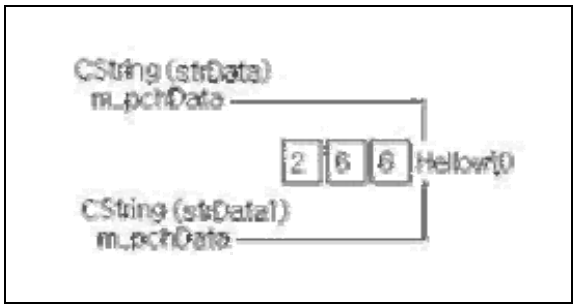

그림 3

여기까지가 가장 기본적인 CString클래스의 동작입니다. 위의 그림에서 참조카운터가 1에서 2로 변한 것을 볼 수 있습니다. 그럼 이런 것이 어떻게 앞서 말한 단순 공유버퍼에서 발생 되었던 문제를 해결하는지 잠시 살펴 보겠습니다.

공유되는 m_pchData를 가지고 있는 객체 strData 와 strData1이 있다고 하겠습니다. 이 두 개의 객체 중에 strData가 소멸되어야 할 때 strData의 소멸자가 호출됩니다. strData의 소멸자에서는 m_pchData앞에 있는 CStringData의 nRefs의 참조카운터를 봅니다. 그리고 이 값이 1보다 크다면 이것은 결과적으로 또 다른 CString, 즉 strData1이 공유하고 있다는 뜻이 됩니다. 그래서 소멸자는 단순히 nRefs의 값을 하나 감소 시키고 아무 일 없었던 것처럼 종료합니다. strData1이 소멸될 때도 strData1의 소멸자가 호출되고 이 때도 역시 CStringData의 nRefs의 참조카운터가 조사되고 이때는 값이 1이므로 자신만이 m_pchData를 참조 하고 있다는 증거입니다. 그래서 이번에는 m_pchData및 앞에 존재하는 CStringData구조체를 메모리에서 삭제합니다. 이와 같은 논리는 CString객체가 소멸될 때 뿐 아니라 strData나 strData1 이 += 같은 연산자 오버로딩 함수에 의해 갱신 되어야 할 때도 똑같이 동작합니다. 공유되고 있는 객체 중에 하나가 변경되어야 할 때 자신의 m_pchData가 공유되고 있다면 공유되고 있는 m_pchData의 nRefs를 하나 감소시키고 새로운 m_pchData버퍼를 할당 받아 공유하고 있던 원본의 m_pchData를 복사합니다. 즉 공유 관계를 끊고 새로운 버퍼로 공유구조를 만들게 됩니다. += 에 대한 실제 변경동작은 이런 공유 관계가 새로이 만들어 지고 난 후에 이루어 집니다. 이런 참조카운터 논리 덕분에 이전에 살펴 보았던 단순 공유버퍼에서 발생되었던 문제는 완전히 사라지게 됩니다.

CString을 사용하는 과정에서 가장 하기 쉬운 실수는 공유버퍼에 대한 이해의 부족에서 오는 경우가 대부분 입니다. CString을 사용하는 것에 대한 최종 목표는 m_pchData에 저장된 문자열입니다. m_pchData는 protected멤버 변수이므로 m_pchData를 사용하기 위해서는 CString으로 부터 m_pchData에 대한 포인터를 반환 받아야 합니다. CString 은 m_pchData를 조작하는 많은 멤버함수를 가지고 있습니다. 이런 멤버함수는 크게 두 가지 부류가 있는데, 하나는 m_pchData에 변화를 주는 것과 m_pchData버퍼에 저장된 값에 변

화를 주지 않는 것입니다. 만약 CString자신이 제공하는 멤버함수가 m_pchData에 대한 변화 가능성과 공유를 하고 있다면 공유 버퍼의 안전한 분리 논리를 사용합니다. 그리고 만약 자신이 제공하는 멤버함수가 m_pchData에 대한 변화를 가하지 않는 것이라면 공유여부 자체가 무시됩니다. 만약 우리가 m_pchData를 반환 받기 위해 CString의 연산자 오버로딩 함수인 operator LPCTSTR() const사용했다고 하겠습니다. 그리고 이렇게 얻은 포인터에 대해서 문자열 대입논리나 함수를 사용했다면 이것은 아주 심각한 문제를 만듭니다. 왜냐하면 operator LPCTSTR() const은 m_pchData를 반환하는 함수이지만 m_pchData가 변경될 가능성이 없다고 추측하는 함수이기 때문입니다. 즉 m_pchData에 대해서 공유여부의 검사 없이 단순 반환만을 구현한 함수입니다.

```
CString strData = "Hellew";

LPCTSTR lpszData = (LPCTSTR) strData;

LpszData[4] = 'o';
```

이와 같이 문자열에 변화를 가하는 경우는 위의 예와 같은 단순 대입문 외에 문자열 조작에서 흔희 사용되는 memmove, strcpy, strncpy, strtok 함수가 있습니다. 그렇다면 이런 논리나 함수를 사용해서 문자열에 변화를 가해야 하는 경우라면 어떻게 해야 할까요? 단순히 문자열이나 문자를 뒤에 추가로 문자열을 더해야 하는 경우라면 CString에서 제공하는 += 연산자오버로딩 함수를 사용해야 합니다. CString의 문자열 가운데 특정한 문자에 변화를 가해야 하는 경우라면 strData[4]='o같이 CString에서 제공하는 []연산자오버로딩 함수를 사용해야 합니다. []연산자오버로딩 함수는 변화 가능성이 있다고 추측 되는 함수입니다. 그 외에 strtok함수나 API로 부터 CString으로 문자열을 반환 받아야 할 경우라면 꼭 CString의 GetBuffer함수를 사용해야 합니다. GetBuffer의 반환 값은 LPTSTR이고 공유버퍼 논리를 사용하여 공유관계를 끈어 버립니다. 그러므로 자유롭게 데이터를 변경하는 논리를 사용할 수 있습니다. 그러나 GetBuffer이란 함수를 사용할 때 특별히 중요한 것이 한가지 있습니다. GetBuffer는 하나의 인자를 갖습니다. 이 인자는 공유관계를 끈을 때 새롭게 할당 받을 버퍼의 크기입니다. GetBufer는 문자열을 주는 함수가 아니고 새로운 버퍼만 할당 받고 그 포인터를 반환하는 함수 이므로 m_pchData앞에 있는 CStringData의 nDataLength의 값을 확정 지을 수 없습니다. nDataLength는 GetBuffer 호출 전의 값을 가지게 됩니다. 일반적으로 이 값은 0이 될 것이고 이것은 여러 가지 문제점을 만듭니다. CString가 제공하는 TrimLeft나 GetLength함수는 nDataLength값을 가지고 연산을 수행합니다. 그러므로 GetBuffer 함수의 호출 후에 아무런 조치 없이 TrimLeft나 GetLength를 사용하면 예상하지 못한 결과를 얻게 됩니다. 여기에 대한 해결책은 GetBuffer다음에 항상 한 쌍을 이루는 ReleaseBuffer를 호출하라는 것입니다. ReleaseBuffer를 호출하기 전에는 절대 CString의

멤버함수를 호출해서는 안됩니다. 아래와 같이 부득이 하게 구조적 예외처리를 위해 GetBuffer을 사용해야 한다면 GetBufferSetLength를 사용해야 합니다. GetBufferSetLength는 GetBuffer에서 있던 문제를 보완한 함수입니다.

```
1:    CString strData;
2:
3:    TRY
4:    {
5:       LPTSTR lpszData = strData.GetBufferSetLength (10);
6:       :
7:    }
8:    END_TRY
9:
10:   strData.ReleaseBuffer ();
```

이것도 TrimLeft에 대한 문제점 만을 해결 하므로 가급적이면 ReleaseBuffer 라는 함수를 반드시 먼저 사용기 바랍니다.

CString이 제공하는 공유버퍼의 논리는 메모리의 효율적인 운용 외에 정확이 이해하고 활용하게 되면 인자 전달이나 반환시 빠른 실행 속도를 낼 수도 있습니다. 어떤 사용자 클래스의 내부에서 멤버로 가지고 있는 CString의 자료를 반환할 때 반환 함수는 LPCTSTR로 반환하는 것보다는 CString이나 CString의 참조형(const CString&)으로 반환하는 것이 더 빠른 속도를 냅니다.

```
1:    CSample
2:    {
3:    private:
4:       CString m_strData;
5:    public:
6:       LPCTSTR GetLPCTSTR (void) const          { return (LPCTSTR) m_strData; }
7:       const CString& GetData (void) const      { return (const CString&)
      m_strData; }
8:    };
9:
10:
11:   CSample Sample;
12:   CString strDes = Sample.GetLPCTSTR ();
```

```
13:    CString strDes = Sample.GetData ();
```

똑같은 값을 반환하는 두 가지 경우라도 첫 번째의 경우는 CSample객체의 m_strData를 LPCTSTR로 변환하기 위한 한번의 함수 호출 및 이 반환 받은 LPCTSTR을 strDes로 전달하여 새로이 버퍼를 할당하고 복사하는 논리가 실행됩니다. 그러나 GetData를 참조형을 반환하므로 함수 호출을 동반하지 않고, 이렇게 반환된 참조형 값도 strDes에 전달되어 새로이 할당 되는 것이 아니라 공유되기 위한 포인터 복사만이 실행 되므로 첫번째 논리보다 빠른 실행 속도를 냅니다.