Visual C++ 2008을 이용한 디버깅

2008-04-18

http://cafe.naver.com/windev

작성자: 최호성(cx8537@naver.com)

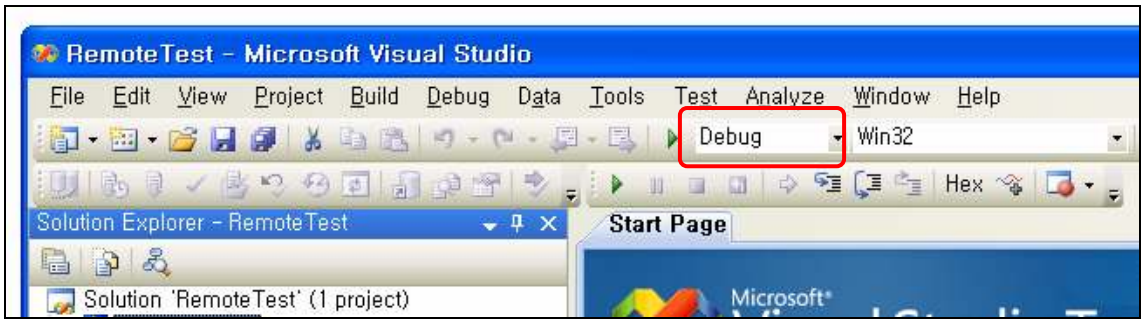
디버깅에 대하여

개발자의 실력을 평가하는 가장 좋은 방법 중 하나로 디버깅 실력을 언급하지 않을 수 없습니다. 사실 경력 1년차 개발자나 3년차 개발자나 이론에 대한 지식적 수준은 많은 차이를 보이지 않는 것이 일반적입니다. 그러나 생산성에 있어서는 많은 차이를 보입니다. 이와 같은 차이를 보이는 가장 결정적인 이유 가운데 하나는 디버깅 실력 때문이라 할 수 있습니다. 물론 이는 어디까지나 제 사견이며 이에 대해 반론을 제기할 분들도 계시리라 생각합니다. 하지만 디버깅 기술이 개발자의 실력을 좌우한다는 것에는 크게 이견이 없으실 것입니다.

윈도우 프로그래밍과 상관 없는 이러한 내용까지 언급하는 이유는 독자 여러분들께 디버깅 실력의 중요성을 강조하고 싶어서 입니다. 스스로 지난 날을 돌아보면서 아쉬운 것은 지금 알고 있는 디버깅 기법을 좀더 일찍 알았더라면 빨리 일을 끝내고 퇴근해도 되었을 일들이 참 많았다는 생각이 드는 것입니다. 최소한 일찍 퇴근은 둘째 치더라도 밤새는 일은 없었지 않았을까 하는 생각도 해봅니다.

이런 이유로 이번 장에서는 Visual C++ 2008을 이용한 디버깅 기법에 대해 소개할까 합니다. 가장 기초적인 디버깅 방법에서부터 원격 디버깅 기법에 이르기까지 다양한 기법들에 대해 언급할 것입니다. 이 설명들을 보다 완벽하게 이해하기 위해서는 멀티스레드 프로그래밍 경험이 있다는 전제와 기본적인 전산개론 지식(정보처리 기능사, 기사수준)이 필요합니다.

빌드 모드



Visual C++에서 새로운 프로젝트를 생성하면 기본적으로 그림과 같이 디버그 모드로 설정되어 있습니다. 디버그 모드는 프로그램을 개발하는 시점에서 개발자의 디버깅 편의를 위한 코드들이 추가된 빌드모드(혹은 설정)라 할 수 있습니다. 반대로 릴리즈 모드는 디버깅을 위한 코드들이 생략되었을 뿐 아니라 컴파일러가 여러분이 작성한 코드를 최적화고 성능

을 극대화 하여 빌드하는 모드 입니다. 물론 이것이 전부는 아니지만 정말 간단하게만 이야기 하자면 이 정도가 될 수 있습니다.

우리가 작성하는 C, C++ 코드는 고급언어입니다. 이 코드는 인간이 알아보기 위한 것이지 기계(CPU)가 알아 보기 위한 코드가 아닙니다. 이 때문에 컴파일러는 인간이 작성한 코드를 기계가 알아 볼 수 있도록 변환해줍니다. 이 과정을 컴파일이라 합니다. 여기서 중요한 것은 같은 C 코드라고 하더라도 컴파일 된 기계어는 기계(CPU)에 따라서 달라질 수 있다는 것입니다. 일반적으로 우리가 사용하고 있는 Visual C++(x86용 Visual C++)로 컴파일한 기계어는 인텔 호환 CPU가 인식할 수 있는 기계어(어셈블리어) 입니다. 지금 이런 복잡한 이야기를 하고 있는 이유는 **빌드 모드(Debug, Release)에 따라 결과 기계어 코드가 달라지기 때문**입니다. 논리적으로는 동일한 코드일 것이나 컴파일 옵션에 따라 어셈블리 코드는 완전히 달라질 수 있습니다.

그리고 모드에 따라 크게 달라지는 것이 하나 더 있습니다. 그것은 **메모리 관리 부분**입니다. 디버그 모드에서는 메모리를 할당 할 때 코드에서 정의한 크기만큼 메모리가 할당되는 것이 아니라 이를 추적하기 위한 정보들을 넣기 위한 공간을 포함하여 더 큰 메모리가 할당된다고 할 수 있습니다. 이는 아주 중요한 문제입니다. 디버그 모드로 컴파일된 실행 파일이 릴리즈 모드로 컴파일 된 라이브러리(DLL)를 로드하여 함수를 호출할 경우 문제가 발생할 수 있습니다. 이에 대해서는 차후에 좀더 자세히 언급하도록 하겠습니다.

이론적인 설명은 이 정도로 해두고 직접 코드를 보면서 이해하도록 합니다. 다음의 코드는 빌드모드에 따라 결과(어셈블리어)가 달라지는 것을 극단적으로 보이는 예입니다.

```
155 ////////////////////////////////////////////////////////////////////////////
156 void CModeDemoDlg::OnBnClickedButtonModedemo()
157 {
158     int nData = 0;
159     for(int i = 0; i < 10; ++i)
160     {
161         nData = 10;
162     }
163 }
164
```

매우 간단한 코드입니다. 이 코드의 내용을 보면 불필요한 내용들뿐입니다. Int형 기억공간에 10회 반복하여 10을 할당하는 것이 전부입니다. 그러므로 이 함수를 수행한다고 하더라도 논리적인 변화나 프로그램 실행에 영향을 줄만한 내용이 사실상 없습니다. 다시 언급하지만 이 코드가 **불필요한 내용들이라는 사실에 주목**하시기 바랍니다.

이 코드를 디버그 모드로 빌드하여 디스어셈블한 코드를 보면 다음과 같습니다.

```
//////////////////////////////////////////////////////////////////////////
void CModeDemoDlg::OnBnClickedButtonModedemo()
{
00412F10  push        ebp
00412F11  mov         ebp,esp
00412F13  sub         esp,0E4h
00412F19  push        ebx
00412F1A  push        esi
00412F1B  push        edi
00412F1C  push        ecx
00412F1D  lea         edi,[ebp-0E4h]
00412F23  mov         ecx,39h
00412F28  mov         eax,0CCCCCCCCh
00412F2D  rep stos    dword ptr es:[edi]
00412F2F  pop         ecx
00412F30  mov         dword ptr [ebp-8],ecx
    int nData = 0;
00412F33  mov         dword ptr [nData],0
    for(int i = 0; i < 10; ++i)
00412F3A  mov         dword ptr [i],0
00412F41  jmp         CModeDemoDlg::OnBnClickedButtonModedemo+3Ch (412F4Ch)
00412F43  mov         eax,dword ptr [i]
00412F46  add         eax,1
00412F49  mov         dword ptr [i],eax
00412F4C  cmp         dword ptr [i],0Ah
00412F50  jge         CModeDemoDlg::OnBnClickedButtonModedemo+4Bh (412F5Bh)
    {
        nData = 10;
00412F52  mov         dword ptr [nData],0Ah
00412F59  jmp         CModeDemoDlg::OnBnClickedButtonModedemo+33h (412F43h)
    }
}
00412F5B  pop         edi
00412F5C  pop         esi
00412F5D  pop         ebx
00412F5E  mov         esp,ebp
00412F60  pop         ebp
00412F61  ret
```

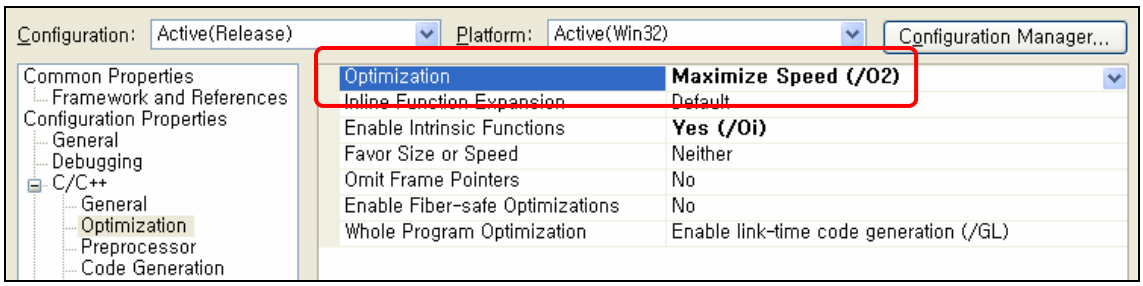
복잡한 어셈블리어 코드가 나오니 당황스러울 수 있겠으나 자세히 살펴보면 다는 몰라도 대충이라도 내용을 어림 짐작할 수 있을 것입니다. 특히 nData = 10; 코드에 대한 디스어셈블 결과는 충분히 이해할 수 있는 것입니다. (※ 0Ah는 16진수 A 즉 10진수 10이 됩니다.) 그리고 생각보다 많은 어셈블리 명령이 필요하다는 사실을 알 수 있습니다.

그렇다면 동일한 코드를 릴리즈 모드로 컴파일 했을 때는 얼마나 어셈블리 코드가 줄어드는지 다음의 코드를 보시기 바랍니다.

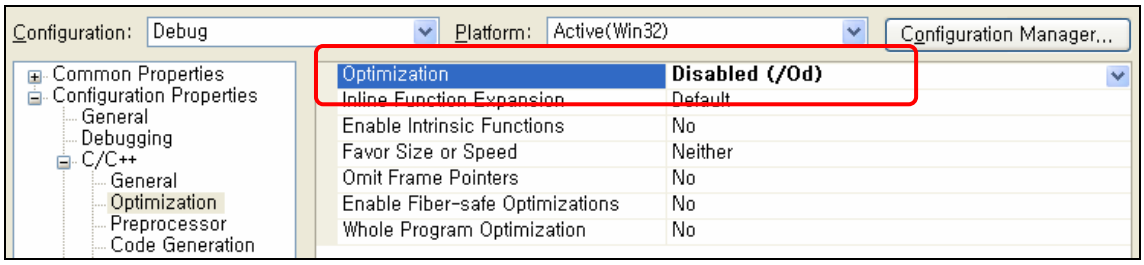
```
////////////////////////////////////////////////////////////////////////////
void CModeDemoDlg::OnBnClickedButtonModedemo()
{
    int nData = 0;
    for(int i = 0; i < 10; ++i)
    {
        nData = 10;
    }
}
00401440  ret
```

다시 한번 당황하셨을 수도 있겠습니다. 동일한 코드를 릴리즈 모드로 빌드한 결과 어셈블리 명령어 하나(ret)로 끝났습니다. 앞서 이미 언급했듯이 작성된 코드가 불필요한 내용들로 가득하기 때문입니다. 컴파일러가 컴파일 도중에 이를 판별하여 아무런 명령도 수행하지 않고 함수가 리턴(ret) 하도록 번역한 것입니다. 현재 우리가 사용하는 컴파일러들도 지속적으로 발전해 왔습니다. 이제는 단순히 기계어로 코드를 번역하는 수준을 넘어 논리적으로 부적절한 부분을 제거하거나 연산속도를 증가시키기 위해 어셈블리 레벨에서 코드를 최적화 합니다.

이와 관련된 내용들은 프로젝트 설정(Alt + F7)에서 직접 확인할 수 있습니다. 다음 그림은 릴리즈 모드일 때 최적화에 대한 컴파일러 설정화면입니다.



반대로 디버그 모드일 때는 다음과 같습니다.



이외에도 많은 설정상의 차이가 있습니다. 그 모든 것들에 대해 언급할 수는 없으므로 보다 자세한 정보는 MSDN이나 Visual C++ 2008 도움말을 참고하시기 바랍니다.

다음의 디스어셈블 코드는 앞서 예에서 반복문 안에 Sleep(1);을 추가하여 릴리즈 모드로 빌드한 것입니다.

```
//////////////////////////////////////////////////////////////////////////////
void CModeDemoDlg::OnBnClickedButtonModedemo()
{
00401450  push        esi
00401451  push        edi
    int nData = 0;
    for(int i = 0; i < 10; ++i)
00401452  mov         edi,dword ptr [__imp__Sleep@4 (403020h)]
00401458  mov         esi,0Ah
0040145D  lea         ecx,[ecx]
    {
        nData = 10;
        Sleep(1);
00401460  push        1
00401462  call        edi
00401464  sub         esi,1
00401467  jne         CModeDemoDlg::OnBnClickedButtonModedemo+10h (401460h)
00401469  pop         edi
0040146A  pop         esi
    }
}
0040146B  ret
```

Sleep() 함수가 호출되면서 스레드의 컨텍스트 스위칭이 발생하므로 실행에 큰 영향을 주게 됩니다. 때문에 앞서와 같이 아무 것도 하지 않고 리턴하는 것이 아니라 반복문을 10회 수행합니다. 그런데 어셈블리 코드를 잘 보시면 nData = 10; 코드는 수행하지 않는다는 것을 알 수 있습니다.

다음의 코드는 릴리즈 모드로 빌드하되 속도 최적화 옵션을 디버그 모드와 마찬가지로 사용하지 않도록 한 것입니다. nData = 10; 코드가 어셈블리 코드로 번역되었음을 알 수 있습니다.

```
//////////////////////////////////////////////////////////////////////////////
void CModeDemoDlg::OnBnClickedButtonModedemo()
{
004015A0  push        ebp
004015A1  mov         ebp,esp
004015A3  sub         esp,0Ch
004015A6  mov         dword ptr [ebp-0Ch],ecx
    int nData = 0;
004015A9  mov         dword ptr [nData],0
    for(int i = 0; i < 10; ++i)
004015B0  mov         dword ptr [i],0
004015B7  jmp         CModeDemoDlg::OnBnClickedButtonModedemo+22h (4015C2h)
004015B9  mov         eax,dword ptr [i]
004015BC  add         eax,1
004015BF  mov         dword ptr [i],eax
004015C2  cmp         dword ptr [i],0Ah
004015C6  jge         CModeDemoDlg::OnBnClickedButtonModedemo+39h (4015D9h)
    {
        nData = 10;
004015C8  mov         dword ptr [nData],0Ah
        Sleep(1);
004015CF  push        1
004015D1  call        dword ptr [__imp__Sleep@4 (403020h)]
004015D7  jmp         CModeDemoDlg::OnBnClickedButtonModedemo+19h (4015B9h)
    }
}
004015D9  mov         esp,ebp
004015DB  pop         ebp
004015DC  ret
```

흔히 디버그 모드가 릴리즈 모드보다 속도가 느리다고 합니다. 이는 우리가 작성한 코드가 논리적으로 동일한 결과를 얻을 수 있는 전제하에서 어셈블리 레벨 최적화를 적용하지 않았기 때문이기도 하고 실제로도 어셈블리 명령의 개수에서 차이가 나기 때문입니다.

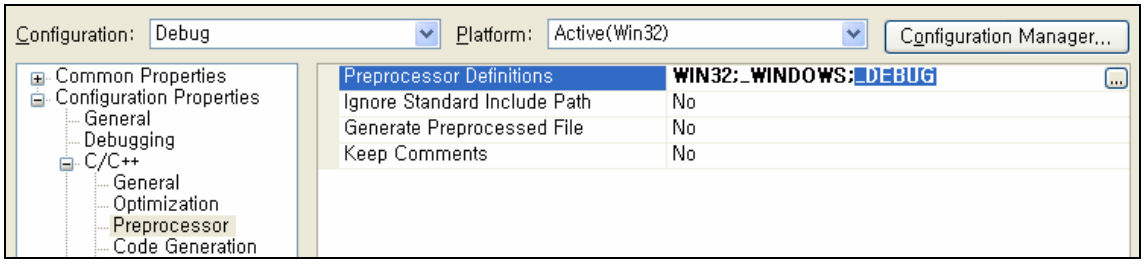
선배로써 한마디!

상당수의 초/중급 개발자들이 디버그 모드일 때는 괜찮다가 릴리즈 모드로 빌드했더니 런타임 에러가 발생한다고 불평할 때가 종종 있습니다. 문제는 에러가 발생한 것이 아니라 이에 대해 어떻게 대응할 것인가에 있습니다. 에러가 발생한 근본적인 원인을 찾아 코드를 수정하는 것이 가장 바람직한 모습이겠으나 이런 저런 변명을 늘어 놓으면서 속도 최적화 옵션을 사용하지 않도록 하여 에러를 덮어버리기도 합니다. 그래도 이 정도는 양반입니다. 가끔은 디버그 모드로 프로그램을 배포하는 배짱 좋은 개발자도 보았습니다. 경우에 따라 최적화 옵션을 제거해야 할 수도 있겠으나 그것은 분명 다른 이유이어야 할 것입니다.

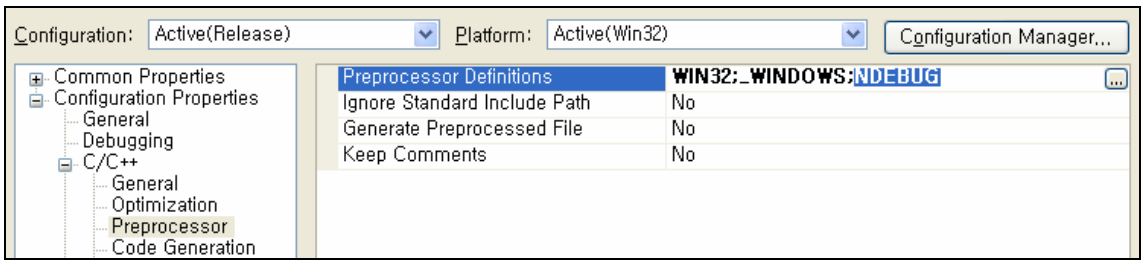
빌드모드에 따라 어셈블리 코드 이외에 메모리 관련처리도 다릅니다. 다음의 코드는 MFC 위저드가 생성한 기본 프레임 코드에 늘 들어가 있는 내용입니다.

```
// ModeDemoDlg.cpp : 구현 파일
//

#include "stdafx.h"
#include "ModeDemo.h"
#include "ModeDemoDlg.h"

//////////////////////////////////////////////////////////////////////////////
#ifdef _DEBUG
#define new DEBUG_NEW
#endif
```

_DEBUG가 정의(#define)되어 있다면 **new 연산자를 DEBUG_NEW로 재정의** 하라는 내용입니다. 디버그 모드일 때 프로젝트 설정을 보면 다음과 같은 내용이 들어있습니다.



전처리기 정의에 _DEBUG를 정의하고 있습니다. 릴리즈 모드일 때는 다음 그림과 같이 **NDEBUG**가 정의 되어있고 _DEBUG는 정의하지 않습니다.



그러므로 디버그 모드로 빌드하면 new 연산자의 연산이 릴리즈 모드일 때와 다를 수 밖에 없습니다. 다음의 코드는 **Afx.h** 파일에서 DEBUG_NEW를 정의한 코드입니다.

```
1343 #if defined(_DEBUG) && !defined(_AFX_NO_DEBUG_CRT)
1344
1345  // Memory tracking allocation
1346  void* AFX_CDECL operator new(size_t nSize, LPCSTR lpszFileName, int nLine);
1347  #define DEBUG_NEW new(THIS_FILE, __LINE__)
1348 #if _MSC_VER >= 1200
1349  void AFX_CDECL operator delete(void* p, LPCSTR lpszFileName, int nLine);
1350  #endif
1351
```

new 연산이 이루어진 코드상의 구체적인 위치를 명시하는 부분이 추가된다는 사실을 알 수 있습니다. 이는 디버그 모드로 실행했을 때 메모리 할당 및 삭제와 같은 연산을 추적하기 위한 것입니다. 디버그 모드로 실행 한 후 메모리 누수가 발생하면 Visual C++는 구체적인 코드 위치까지 알려줍니다.

다음의 코드는 int형 메모리를 할당한 후 해제하지 않는 코드 예 입니다

```
168  ////////////////////////////////////////////////////////////////////////////
169 void CModeDemoDlg::OnBnClickedButtonMemoryleak()
170  {
171      int* pnData = new int;
172
173      *pnData = 0x11223344;
174  }
175
```

이 코드를 수행하고 프로그램을 종료하면 출력(Output)창에 다음과 같은 메시지가 출력됩니다.

```
Dumping objects ->
c:₩documents and settings₩최호성₩my documents₩visual studio
2008₩projects₩modedemo₩modedemo₩modedemodlg.cpp(171) : {534} normal block at 0x0038BDB0, 4
bytes long.
 Data: <D3" > 44 33 22 11
Object dump complete.
The program '[2484] ModeDemo.exe: Native' has exited with code 0 (0x0).
```

해제하지 않는 메모리의 크기뿐만 아니라 구체적인 내용까지 출력해줍니다. 이를 통해서 개발자는 누수된 메모리가 할당된 위치(예에서는 ModeDemoDlg.cpp의 171번행)를 명확히 알 수 있습니다. 디버그 모드로 프로그램을 실행하였다면 늘 출력화면을 주의 깊게 보시기 바랍니다. 개발자가 놓친 부분에 대해 알려주는 정보가 출력될 수 있습니다.

디버그 모드와 릴리즈 모드의 차이가 지금 언급한 이것이 전부는 아니겠으나 이 두 가지 정도만 알고 있어도 충분하리라 생각합니다. 이 두 가지를 요약하여 정리하면 다음과 같습니다.

① 코드 최적화 옵션에 따른 기계어의 차이
② 메모리 할당 연산(new, malloc() 등)시 추적을 위한 코드추가 여부

끝으로 메모리 관리방식의 차이 때문에 발생할 수 있는 문제에 대해 알아 두어야 합니다. 이는 DLL과 밀접한 관련이 있습니다. A.EXE라는 실행 파일이 B.DLL 파일을 로딩하여 사용한다고 가정했을 때, 만일 A.EXE는 릴리즈 모드로 빌드된 것이고 B.DLL은 디버그 모드로 빌드된 것이라면 과연 A.EXE는 아무 문제없이 실행될 수 있을 것인지 생각해야 합니다. 결론부터 말씀 드리면 대부분 에러가 발생하여 프로그램이 비정상 종료되거나 예기치 못한 실행결과를 산출할 수도 있습니다.

이는 매우 중요한 문제입니다. 초보 시절에 이런 문제에 부딪히면 쉽게 해결하기가 어렵습니다. 왜냐하면 코드의 논리상에는 아무런 문제가 없기 때문입니다. 이는 완전히 지식적인 혹은 경험적인 문제에 해당되는 것입니다. 다음의 그림은 이와 같은 상황을 좀더 구체적으로 설명한 가상의 코드 예입니다.

A.EXE

```
void A()
{
        …
        CString strTest = _T(“”);
        FuncB(strTest);
        …
}
```

B.DLL

```
void FuncB(CString& strData)
{
        …
        strData.Format(“%s”,
                “TEST”);
        …
}
```

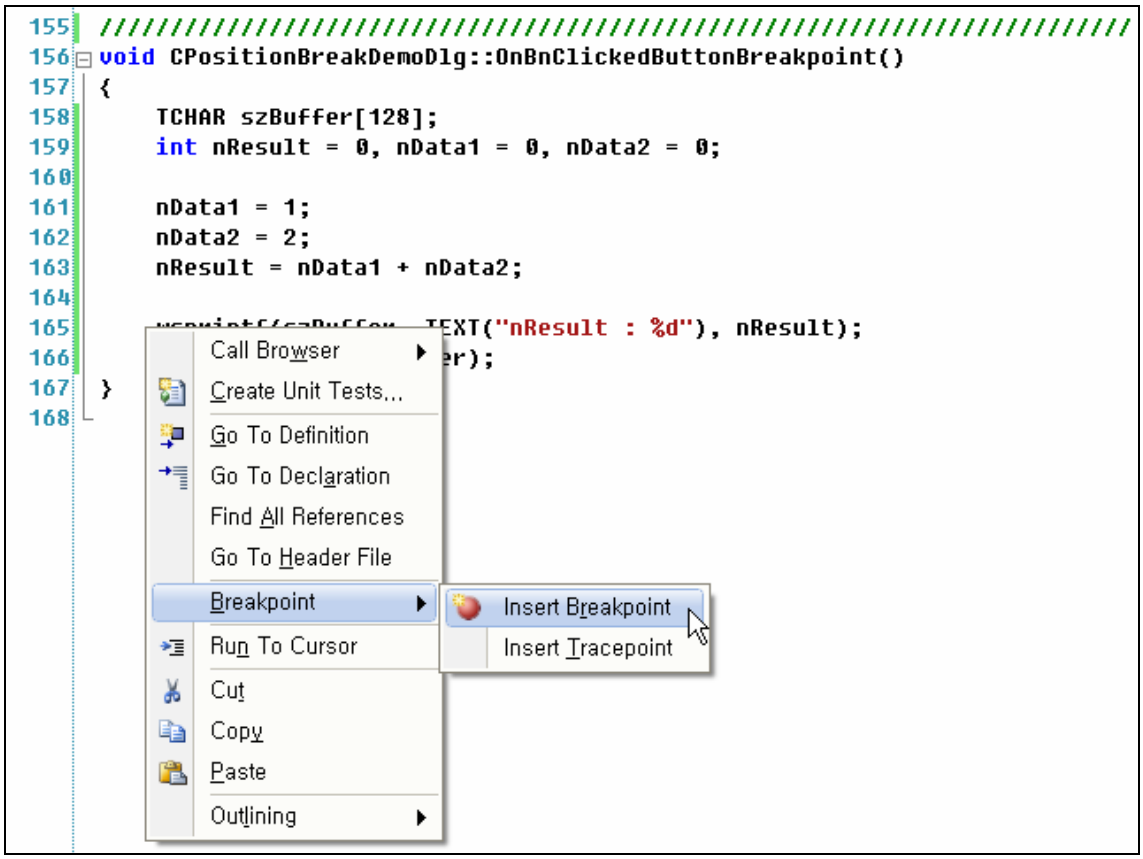
물론 이 코드가 항상 문제를 만드는 것은 아닙니다. 게다가 어떤 컴파일러를 이용하여 컴파일 했는가에 따라 결과는 달라질 수 있습니다. 중요한 것은 언제 어느 때 어떤 이유로 프로그램이 비정상 종료될지 예측하기 어렵다는 것입니다. 그러므로 **실행파일이 릴리즈 모드로 빌드된 것이라면 당연히 연관된 DLL들도 릴리즈 모드로 빌드된 것이어야 합니다.** 하나는 디버그 모드 다른 하나는 릴리즈 모드와 같은 조합은 절대 바람직하지 않습니다.

위치 브레이크 포인트

Visual C++ 2008의 디버거를 사용할 때 가장 많이 사용하는 기능이 바로 위치 브레이크 포인트 입니다. 아마도 지금 이 글을 읽기 전에 위치 브레이크 포인트를 설정/해제하는 정도를 이미 알고 있을 수도 있을 것입니다. 하지만 지금은 모른다고 가정하고 설명을 이어가겠습니다.

위치 브레이크 포인트는 특정 코드가 실행되는 시점이 되면 디버그 할 수 있도록 실행을 잠시 멈추도록 하는 역할을 합니다. 이 위치 브레이크 포인트는 통상 "위치"라는 말을 빼고 브레이크 포인트라 부릅니다. 그런데 제가 지금 구체적으로 "위치"라는 말을 언급하고 있는 이유는 "데이터 브레이크 포인트"와 구별하기 위해서 입니다. 데이터 브레이크 포인트에 대해서는 다음 장에서 상세히 설명할 것입니다.

위치 브레이크 포인트를 설정하기 위해서는 마우스 오른쪽 버튼을 눌러 다음 그림과 같이 팝업 메뉴에서 "Insert Breakpoint" 메뉴를 선택하면 됩니다. 아니면 단축키 F9키를 눌러 브레이크 포인트를 설정/해제할 수 있습니다.

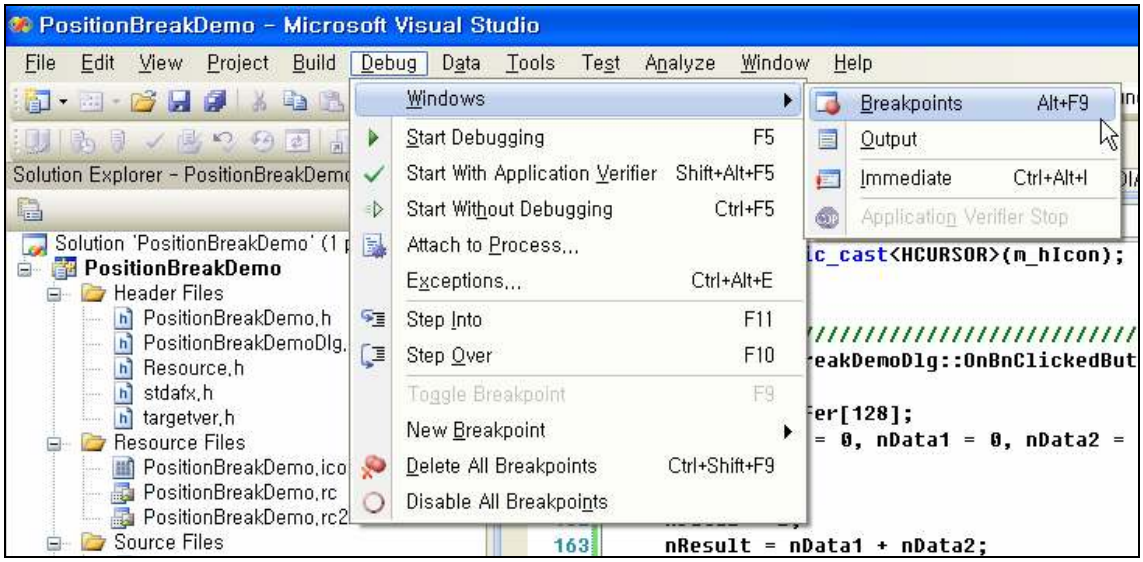


위치 브레이크포인트가 설정되면 다음 그림에서 보는 것과 같이 해당 소스코드 앞에 붉은

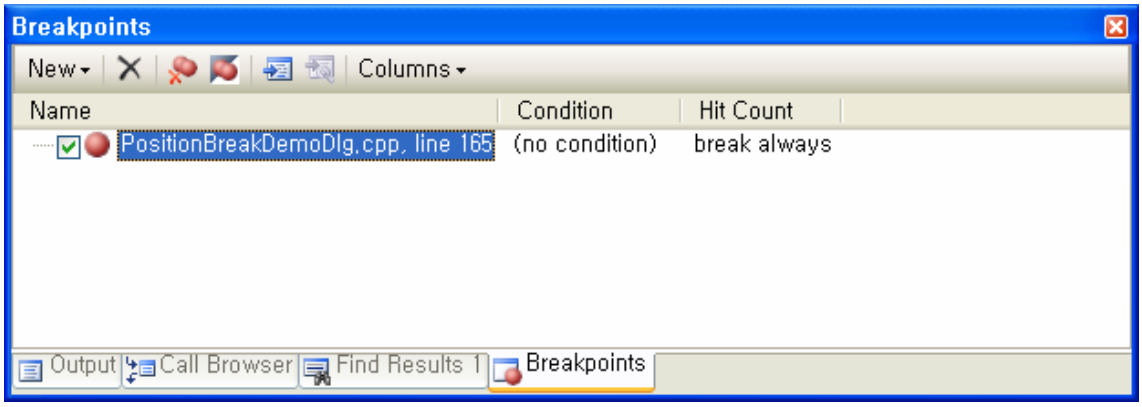
색 원표시가 나타나게 됩니다. 이는 이 위치에 브레이크 포인트가 설정되었음을 의미합니다. 만일 해제하고 싶다면 다시 F9키를 눌러 해제하시면 됩니다.

```
155 ////////////////////////////////////////////////////////////////////////////
156 void CPositionBreakDemoDlg::OnBnClickedButtonBreakpoint()
157 {
158     TCHAR szBuffer[128];
159     int nResult = 0, nData1 = 0, nData2 = 0;
160
161     nData1 = 1;
162     nData2 = 2;
163     nResult = nData1 + nData2;
164
165     wsprintf(szBuffer, TEXT("nResult : %d"), nResult);
166     AfxMessageBox(szBuffer);
167 }
168
```

이와 같이 프로젝트 내에서 설정된 브레이크 포인트들을 소스코드 에디터 화면에서만 확인할 수 있는 것은 아닙니다. Alt + F9 키를 눌러 브레이크 포인트 관리 윈도우를 보시면 브레이크 포인트에 대한 보다 자세한 정보를 볼 수 있습니다.



다음 그림은 브레이크 포인트 관리 윈도우 화면입니다. PositionBreakDemoDlg.cpp 파일 165번째 행에 브레이크 포인트가 설정되었음을 알 수 있습니다.

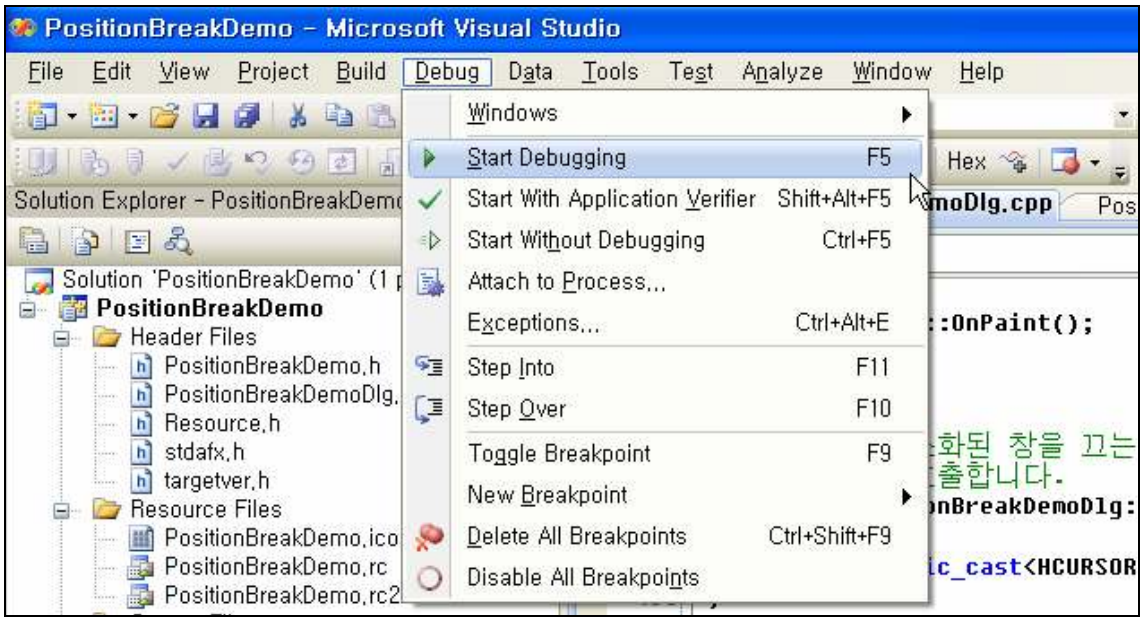


이름(Name) 필드에 구체적인 위치가 보이고 그 항목 앞에 체크박스가 있습니다. 만일 이 체크를 제거하면 브레이크 포인트는 동작하지 않습니다. 코드에서 확인하면 다음과 같이 브레이크 포인트 표시 모양이 변경됩니다.

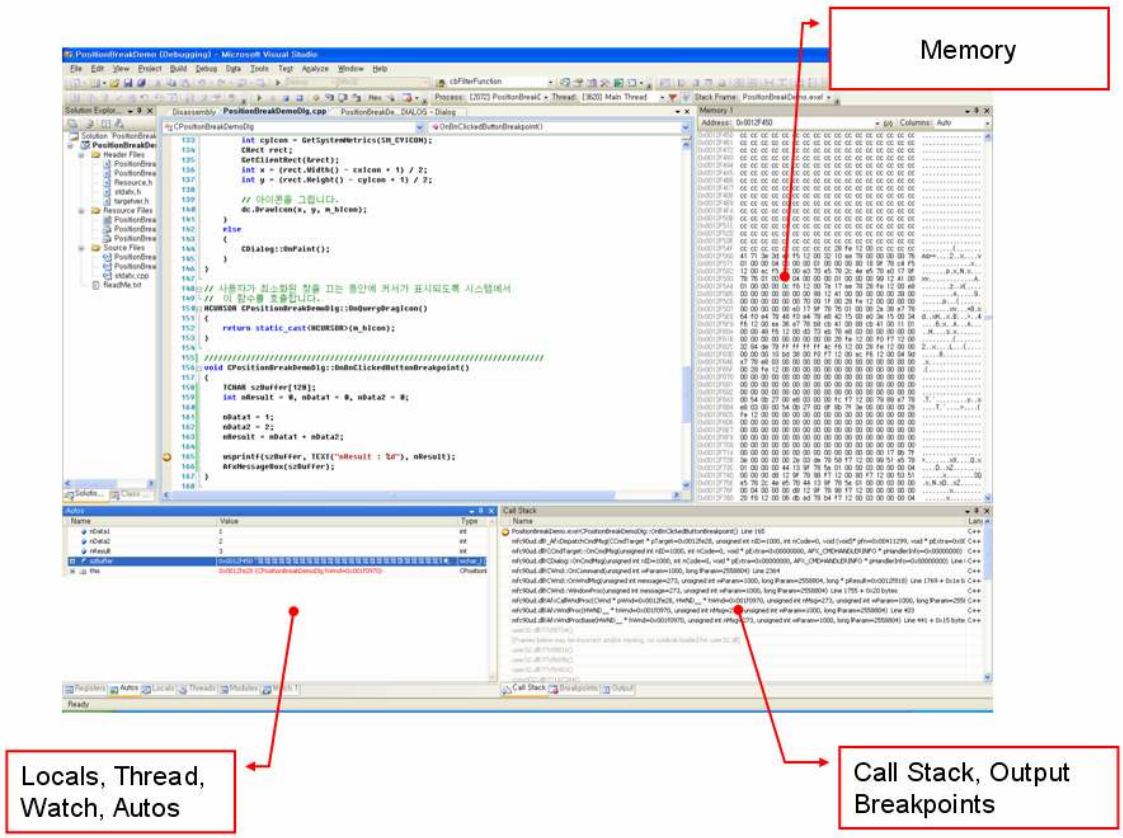
```
164
165     wsprintf(szBuffer, TEXT("nResult : %d"), nResult);
166     AfxMessageBox(szBuffer);
167 }
168
```

굳이 의미를 따지자면 설정은 되어있으나 동작하지 않는(Disabled) 브레이크 포인트가 되는 것입니다.

이처럼 설정된 브레이크 포인트가 동작하도록 하려면 반드시 **디버그 모드로 실행**하여야 합니다. 그러기 위해서 다음과 같이 "Start Debugging" 메뉴를 선택하거나 단축이 **F5**를 눌러 실행합니다.



그러면 Visual C++의 화면이 디버그 모드로 전환 됩니다. 다음 그림은 필자가 주로 구성하는 Visual C++의 디버그 모드 화면입니다.



꼭 이와 같이 하실 필요는 없습니다. 개인 취향이나 환경에 따라 적절히 설정하시면 됩니다. 다만 이와 같은 설정을 추천하는 이유는 이 모든 정보를 한 화면에서 봐야 할 일이 자주 발생하기 때문입니다.

그러면 구체적인 실행 예를 살펴보도록 하겠습니다.

```
155 ////////////////////////////////////////////////////////////////////////////
156 void CPositionBreakDemoDlg::OnBnClickedButtonBreakpoint()
157 {
158     TCHAR szBuffer[128];
159     int nResult = 0, nData1 = 0, nData2 = 0;
160
161     nData1 = 1;
162     nData2 = 2;
163     nResult = nData1 + nData2;
164
165     wsprintf(szBuffer, TEXT("nResult : %d"), nResult);
166     AfxMessageBox(szBuffer);
167 }
168
```

OnBnClickedButtonBreakpoint() 함수는 PositionBreakDemo 예제에서 “Break point” 버튼을 눌렀을 때 호출되는 함수입니다. 앞서 그림에서 브레이크 포인트에 노란색 화살표가 중첩된 것을 볼 수 있는데 이 노란색 화살표가 의미하는 것은 그 행을 수행하기 직전이라는 뜻입니다. 그러므로 코드에서는 **wsprintf() 함수를 호출하기 직전**이 됩니다.

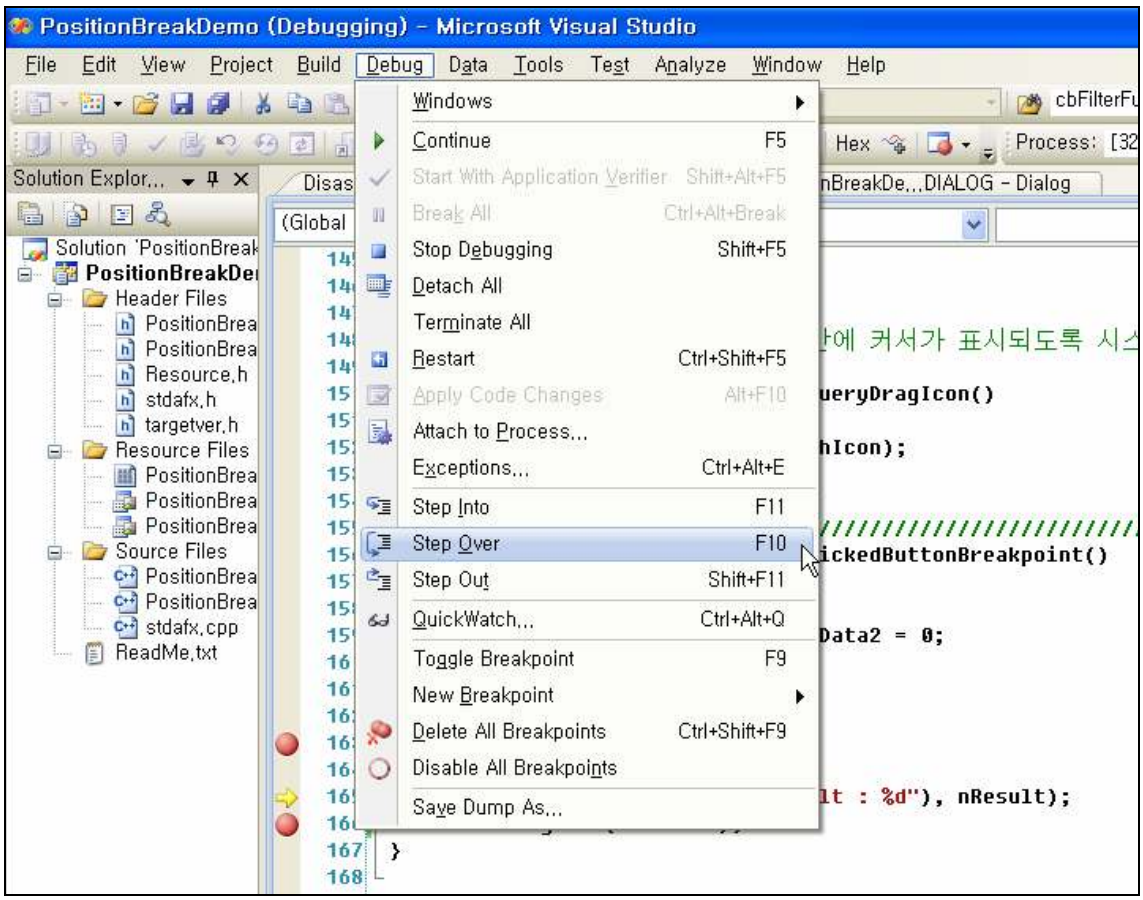
프로세스의 실행은 일시 정지된 상태를 유지합니다. 이 상태에서 각 변수의 값이나 메모리 혹은 호출 스택을 검사하여 문제를 찾아내면 됩니다. 이에 대해서는 좀더 자세히 다룰 것입니다. 아무튼 이 상태에서 다시 F5 키를 누르면 다음 브레이크 포인트가 있는 위치까지 계속 실행이 됩니다. 이는 다음 그림과 같이 브레이크 포인트가 163번 행과 166번 행 두 곳에 설정되었을 때 163번 행에서 브레이크 포인트가 동작하고 다시 F5 키를 누르면 166번 행을 실행하려 할 때 다시 브레이크 포인트가 동작하는 것을 의미합니다.

```
155 ////////////////////////////////////////////////////////////////////////////
156 void CPositionBreakDemoDlg::OnBnClickedButtonBreakpoint()
157 {
158     TCHAR szBuffer[128];
159     int nResult = 0, nData1 = 0, nData2 = 0;
160
161     nData1 = 1;
162     nData2 = 2;
163     nResult = nData1 + nData2;
164
165     wsprintf(szBuffer, TEXT("nResult : %d"), nResult);
166     AfxMessageBox(szBuffer);
167 }
168
```

그러므로 디버그 모드 실행은 브레이크 포인트가 나타나기 전까지 프로세스를 실행하는 것이라 할 수 있습니다. 만일 163번 행의 브레이크 포인트가 동작했을 때 F10키를 누르면 다음 브레이크 포인트가 아니라 프로시저 단위로 실행이 됩니다. 다음 그림은 F10 키를 눌러 실행을 추적하는 화면입니다.

```
155 ////////////////////////////////////////////////////////////////////////////
156 void CPositionBreakDemoDlg::OnBnClickedButtonBreakpoint()
157 {
158     TCHAR szBuffer[128];
159     int nResult = 0, nData1 = 0, nData2 = 0;
160
161     nData1 = 1;
162     nData2 = 2;
163     nResult = nData1 + nData2;
164
165     wsprintf(szBuffer, TEXT("nResult : %d"), nResult);
166     AfxMessageBox(szBuffer);
167 }
168
```

이 F10키는 Step Over 메뉴에 대한 단축키 입니다.



Step Into(F11) 메뉴는 보다 상세하게 함수를 추적하는 메뉴입니다. 만일 현재 실행할 코드가 함수의 호출일 경우 해당 함수 코드를 찾아가 한 행씩 추적하여 실행합니다. 대부분의 개발자들이 디버깅을 하면서 가장 많이 사용하는 단축키를 정리해보면 다음과 같습니다.

① F5 (디버그 모드 실행)
② F9 (브레이크 포인트 설정/해제)
③ F10 (프로시저 단위 실행)
④ F11 (행 단위 실행)

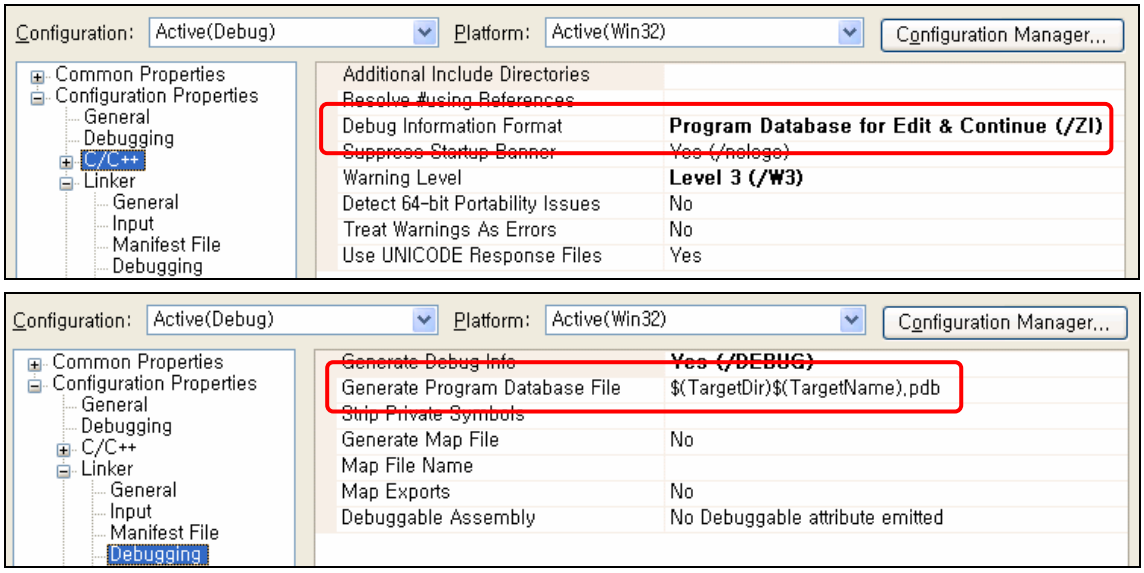
아무래도 메뉴를 선택하는 것보다는 단축키를 이용하는 것이 훨씬 편리할 것입니다. 참고로 Ctrl + F5 키를 누르면 빌드된 파일을 일반 모드로 실행합니다. 이 때는 실행된 프로세스와 디버거가 서로 분리되어 있으므로 디버깅을 할 수 없습니다. 이 부분에 대해서는 “실행 중인 프로세스 연결”에서 상세히 다룹니다.

프로그램 디버그 데이터 베이스와 호출 스택

빌드모드에 따라서 빌드 결과는 확연히 차이가 난다고 이미 설명했습니다. 그리고 디버그 모드 실행에 대해서도 여러분은 이미 알고 있습니다. 그런데 여기서 한 가지 중요한 사실을 묵과하고 넘어간 부분이 있습니다. 그것은 상당수의 사람들이 디버그 모드로 빌드한 경우에만 디버깅이 가능한 것으로 알고 있다는 것입니다. 그러나 그것은 잘못 알고 있는 것입니다. **빌드모드에 상관없이 링크가 완료된 바이너리 파일에 대한 프로그램 디버그 데이터 베이스 파일만 있다면 디버그 모드 실행이 가능**합니다.

그렇다면 이제는 이 프로그램 디버그 데이터 베이스 파일(XXX.pdb)의 역할에 대해 알아야 합니다. 우선 이 파일은 빌드된 바이너리를 디버깅할 수 있는 정보를 담고 있는 파일입니다. 어셈블리 코드단위로 실행이 될 때 그 코드가 C, C++ 소스 코드 어디에 해당하는 것인지 함수 혹은 변수 이름은 무엇인지 등에 대한 정보가 담겨 있습니다.

다음 그림은 PdbDemo 예제의 디버그 모드 설정중에서 프로그램 디버그 데이터베이스 파일과 관련된 설정을 보여주는 것입니다.



Visual C++ 2008은 **기본적으로 모드에 상관없이 PDB 파일을 생성**하는 것이 기본 설정입니다. 이 때문에 **릴리즈 모드로 빌드했어도 F5키를 누르면 디버그 모드로 실행이 되는 것**입니다. 그렇다 보니 아예 릴리즈 모드로 설정한 상태에서 개발을 진행하기도 합니다.

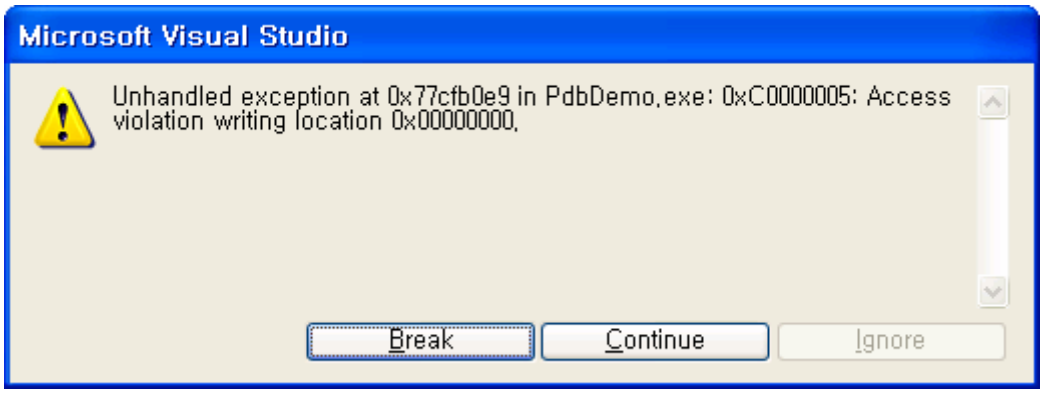
다음 그림은 PdbDemo 예제를 릴리즈 모드로 빌드했을 때 생성된 PDB 파일의 모습을 윈도우 탐색기로 확인한 것입니다.

이름	크기	종류	수정한 날짜
PdbDemo.exe	84KB	응용 프로그램	2008-04-17 오후 11:03
PdbDemo.pdb	4,027KB	Program Debug Database	2008-04-17 오후 11:03

다음 코드는 강제로 에러를 발생시키기 위해 만든 코드 예 입니다. 릴리즈 모드 상태에서 F5 키를 눌러 디버그 모드로 실행하고 “PDB Test” 버튼을 누르면 이 코드가 실행됩니다.

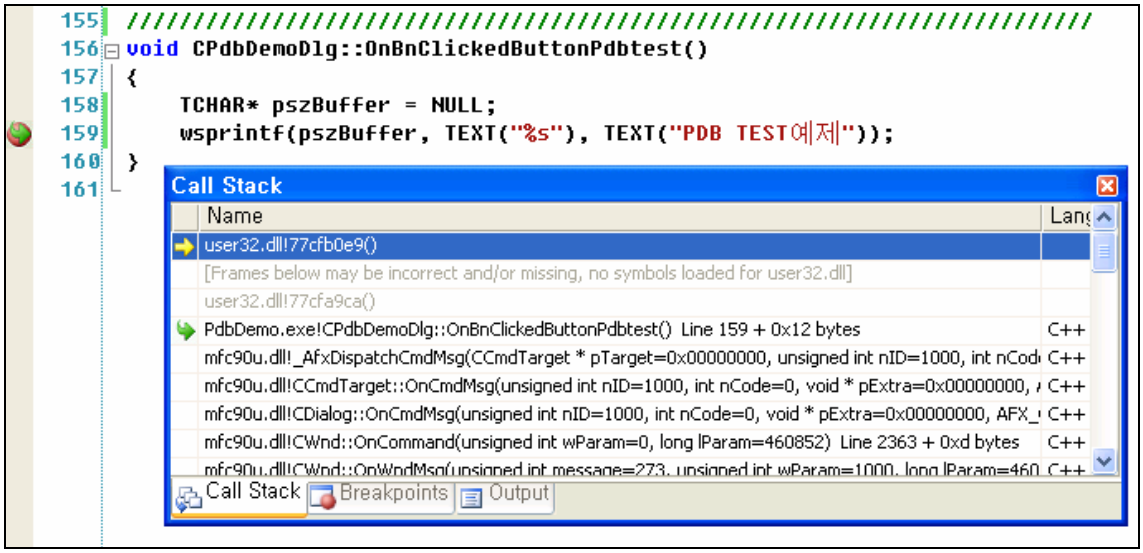
```
155 ////////////////////////////////////////////////////////////////////////////
156 void CPdbDemoDlg::OnBnClickedButtonPdbtest()
157 {
158     TCHAR* pszBuffer = NULL;
159     wsprintf(pszBuffer, TEXT("%s"), TEXT("PDB TEST예제"));
160 }
161
```

pszBuffer는 NULL을 가리키고 있으므로 wsprintf() 함수가 호출되면 에러가 발생할 것입니다. 159번 행에서 에러가 발생할 것이 확실시 되므로 이 행에 위치 브레이크 포인트를 설정하고 실행해보면 다음과 같이 에러가 발생하는 것을 확인 할 수 있습니다.

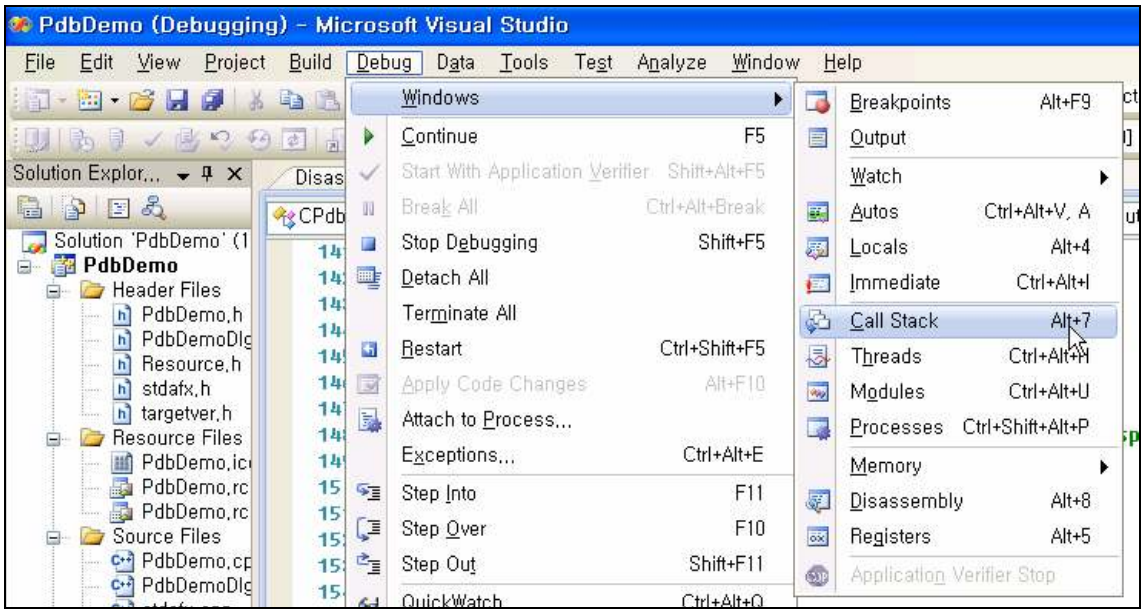


내용을 좀 보면 PdbDemo.exe파일이 실행되던 중 0x77CFB0E9 번지의 명령이 0번지에 쓰기를 시도를 해서 예외처리가 되었다고 나옵니다. 여기서 “Continue” 버튼을 눌러 계속 실행 하려 해도 더 이상 실행은 불가능 합니다.

우선 실행을 중단하기 위해 “Break” 버튼을 클릭하면 다음 그림과 같은 모습을 볼 수 있습니다.



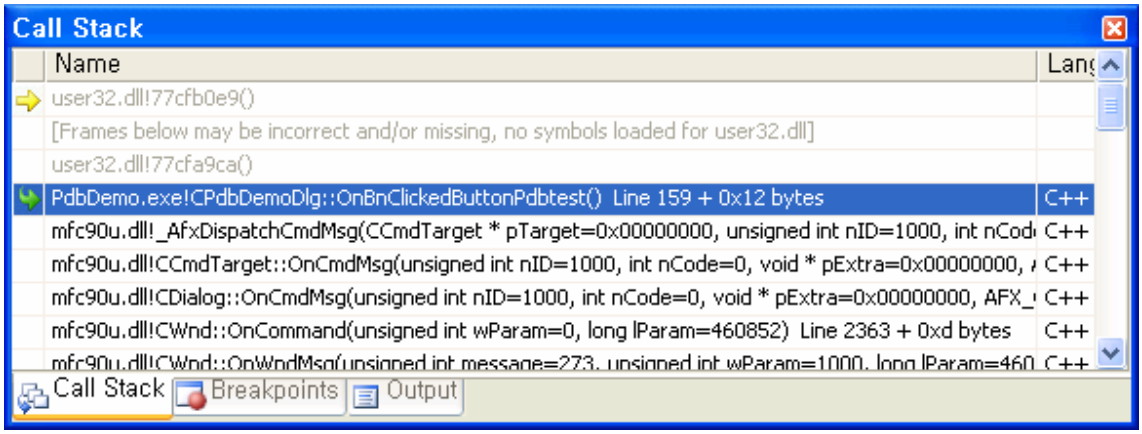
에러를 발생시킨 **0x77CFB0E9 번지는 USER32.DLL 라이브러리가 제공한 어떤 함수** 임을 알 수 있습니다. USER32.DLL은 윈도우 운영체제 라이브러리 입니다. 만일 이 라이브러리에 대한 PDB 파일이 있었다면 보기 어려운 주소가 아니라 구체적인 함수 이름으로 출력되었을 것입니다. 이 부분에 대해서는 잠시 후에 다시 언급하기로 하겠습니다. 지금은 이 정보가 출력된 윈도우가 어떤 것인지에 대해 알아 보겠습니다. 이 윈도우는 **호출 스택(Call Stack) 디버그 윈도우** 입니다.



이 호출 스택 정보를 보면 함수 호출의 상관 관계에 대해 알 수 있습니다. C, C++로 작성된 프로그램은 함수(main(), WinMain() 함수)로 시작해 함수로 끝난다는 것을 이미 배웠을

것입니다. 그리고 그 함수 내부에 선언된 지역변수들은 함수 호출 시 자동으로 생성되는 스택에 그 영역이 설정되므로 자동변수라고 한다는 것도 배웠을 것입니다. 이 때문에 재귀호출과 같은 것들이 가능합니다. 함수가 리턴하면 각 함수 호출 시 생성되었던 스택은 사라지게 됩니다. 그러므로 호출 스택 맨 위에 보이는 것은 현재 실행중인 즉 맨 마지막에 호출된 함수가 됩니다.

그렇기 때문에 이 정보를 알면 문제가 발생한 지점(함수)을 쉽게 찾아낼 수 있습니다. 다음 그림에서 알 수 있듯 OnBnClickedButtonPdbtest()에서 에러가 발생하였습니다. 그리고 친절하게도 소스 코드 159번 행이라는 것까지 알려줍니다.



이 정도의 정보를 알 수 있다면 개발자는 에러가 발생한 위치를 판별해 내는데 아무런 문제가 없을 것입니다. 한 가지 아쉬운 것은 최종적으로 에러를 발생시킨 0x77CFB0E9 번지가 어떤 것인지 구체적으로 알지 못했다는 것입니다.

이것이 무엇인지 구체적으로 알아내기 위해 골치 아픈 리버스 엔지니어링을 할 수도 있을 것입니다. 그러나 굳이 그럴 필요가 없습니다. 왜냐하면 MS가 윈도우 운영체제의 심볼 파일을 제공해주기 때문입니다. 아래 링크를 찾아가시면 윈도우 심볼 설치파일을 다운로드 받을 수 있습니다.

http://www.microsoft.com/whdc/devtools/debugging/symbolpkg.mspx

주의 할 것은 자신이 사용하고 있는 운영체제에 맞는 것을 다운로드 받으셔야 한다는 것입니다. 한글 Windows XP Professional SP2를 사용 중 이라면 당연히 XP SP2에 해당하는 심볼을 다운로드 받아야 합니다. 그런데 그 부분을 잘 보시면 다시 Retail 버전과 Checked 버전으로 나뉘는 것을 볼 수 있습니다.

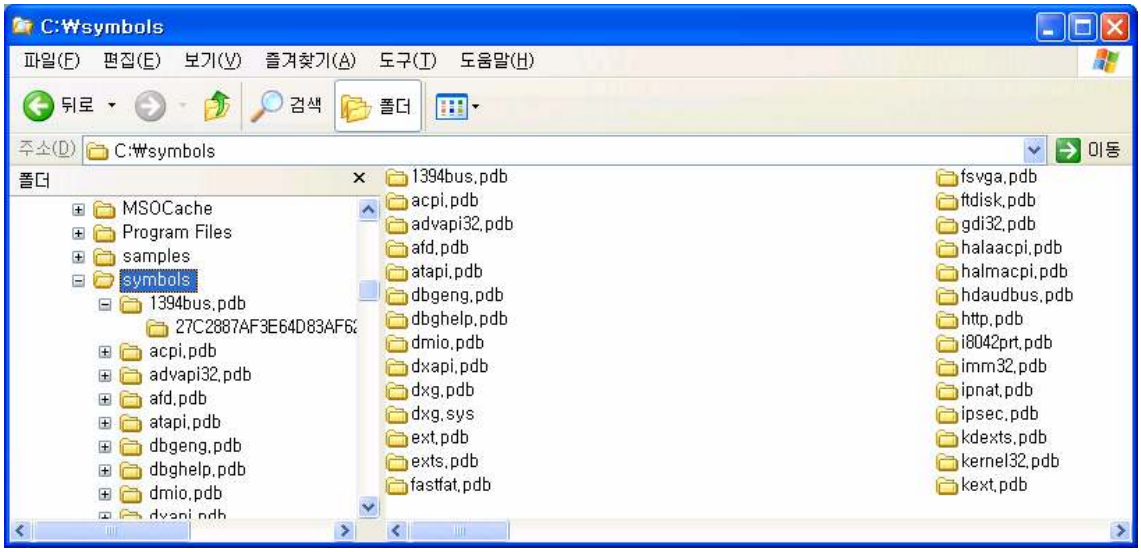
Reduced download size: Windows XP Service Pack 2

These packages are a smaller download size than the full set of Windows XP with Service Pack 2 symbols. They contain only the symbols for the files that ship with the Windows XP Service Pack 2. If you already have the Windows XP symbols installed, you can install these to the same location and you will have a full set of Windows XP with Service Pack 2 symbols.

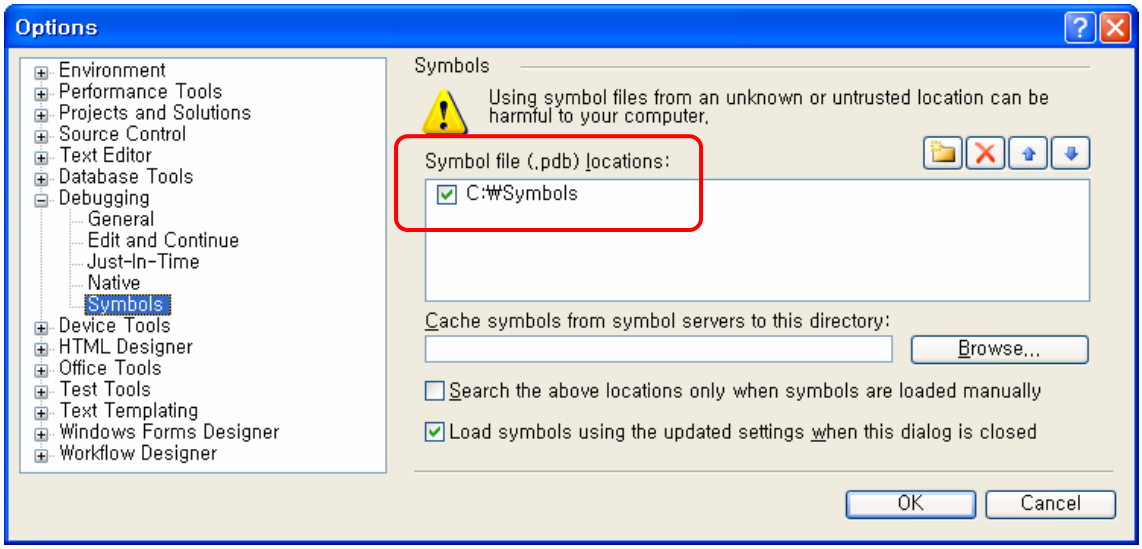
- Windows XP Service Pack 2 x86 retail symbols, all languages (File size: 145 MB - Most customers want this package.)
- Windows XP Service Pack 2 x86 checked symbols, all languages (File size: 132 MB)

Checked 버전은 Debug 버전을 의미합니다. 운영체제도 프로그램입니다. 따라서 릴리즈 모드로 빌드된 버전과 디버그 모드로 빌드된 버전이 있을 수 있습니다. 특수한 상황이 아니라면 모두 Retail 버전을 설치하였을 것입니다. 그러므로 반드시 Retail 버전에 대한 심볼을 다운로드 받아서 설치하여야 할 것입니다.

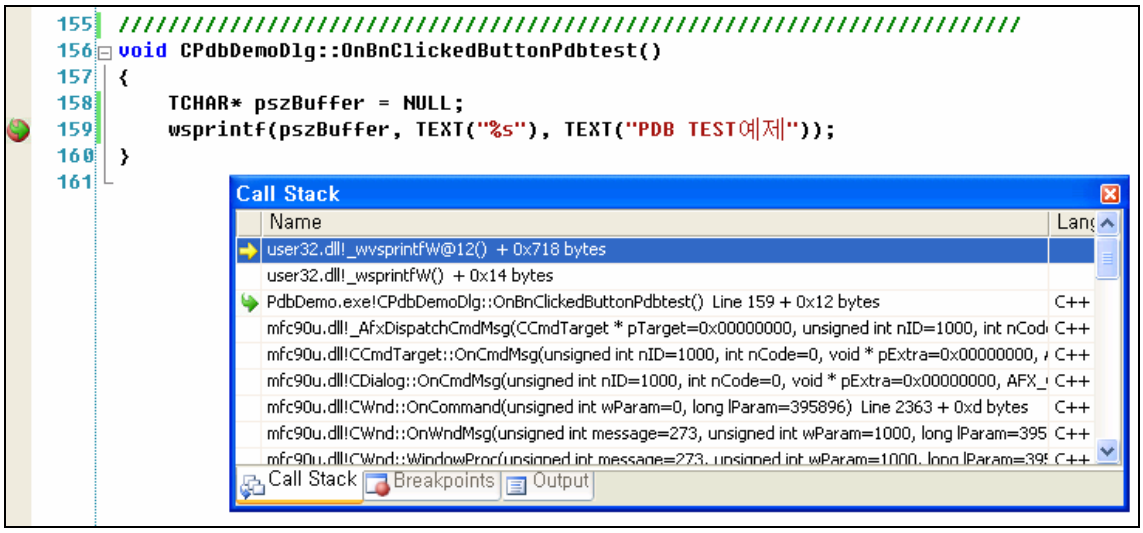
이 심볼을 받아서 설치하면 다음 그림과 같은 모습을 윈도우 탐색기로 확인할 수 있습니다. 많은 PDB 파일들이 보일 것입니다. 찾아보시면 USER32.DLL에 대한 PDB도 볼 수 있습니다.



심볼을 설치했다고 당장 Visual C++ 디버깅 화면에서 전과 달리 함수 이름으로 호출 스택 정보가 출력되는 것은 아닙니다. 이 심볼들을 Visual C++ 디버거가 인식할 수 있도록 설정을 변경하여야 합니다. 그러기 위해서 “Tools”의 “Options”메뉴를 선택하고 다음과 같이 심볼이 설치된 경로를 등록합니다. 이 경로는 각자 다를 수 있으므로 심볼이 설치된 절대 경로를 확인하여 적절한 경로를 명시하여야 할 것입니다.



윈도우 운영체제 심볼 설치 및 등록을 마친 후 앞서의 에러 코드를 실행하여 호출 스택을 보면 좀더 구체적인 정보가 출력되는 것을 알 수 있습니다.



굳이 윈도우 심볼을 설치하지 않아도 디버깅을 할 수 있습니다. 하지만 설치 및 등록해주시면 좀더 상세하게 원인 분석이 가능합니다. 그러므로 가급적 운영체제 심볼 설치를 권장합니다.

데이터 브레이크 포인트와 메모리

데이터 브레이크 포인트는 위치 브레이크 포인트와 달리 **특정 주소 공간의 값이 변경되면 동작하는 브레이크 포인트** 입니다. 따라서 프로그램이 지정한 메모리의 값을 바꾸는 코드가 실행되지 않는다면 데이터 브레이크 포인트는 동작하지 않습니다. 반대로 지정한 메모리의 값이 바뀐다면 언제든 동작합니다. 이 글만 봐서는 이러한 브레이크 포인트를 어떻게 활용하여야 할지 판단이 서지 않을 수도 있습니다. 그러나 데이터 브레이크 포인트를 잘 활용하시면 위치 브레이크 포인트로는 해결할 수 없는 문제를 해결하거나 논리적인 버그를 손쉽게 해결할 수 있습니다.

그리고 대부분의 초보 개발자들은 할당 연산자를 의심하지 않습니다. 또한 배열을 잘못 다뤄서 생기는 문제를 쉽게 해결하지 못합니다. 여기 잘못 다뤘다는 말은 배열의 경계를 벗어난 읽기/쓰기 문제를 말하는 것입니다. 그러므로 **개발자는 항상 메모리의 변화를 잘 관찰하면서 디버깅을 해야 합니다.**

우선 데이터 브레이크 포인트에 대해 자세히 알아보도록 하겠습니다. 앞서 언급한 위치 브레이크 포인트의 설정 기준은 실행 코드였습니다. 그렇기 때문에 별다른 조건이라는 것이 있을 필요성이 없었던 것입니다. 그러나 데이터 브레이크 포인트는 메모리 값을 기준으로 합니다. 그 메모리에는 어떤 값이 들어가게 될지 정해진 것이 아니므로 매우 유동적이라 할 수 있습니다.

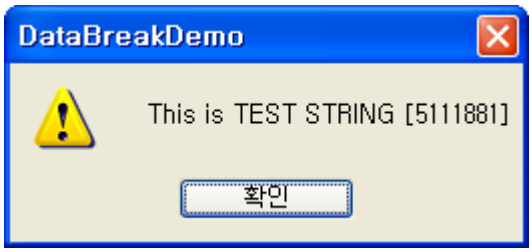
아래 코드를 빠르게 읽고 AfxMessageBox() 함수가 어떤 문자열을 출력할 지 예상해 보시기 바랍니다.

```
155 ////////////////////////////////////////////////////////////////////////////
156 TCHAR g_szName[16];
157 int g_nResult = 0;
158
159 ////////////////////////////////////////////////////////////////////////////
160 void CDataBreakDemoDlg::OnBnClickedButtonDatabreakpoint()
161 {
162     CString tmp = _T("");
163     g_nResult = 0;
164     wsprintf(g_szName, TEXT("%s"), TEXT("This is TEST STRING"));
165
166     tmp.Format(TEXT("%s [%d]"), g_szName, g_nResult);
167     AfxMessageBox(tmp);
168 }
169
```

깊이 생각하지 않고 보면 무조건 “0”이 출력되어야 맞을 것입니다. 혹시 아무리 자세히 들여다 봐도 “0”이 출력될 것이라고 판단 되어도 너무 걱정하시거나 스스로를 폄하시기지

는 마시기 바랍니다. 반대로 답을 어느 정도 맞추신 분들은 구체적으로 어떤 값이 출력되는지 생각해 보시기 바랍니다. 그리고 그 값이 빌드모드에 상관없이 늘 같은 값인지도 고민하여야 합니다.

이 예는 정말 중요한 예 입니다. 지금은 코드가 짧기 때문에 문제점이 확연하게 보일 수 있지만 코드가 길어지고 프로그램이 커졌을 때는 전혀 다른 이야기가 됩니다. 아무리 경험 많은 개발자라 하더라도 실수 할 수 있기 때문입니다. 이 코드를 빌드하여 결과를 확인하면 다음과 같습니다.



디버그 모드

릴리즈 모드

만일 앞서 코드 예에서 전역변수 선언 부분을 빼고 확인한다면 분명 “0”을 출력하고자 했음을 확실히 알 수 있습니다. 다음의 코드를 다시 살펴보시기 바랍니다.

```
159 ////////////////////////////////////////////////////////////////////////////
160 void CDataBreakDemoDlg::OnBnClickedButtonDatabreakpoint()
161 {
162     CString tmp = _T("");
163     g_nResult = 0;
164     wsprintf(g_szName, TEXT("%s"), TEXT("This is TEST STRING"));
165
166     tmp.Format(TEXT("%s [%d]"), g_szName, g_nResult);
167     AfxMessageBox(tmp);
168 }
169
```

163번 행의 코드에서 g_nResult를 0으로 확실하게 초기화 했습니다. 그렇기 때문에 166번 행의 코드가 수행되면 g_nResult 값이 0일 것이라 예상할 수 있습니다. **g_szName의 선언이 한 눈에 보이지 않기 때문**에 무시하고 넘어가기 쉽습니다.

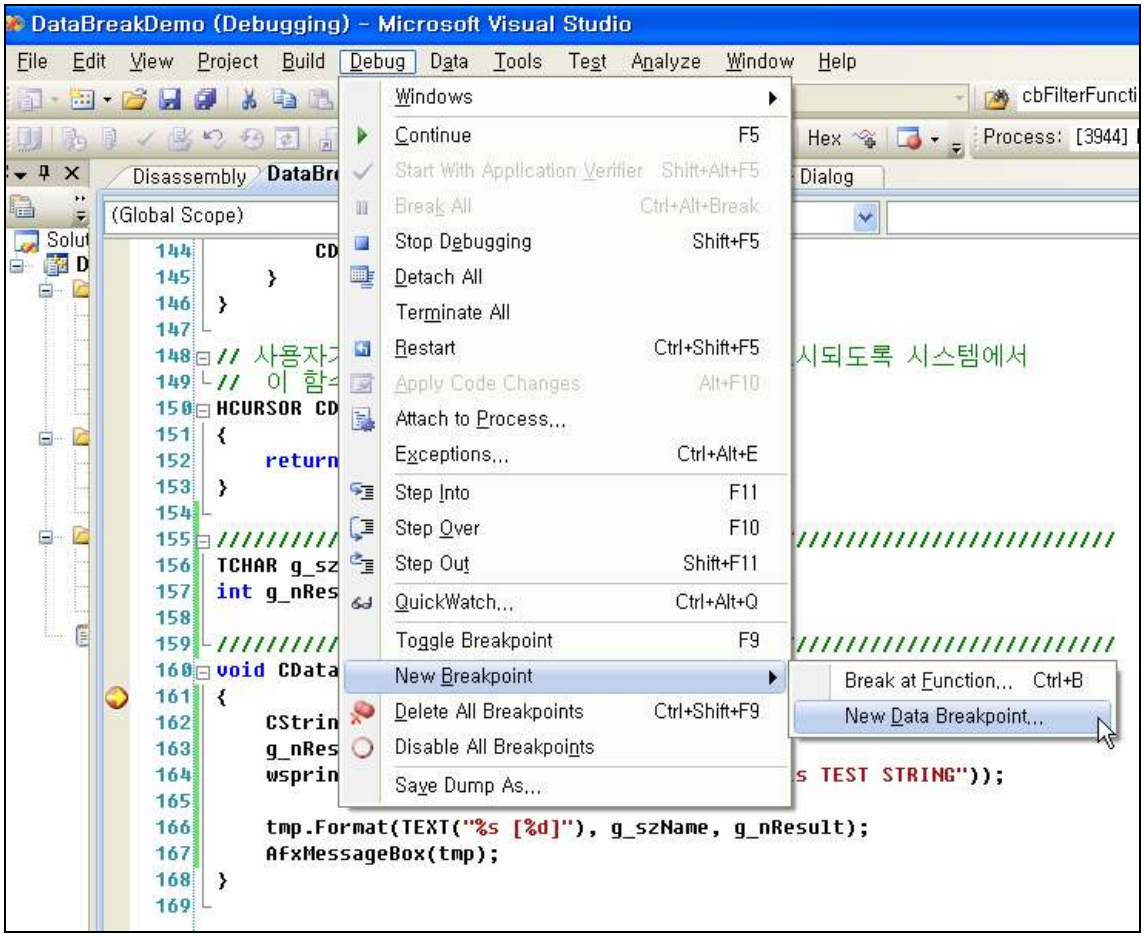
중요한 것은 이 에러가 발생한 원인지 무엇인지 규명하고 어떻게 해결하여야 할지 알아야 합니다. 여러 해결 방법이 있겠으나 이 경우 제가 가장 추천하고 싶은 방법은 **데이터 브레이크 포인트를 이용하는 것**입니다. 예제 코드의 결과적인 문제는 g_nResult 값이 0이 아닌 다른 값으로 채워졌다는데 있습니다. 사실 지금은 코드가 매우 짧기 때문에 164, 166번 행

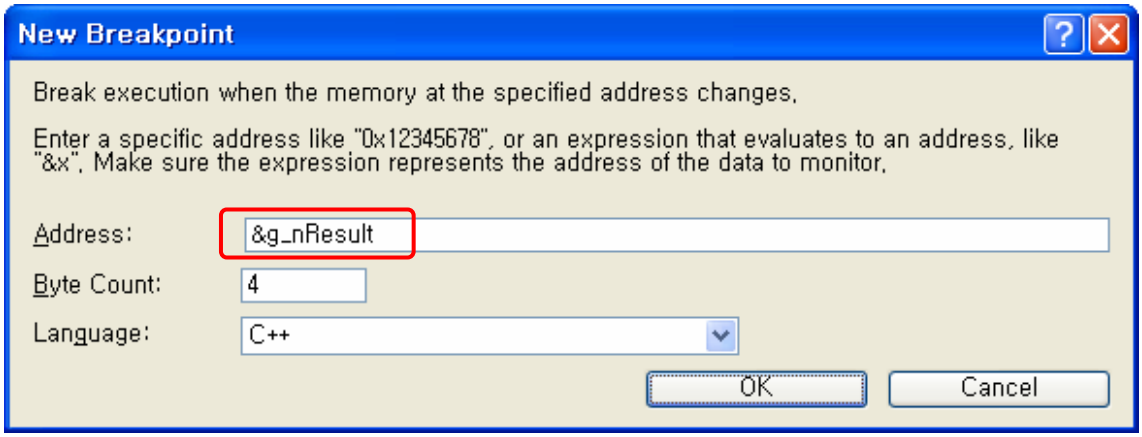
의 코드가 원인일 것이라고 직관적으로 예측할 수 있습니다.

이를 좀더 명확히 이해하기 위해 OnBnClickedButtonDatabreakpoint() 함수에 위치 브레이크 포인트를 설정하고 디버그 모드로 빌드한 후 디버그 모드로 실행합니다. 그러면 다음과 같이 위치 브레이크 포인트가 동작하는 것을 볼 수 있습니다.

```
159 ////////////////////////////////////////////////////////////////////////////
160 void CDataBreakDemoDlg::OnBnClickedButtonDatabreakpoint()
161 {
162     CString tmp = _T("");
163     g_nResult = 0;
164     wsprintf(g_szName, TEXT("%s"), TEXT("This is TEST STRING"));
165
166     tmp.Format(TEXT("%s [%d]"), g_szName, g_nResult);
167     AfxMessageBox(tmp);
168 }
169
```

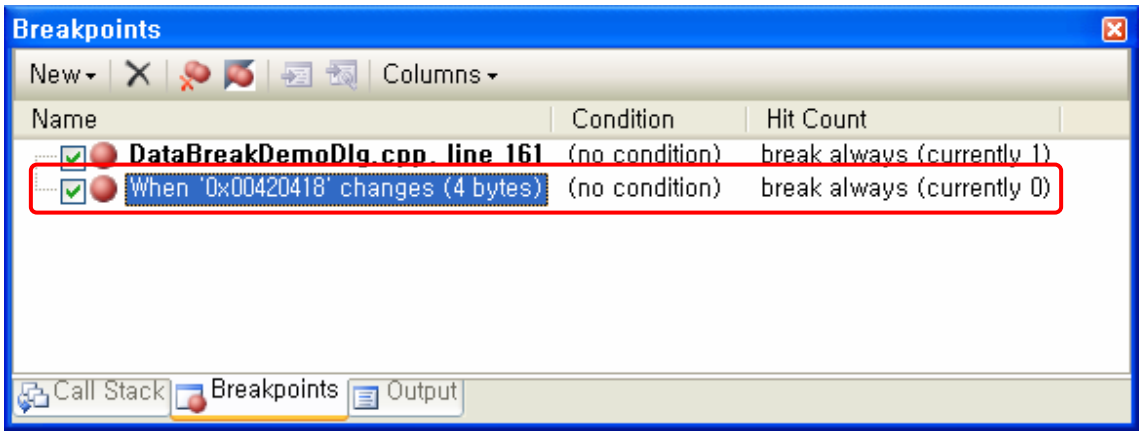
이와 같이 실행이 일시 중지된 상태에서 데이터 브레이크 포인트를 설정합니다.



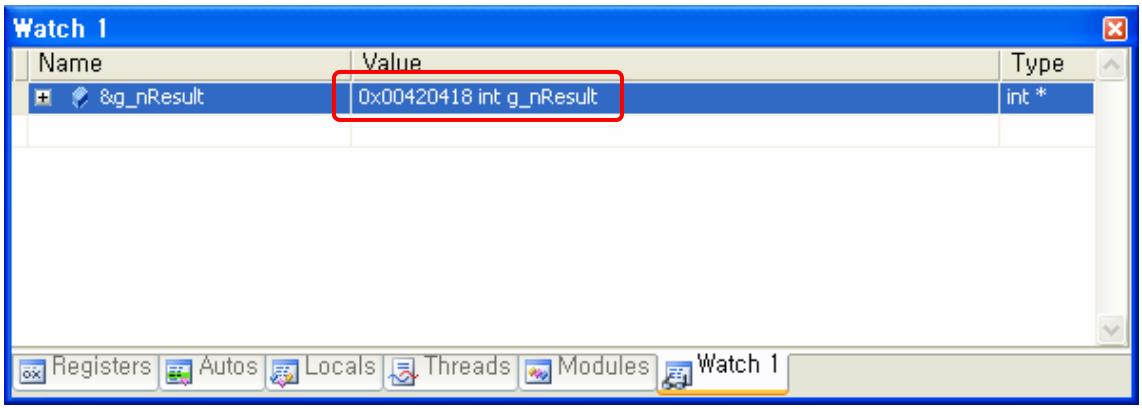


Address 항목에 "&g_nResult"라고 입력합니다. 말 그대로 어떤 주소를 입력하면 됩니다. 이 주소라는 것이 32비트 시스템에서는 당연히 4바이트 데이터입니다. 그러므로 바이트 카운트 값을 수정할 필요 없이 그냥 두면 됩니다. 그러면 전역변수 g_nResult의 값이 바뀌는 순간에 브레이크 포인트가 작동하게 됩니다. 만일 g_nResult의 값이 10인데 다시 10을 할당할 경우에는 값이 변하지 않으므로 동작하지 않습니다. 이점 주의하시기 바랍니다.

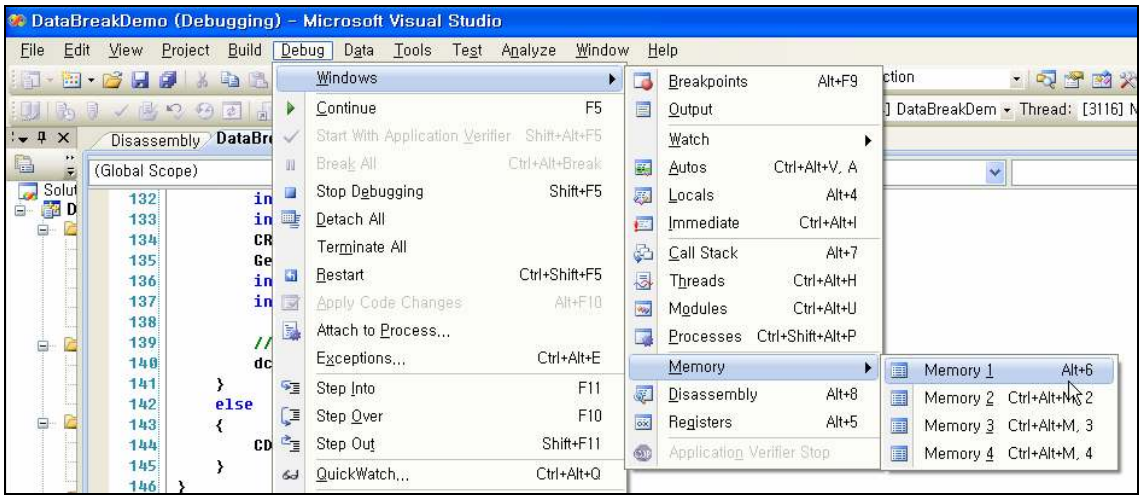
아무튼 이러한 방법으로 데이터 브레이크 포인트를 추가하면 다음과 같이 브레이크 포인트 관리 윈도우에 새로운 항목이 등록됩니다.

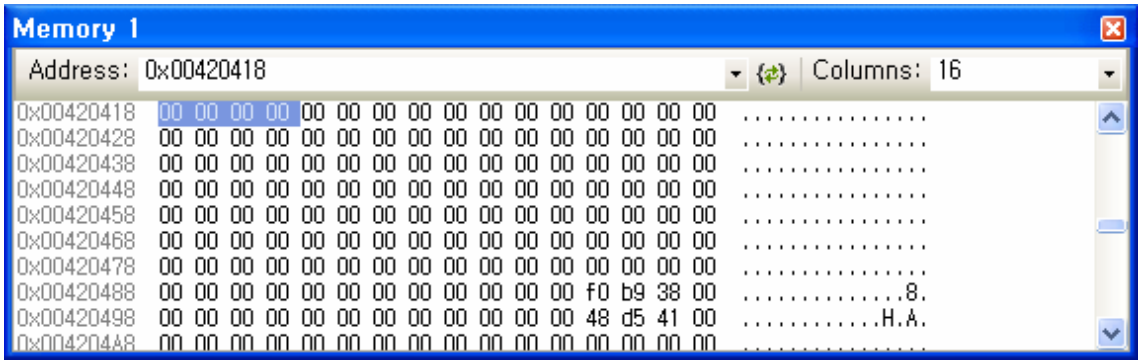


그런데 &g_nResult 라는 문자열은 없고 어떤 16진수 값(0x00420418)이 들어있습니다. 이 값은 g_nResult 변수의 주소값 입니다. 조사식 윈도우(Watch)에 "&g_nResult" 라고 입력해보면 확인할 수 있습니다.



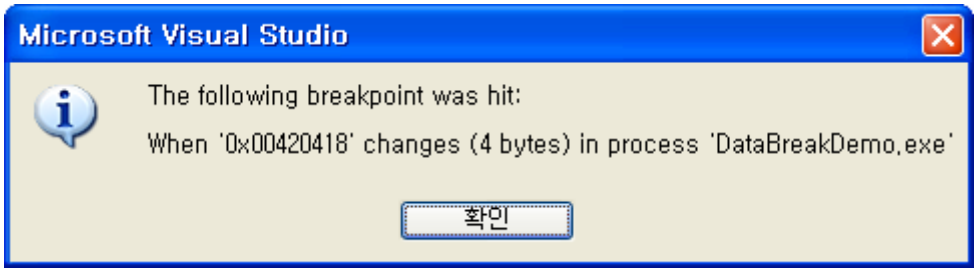
이와 같이 특정 변수에 대한 주소를 확인하였다면 메모리 윈도우를 띄우고 해당 메모리의 값들을 직접 보면서 확인합니다.





주소창에 g_nResult의 주소를 직접 입력하여도 되고 조사식 윈도우에 등록한 항목을 클릭하여 메모리 윈도우에 드롭해도 됩니다. 이러한 메모리 윈도우는 최대 4개까지 동시에 띄워서 볼 수 있습니다. 중요한 것은 **0x00420418 번지는 g_nResult 변수의 주소**값이라는 사실입니다. 이점을 확실히 한 상태에서 예제 코드를 디버그 모드로 계속 실행합니다.

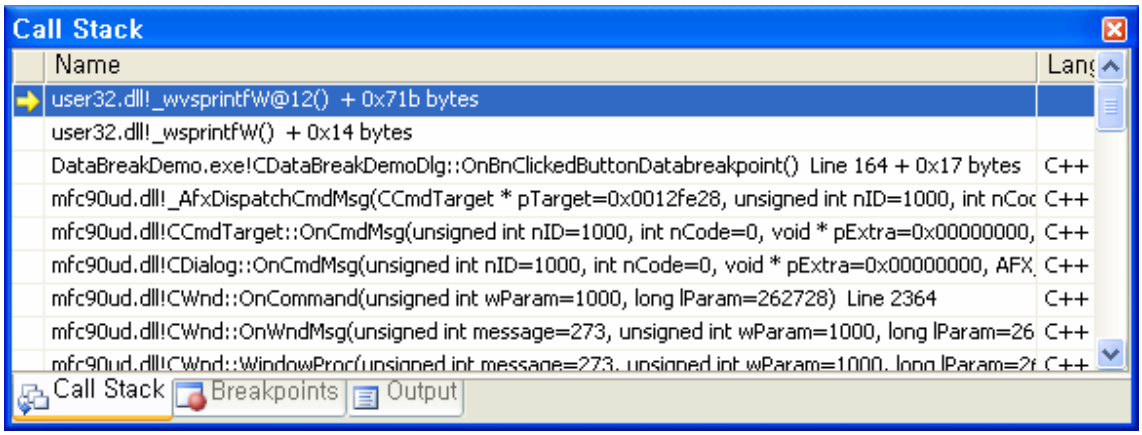
그러면 다음과 같이 데이터 브레이크 포인트가 동작했다는 메시지를 확인할 수 있습니다. 참고로 한글판 Visual C++ 2008에서는 "적중했다"라는 메시지가 나옵니다.



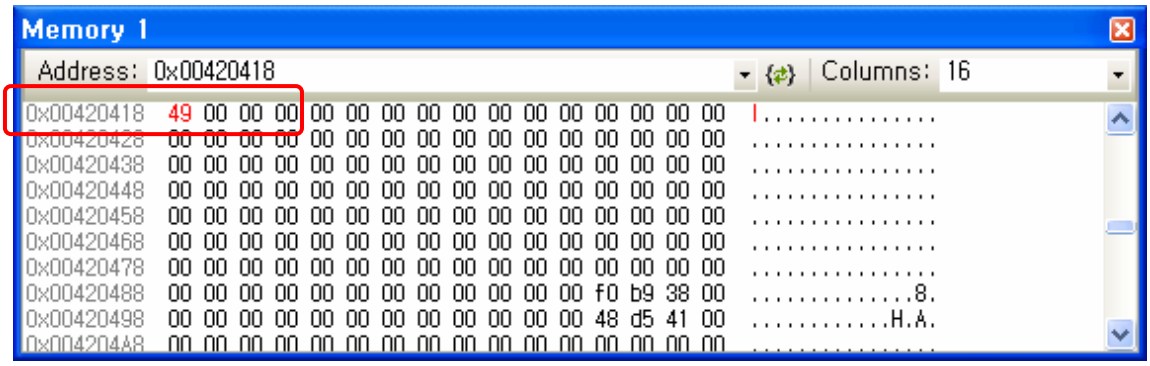
이 메시지를 확인하면 디스어셈블리 윈도우가 나타나면서 어느 번지의 어떤 어셈블리 명령을 수행하다가 값이 변경되었는지 구체적으로 보여주게 됩니다.

```
77CFB0DC  cmp         dword ptr [ebp-10h],0
77CFB0E0  je          77CFAA25
77CFB0E6  mov         cx,word ptr [edi]
77CFB0E9  mov         word ptr [esi],cx
77CFB0EC  inc         esi
77CFB0ED  inc         esi
77CFB0EE  inc         edi
77CFB0EF  inc         edi
77CFB0F0  dec         dword ptr [ebp-10h]
77CFB0F3  jmp         _wvsprintfW@12+322h (77CFB0D7h)
```

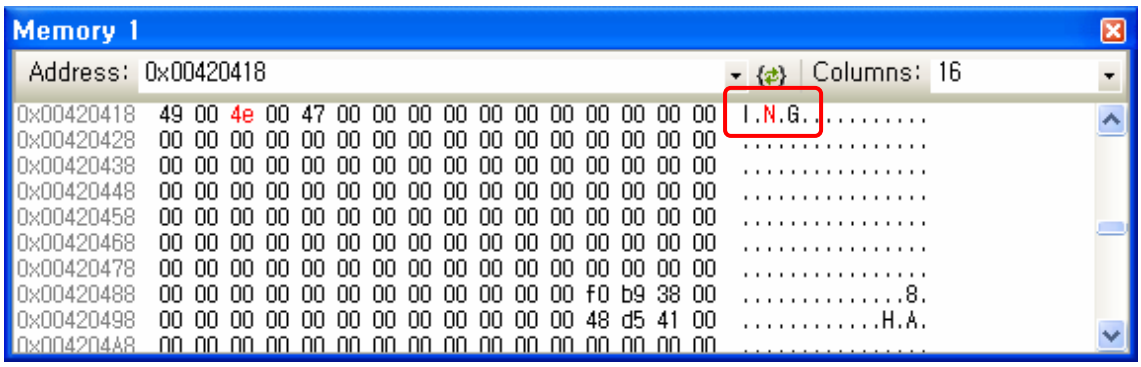
이 어셈블리 코드만 보고 어떤 내용인지 즉시 판별하기는 어렵습니다. 그러므로 이 시점의 호출 스택 정보를 확인합니다. 그러면 구체적으로 어떤 함수인지 알 수 있습니다. 다음 그림에서 보면 wsprintf() 함수 때문임을 알 수 있습니다.



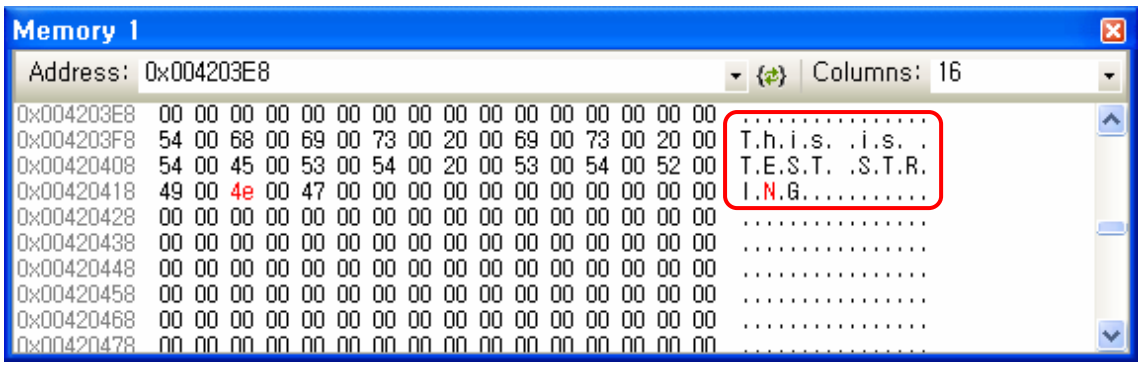
그리고 좀더 정확히 하기 위해 메모리 윈도우의 내용을 확인해 봅니다. 그러면 변경된 메모리의 내용이 붉은 색으로 표시됨을 알 수 있습니다. 다음 그림은 변경된 g_nResult 변수의 메모리 내용을 보이고 있습니다.



여기 보이는 값은 16진수 값입니다. 그러므로 49가 아니라 십진수 107이 되는 것입니다. 현재 디버그 모드로 실행 중 이므로 **F10** 키를 눌러 어셈블리 코드 단위로 계속 실행하도록 합니다. 그러면 다음과 같이 계속 값이 변하는 모습을 볼 수 있습니다.



이 때 메모리 윈도우 오른쪽에 **ASCII 코드영역**을 잘 보시기 바랍니다. 익숙한 유니코드 문자열이 보입니다. 메모리 윈도우의 내용을 위로 몇 줄 스크롤 해보시면 보다 명확한 이유를 알 수 있게 됩니다.



변수는 기억공간입니다. Int형 기억공간이라 해서 문자열을 쓸 수 없는 것이 아닙니다. 반대로 문자열을 저장하기 위한 버퍼에 Int 값이나 주소를 저장 못할 이유는 없습니다. 예제 코드의 논리적 결함은 wsprintf() 함수가 g_szName 버퍼에 “This Is TEST STRING”이라는 문자열을 쓰면서 배열의 경계를 벗어나 다른 변수(g_nResult)의 영역을 덮어 쓰는 것에 있습니다. 이와 같이 배열의 경계를 벗어난 읽기/쓰기 동작은 예기치 못한 치명적인 오류를

만들어 냅니다.

사실 프로그램이 항상 똑 같은 시점에 비정상 종료되면 이는 반가운 일입니다. 해당 위치의 코드를 찾아 수정하면 그만이기 때문입니다. 그러나 평상시에는 잘 돌아가다가 가끔씩 이상한 결과를 내거나 하면 그런 문제는 해결하기가 쉽지 않습니다. 특히 초보 개발자에게는 심각한 수준의 스트레스를 유발하는 원인이 됩니다. 그러므로 지금 설명한 사항을 잘 기억했다가 업무에 활용하시기 바랍니다. 데이터 브레이크 포인트를 활용할 수 있느냐 그렇지 않느냐는 1주일씩 걸려 문제를 해결할 것이냐 아니면 1시간 이내에 문제를 해결할 것이냐를 가르는 중요한 디버깅 기법입니다.

선배로써 한마디!

할당 연산자라 하더라도 가끔은 의심할 수 있어야 합니다. 그리고 메모리 윈도우 보는 것을 두려워하지 마시기 바랍니다. 16진 데이터를 직접 바라 보는 일이 처음에는 분명 낯설고 어려운 일이지만 익숙해지고 나면 프로그램 코드를 바라보는 눈이 달라질 것입니다. 매트릭스라는 영화를 보셨습니까? 그렇다면 주인공 Neo가 다시 부활하여 바라본 세상은 검은색 안경을 쓴 요원들이 아니라 전부 코드였습니다. 16진수를 보는데 익숙해지시기 바랍니다. 컴퓨터와 C,C++ 코드에 대한 이해가 달라질 것이며 언젠가는 주인공 Neo처럼 새로운 세상을 보게 될 것입니다.

조건 데이터 브레이크 포인트

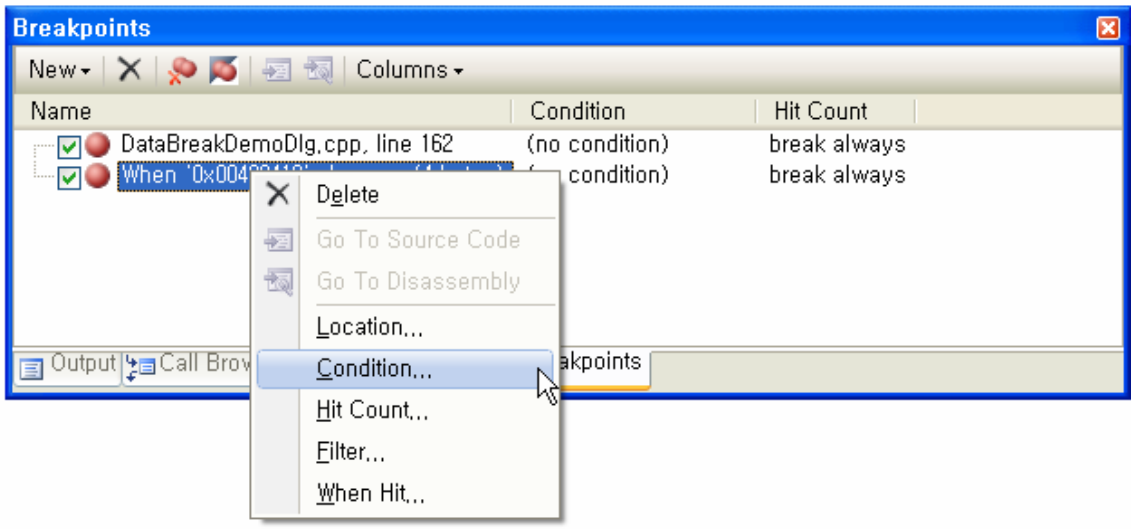
데이터 브레이크 포인트를 잘 쓰는 단계가 되면 또 불편함을 느끼게 됩니다. 이유는 너무 자주 브레이크 포인트가 동작하기 때문입니다. 만일 10000번 정도 루프를 도는 코드에서 특정 코드의 값이 변하는 것을 확인하고자 데이터 브레이크 포인트나 위치 브레이크 포인트를 사용하면 귀찮기 그지 없습니다. 확인할 데이터가 9000번쯤 루프를 돌고 난 후에야 확인 되는 것이라면 이는 정말 답답한 일이 아닐 수 없습니다.

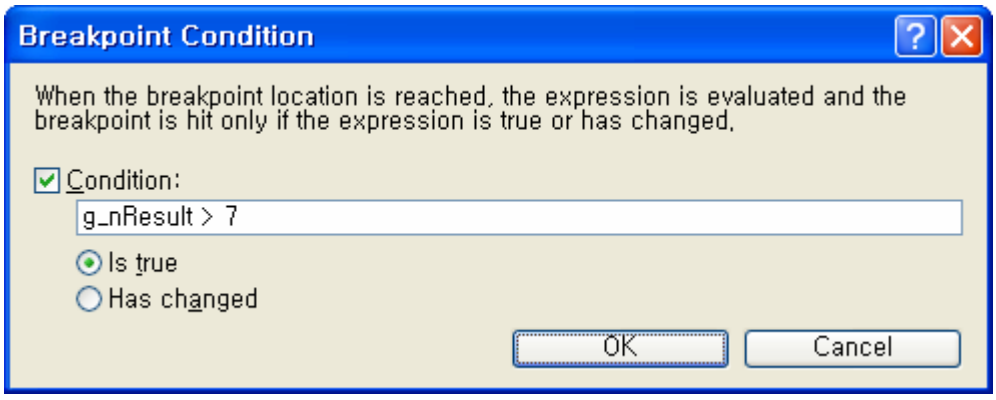
물론 if문 하나 집어넣고 소스 코드를 수정해서 해결하는 손쉬운 방법도 있습니다. 아마도 다수의 개발자가 그런 방법을 택할 것입니다. 그러나 루프처럼 정해진 규칙에 따라 값이 증가하는 경우가 아니라면 어떻게 할까요? 임의의 값에 대응해서 브레이크 포인트를 걸어야 하는 경우는 단순 데이터 브레이크 포인트 만으로는 한계가 있습니다. 그래서 등장한 것인 조건 데이터 브레이크 포인트 입니다.

조건 데이터 브레이크 포인트는 데이터 브레이크 포인트보다 향상된 기능을 제공합니다. 다음의 코드 예를 보시기 바랍니다.

```
171 ////////////////////////////////////////////////////////////////////////////
172 void CDataBreakDemoDlg::OnBnClickedButtonConditionbreak()
173 {
174     srand((unsigned)time( NULL ));
175     g_nResult = rand() % 10;
176 }
177
```

g_nResult에 7 이상의 값이 할당될 때만 브레이크 포인트가 동작하도록 설정할 수 있다면 값이 바뀔 때마다 동작하는 데이터 브레이크 포인트보다 편리하게 디버깅을 할 수 있을 것입니다. 그러기 위해서 이미 등록된 데이터 브레이크 포인트를 조건 데이터 브레이크 포인트로 수정합니다.





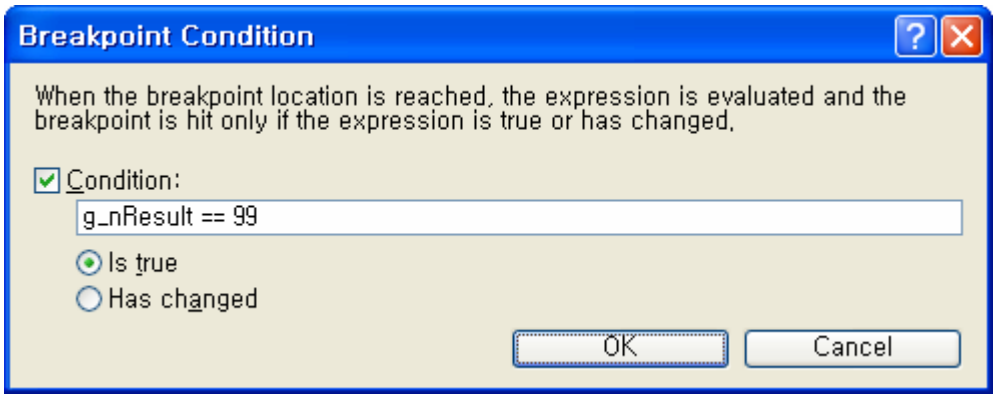
지금 설정한 내용은 굳이 설명하지 않아도 충분히 이해할 수 있을 것입니다. 조건 부분에는 **if문 괄호 안에 쓸 수 있는 조건과 동일**하다고 보시면 됩니다. 이 조건이 참이 되면 조건

브레이크 포인트가 동작합니다. 즉 할당하는 값이 rand() 리턴하는 임의의 값을 10으로 나눈 나머지가 7보다 큰 경우에만 브레이크 포인트가 동작합니다. 이와 같은 조건 데이터 브레이크를 활용하면 임의의 값이나 반복문 내에서 디버깅 해야 할 때 원하는 시점을 신속하게 잡아 낼 수 있습니다.

이러한 조건 데이터 브레이크 기능이 가장 잘 활용되는 경우는 멀티스레드 환경이 아닐까 생각합니다. 다음의 코드 예를 봅시다.

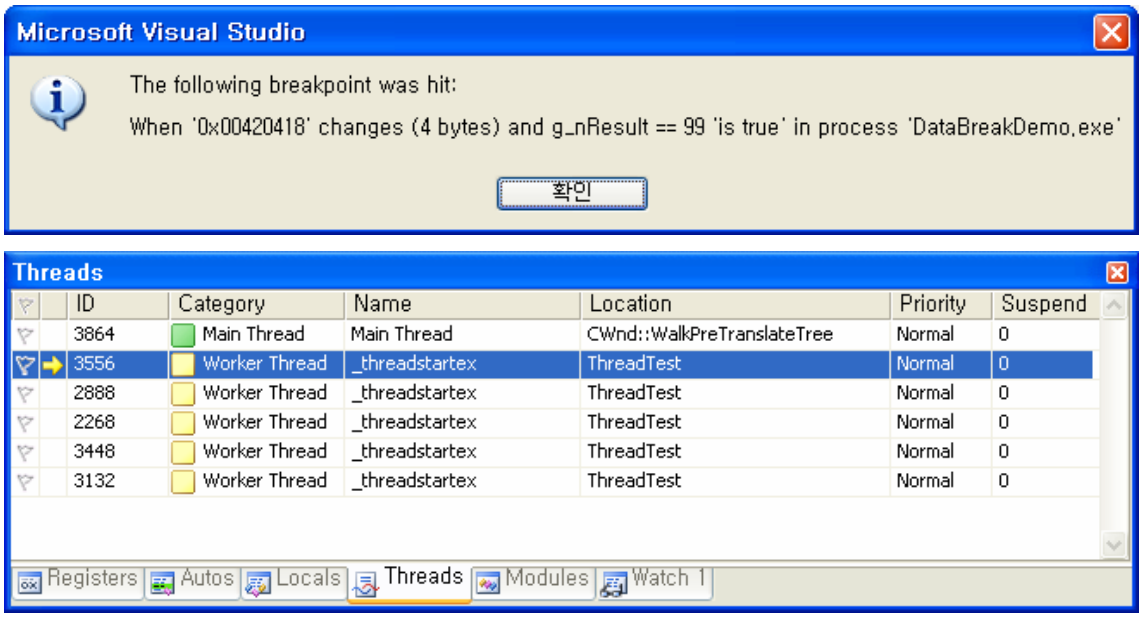
```
179 ////////////////////////////////////////////////////////////////////////////
180 UINT ThreadTest(LPVOID pParam)
181 {
182     srand((unsigned)time( NULL ));
183     for(int i = 0; i < 1000; ++i)
184     {
185         g_nResult = rand() % 100;
186     }
187
188     return 0;
189 }
190
191 ////////////////////////////////////////////////////////////////////////////
192 void CDataBreakDemoDlg::OnBnClickedButtonMultithreadbreak()
193 {
194     for(int i = 0; i < 5; ++i)
195     {
196         AfxBeginThread(ThreadTest, NULL);
197     }
198 }
199
```

ThreadTest() 함수는 총 5개의 스레드로 실행됩니다. 이 함수의 내용을 요약하면 앞서 예와 유사하게 g_nResult 전역변수에 0~99사이의 임의의 값을 1000번 루프를 돌면서 할당하고 있습니다. 설정된 데이터 브레이크 포인트에 다음과 같이 조건을 주고 디버그 모드로 실행해 보겠습니다.



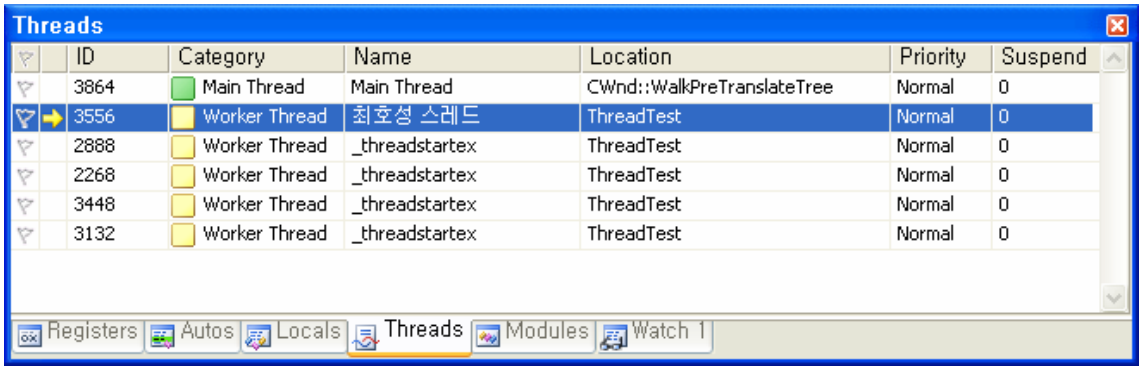
5개의 스레드 중 하나가 언젠가는 g_nResult에 99를 할당할 것입니다. 이 때, “스레드”

윈도우를 살펴보면 다음과 같이 구체적으로 어떤 스레드가 어느 시점에 값을 변경하였는지 나타납니다.



조건을 만족하지 않더라도 스레드 스위칭이 발생하면 그 시점에서 스레드의 진행이 멈추기도 합니다.

각각의 스레드는 고유한 ID 값을 가지고 있으므로 이를 식별하는 데는 무리가 없습니다. 그러나 좀더 직관적으로 디버깅 하고자 한다면 스레드 이름을 변경하여 변화를 확인할 수도 있습니다. 다음 그림은 3556번 스레드의 이름을 "최호성 스레드"라고 변경한 예 입니다.



아무래도 사람은 숫자보다는 문자열이 좀더 눈에 잘 들어오므로 이런 기능을 활용해 보는 것도 좋겠습니다. 이 외에도 멀티스레드 환경에서의 디버그 기능이 Visual C++ 2008에서 많이 향상되었습니다. 그러한 부분들에 대해 다루는 것도 좋겠으나 이 정도의 지식만으로도 충분하리라 생각합니다. 나머지 부분들은 스스로 직접 실습해보면서 의미를 파악하시면 됩니다.

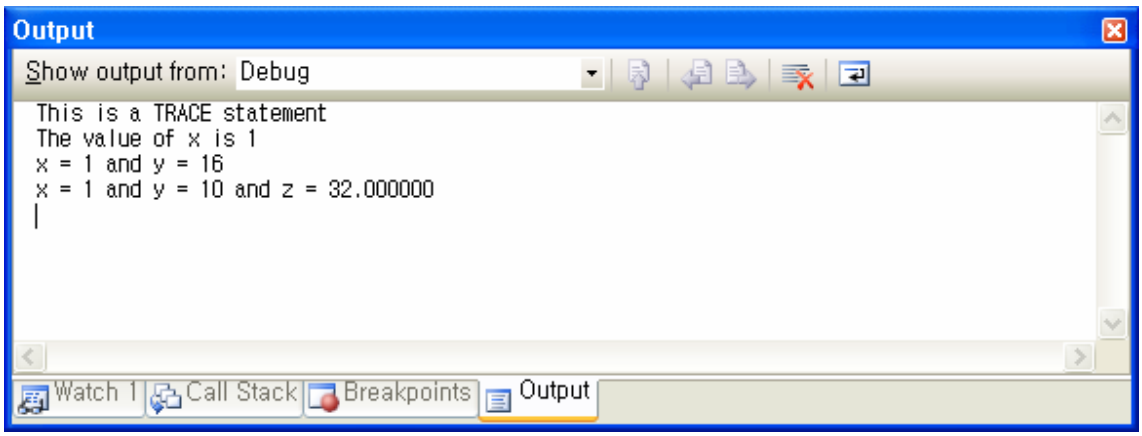
디버그 매크로와 함수

빌드모드를 디버그로 했을 때만 개발자가 볼 수 있는 정보들이 출력되도록 하기 위해 디버그 매크로가 오랜 시간 동안 활용되어 왔습니다. 매크로를 사용해서 얻을 수 있는 최대 장점은 **런타임에 실시간으로 프로그램의 동작 상황을 모니터링 할 수 있다는 것**입니다. 이 말은 브레이크 포인트를 걸어서 프로그램의 실행을 개발자가 임의로 중단하여 내용을 확인하는 것이 아니라 관찰해야 할 대상을 정해주고 프로그램의 흐름을 방해하지 않으면서 과정을 모니터링 할 수 있다는 뜻입니다.

다음의 코드는 Visual C++ 2008 도움말에 들어있는 TRACE 매크로 예제입니다. 사용방법은 매우 직관적이며 printf() 함수를 사용할 줄 아는 개발자라면 누구나 손쉽게 사용할 수 있습니다.

```
163 ////////////////////////////////////////////////////////////////////////////
164 void CDebugMacroDlg::OnBnClickedButtonTrace()
165 {
166     int x = 1;
167     int y = 16;
168     float z = 32.0;
169
170     TRACE( "This is a TRACE statement\n" );
171     TRACE( "The value of x is %d\n", x );
172     TRACE( "x = %d and y = %d\n", x, y );
173     TRACE( "x = %d and y = %x and z = %f\n", x, y, z );
174 }
175
```

이 코드를 디버그 모드로 빌드하고 실행하면 출력 윈도우에서 다음과 같은 메시지를 실시간으로 확인할 수 있습니다.



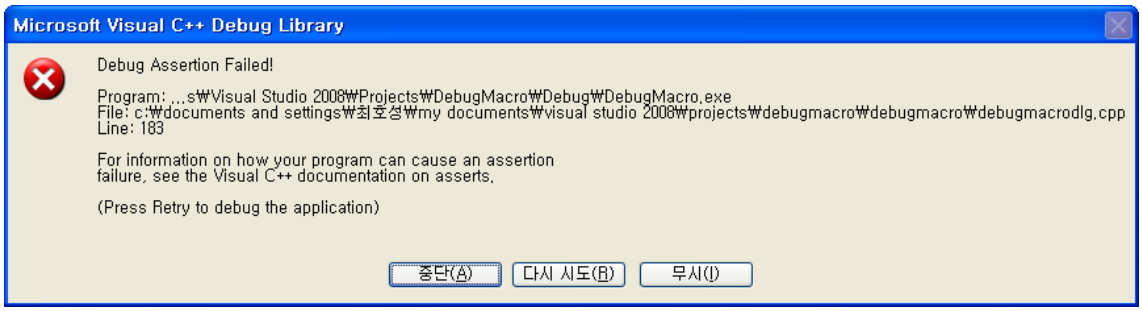
프로그램의 흐름을 멈추지 않기 때문에 경우에 따라서 이러한 디버깅 기법이 유용합니다. 특히 멀티스레드 프로그램을 디버깅 할 때 아주 유용합니다. 뿐만 아니라 윈도우 서비스 프

로그램을 개발할 때도 유용하게 활용할 수 있습니다. 사실 이 부분에 대해 더 자세히 설명해야 할 것들이 남아있으나 Visual C++ 2008을 사용하면 굳이 매크로를 활용하지 않아도 되므로 구체적인 언급은 이쯤 해두겠습니다.

ASSERT() 매크로 역시 비교적 자주 사용되는 디버그 매크로입니다. ASSERT 매크로는 글자 그래도 단언한다는 의미입니다. 다음의 코드 예를 살펴보시기 바랍니다.

```
177 ////////////////////////////////////////////////////////////////////////////
178 void CDebugMacroDlg::OnBnClickedButtonAssert()
179 {
180 //  TCHAR* pszBuffer = NULL;
181     TCHAR* pszBuffer = (TCHAR*)malloc(sizeof(TCHAR) * 128);
182
183     ASSERT(pszBuffer != NULL);
184     wsprintf(pszBuffer, TEXT("%s"), TEXT("TEST STRING"));
185
186     free(pszBuffer);
187 }
188
```

pszBuffer에 메모리를 할당하다가 에러가 발생할 가능성은 사실상 잘 없습니다. 이에 대해 제대로 에러처리를 하려면 사실 예외처리 구조를 적용해야 할 것입니다. 그러나 대부분의 개발자들이 메모리를 할당하는 연산에 대해 그런 처리를 늘 하지는 않습니다. 구조적인 예외처리를 하지 않는 대신 예제 코드처럼 할당된 메모리가 NULL 아님을 단언하는 매크로를 넣어주면 만에 하나 단언조건을 만족했을 때 다음과 같은 에러 메시지를 출력합니다.



코딩을 하는 동안 디버그 모드로 빌드된 파일을 실행했을 때 종종 볼 수 있는 그런 모양을 하고 있습니다. 지금까지 설명한 두 **디버그 매크로는 모두 디버그 모드일 때만 동작합니다. 릴리즈 모드로 빌드하면 코드 자체가 컴파일 타임에 무시되고 컴파일 됩니다.** 따라서 동일한 코드를 릴리즈 모드로 빌드하고 확인해보면 아무런 메시지도 경고 윈도우도 출력되지 않습니다.

이러한 디버그 매크로들과 더불어 반드시 알아 두어야 할 디버그 메시지 출력함수가 하나 있습니다. 바로 **OutputDebugString() 함수** 입니다. 다시 말씀 드리지만 이것은 매크로가 아

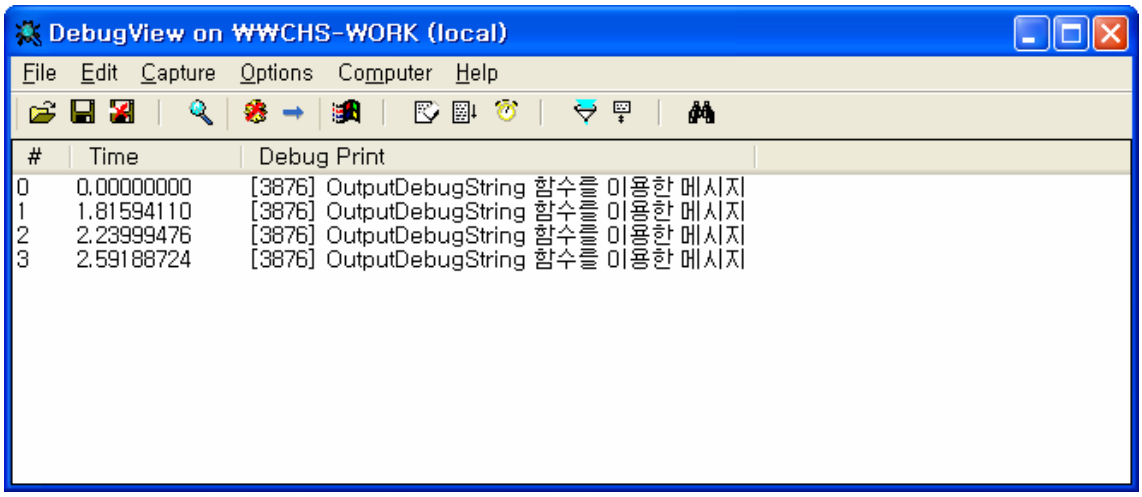
니라 **함수**입니다. 그러므로 릴리즈 모드로 빌드해도 동작합니다. 제 경우를 말씀드리면 사실 매크로를 활용한 디버깅은 거의 하지 않습니다. 개발 자체를 아예 릴리즈 모드로 하며 굳이 런타임에 메시지를 봐야 할 경우에는 OutputDebugString() 함수를 이용합니다. 이 방법을 이용하면 어떤 형식의 응용 프로그램이건 메시지 모니터링 및 디버깅이 가능합니다.

다음의 코드는 OutputDebugString() 함수를 이용한 예입니다.

```
190 ////////////////////////////////////////////////////////////////////////////
191 void CDebugMacroDlg::OnBnClickedButtonDebugdefine()
192 {
193     OutputDebugString(TEXT("OutputDebugString 함수를 이용한 메시지"));
194 }
195
```

이 코드를 릴리즈 모드로 빌드하고 디버그 모드가 아니라 일반 모드로 실행한 상태에서도 디버그 메시지를 볼 수 있습니다. 그리고 디버그 메시지를 보기 위해 굳이 덩치 큰 Visual C++ 2008 디버거를 실행할 필요도 없습니다. **DebugView**라는 작고 간편한 프로그램을 이용해서 실시간으로 메시지를 보면 됩니다. 이 DebugView는 과거 Sysinternals에서 개발한 공개 유틸리티입니다. http://www.sysinternals.com에 가서 무료로 다운 받을 수 있습니다. 가보시면 알겠지만 이 사이트가 통째로 MS사에 인수되었습니다. 덕분에 MSDN 기술문서 페이지로 링크됩니다. 이 사이트의 운영자는 Windows Internals의 저자로 알려진 Mark E. Russinovich 입니다.

다음 그림은 DebugView를 실행하여 OutputDebugString() 함수가 출력한 메시지를 확인한 모습입니다.



DebugView에 보이는 메시지는 OutputDebugString() 함수가 생성한 메시지뿐 만 아니라 디버그 매크로가 생성한 메시지들도 모두 볼 수 있습니다. 결국 이 툴을 이용하면 Visual

C++ 출력 윈도우에서 볼 수 있는 메시지를 모두 볼 수 있게 됩니다. 정말 여러모로 유용한 유틸리티가 아닐 수 없습니다.

결국 제가 추천하는 것은 디버그 매크로 보다는 OutputDebugString() + DebugView 정도로 요약할 수 있겠습니다. 만일 디버그 모드로 빌드했을 때만 메시지가 나오도록 하고 싶다면 다음과 같이 코드를 구성하면 됩니다.

```
190 ////////////////////////////////////////////////////////////////////////////
191 void CDebugMacroDlg::OnBnClickedButtonDebugdefine()
192 {
193 #ifdef _DEBUG
194     OutputDebugString(TEXT("OutputDebugString 함수를 이용한 메시지"));
195 #endif
196 }
197
```

_DEBUG가 정의된 경우에만 메시지가 나오도록 코드를 수정해주면 됩니다. 개인적으로는 코드가 눈에 거슬린다는 이유로 이러한 방법을 잘 사용하지 않고 있으나 이 방법 역시 많이 사용되는 좋은 디버깅 기법입니다.

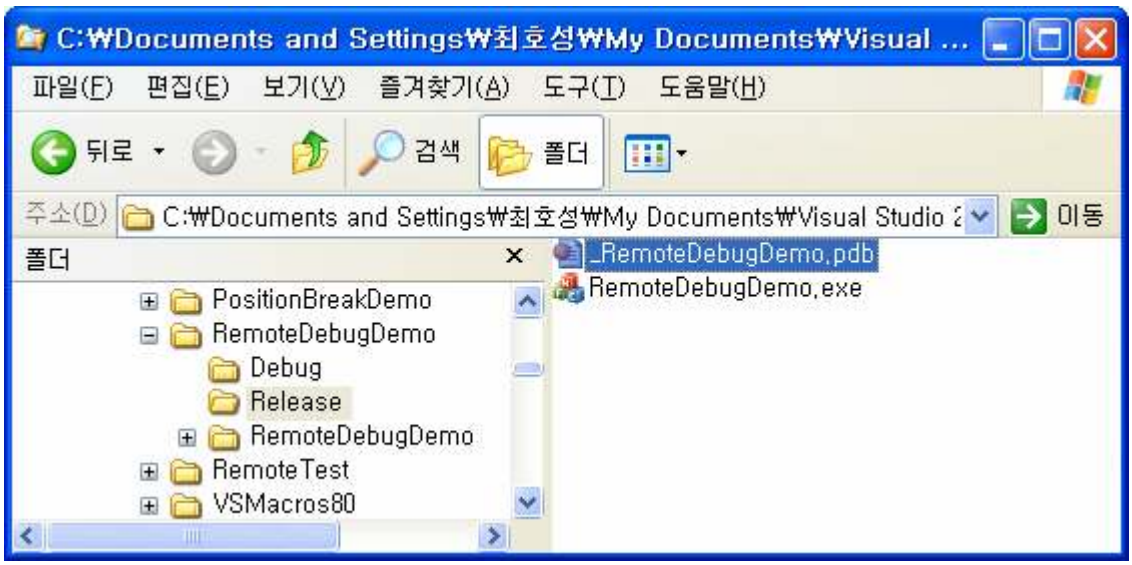
실행중인 프로세스 연결

지금까지 설명한 디버깅 기법들은 디버그 해야 할 대상 프로그램을 Visual C++에서 디버그 모드로 실행했을 때 가능한 것들이 대부분이었습니다. 그러한 이유 때문에 실제 상황에서의 문제를 해결하는데 큰 어려움이 따릅니다. 명확하게 원인을 규명 할 수는 없으나 디버그 모드로 실행할 때는 빌드모드에 상관없이 에러가 발생하지 않는데 탐색기를 통해 실행하면 에러가 발생하는 경우를 생각해 볼 수 있습니다. 생각만 해도 참 피곤한 일이 아닐 수 없습니다. 이러한 문제를 해결해야 하는 주체는 당연히 해당 개발자입니다. 만일 에러가 발생하는 것도 즉시 늘 발생하는 것이 아니라 하루에 혹은 1주일에 한번 어쩌다 발생하는 경우라면 정말 골치가 아픈 일입니다.

이때 가장 유용한 디버깅 방법이 실행중인 프로세스에 Visual C++ 2008 디버거를 연결하는 방법입니다. 이는 이어서 설명할 리모드 디버그 기능과도 연계되는 것입니다. 다음의 코드 예를 보면서 이 기법에 대한 설명을 이어 가겠습니다.

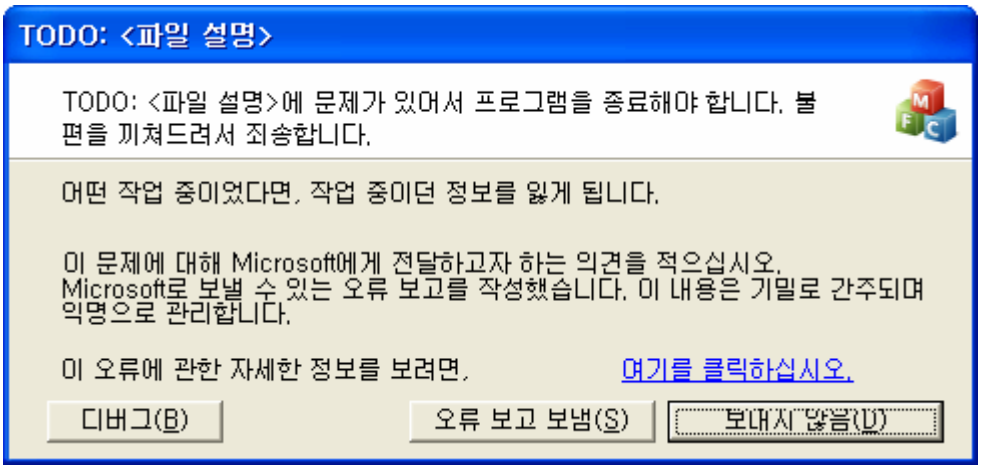
```
155 ////////////////////////////////////////////////////////////////////////////
156 void CRemoteDebugDemoDlg::OnBnClickedButtonError()
157 {
158     int* pnData = NULL;
159     *pnData = 10;
160 }
161
```

이 코드를 실행하면 당연히 에러가 발생할 수 밖에 없습니다. NULL 포인터에 접근했으므로 무조건 프로그램을 비정상 종료됩니다. 이 코드는 RemoteDebugDemo 예제의 일부이며 릴리즈 모드로 빌드하고 실행합니다.

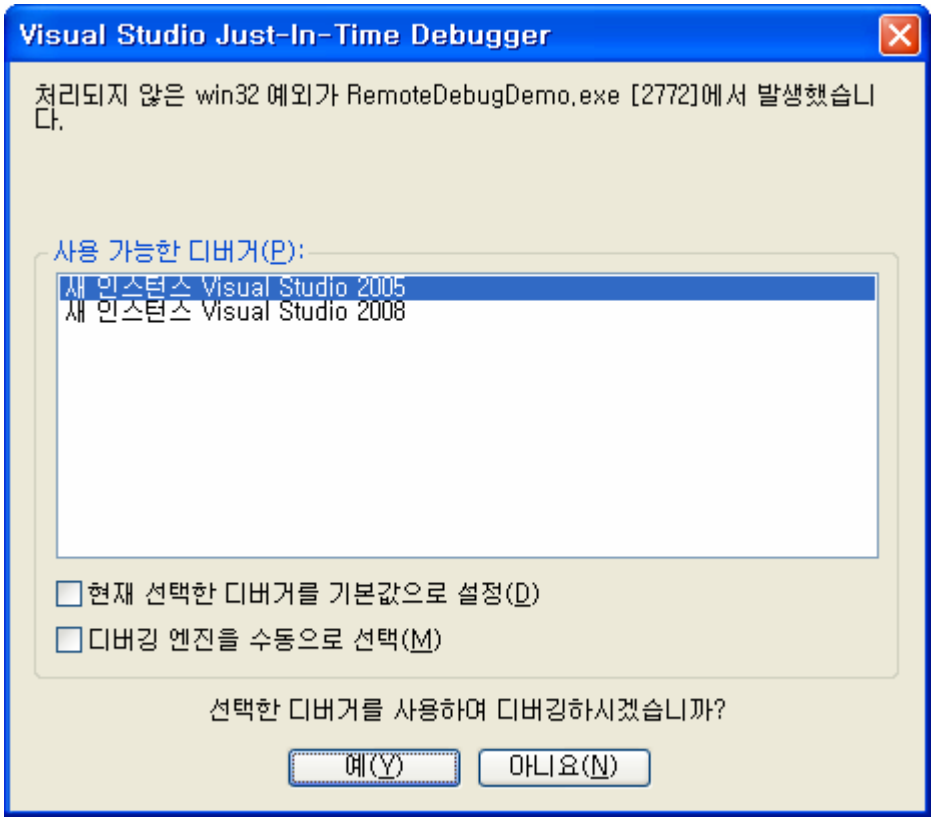


실행에 앞서 반드시 PDB 파일을 삭제하거나 이름을 바꿉니다. 그리고 소스코드를 편집하

던 Visual C++도 종료합니다. 실행해서 확인해 보면 당연히 다음과 같이 에러가 발생할 것입니다.

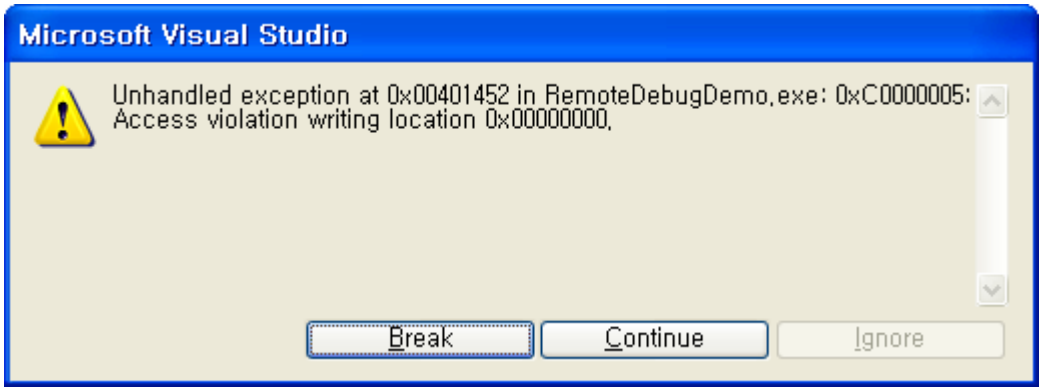


이 상태에서 “디버그” 버튼을 클릭하면 다음과 같이 디버깅 할 디버거를 선택하는 윈도우가 나타납니다.



이와 같은 윈도우가 출력된 것은 컴퓨터에 디버거가 설치되어 있기 때문입니다. 제 경우에는 Visual C++ 6.0/2005/2008 버전이 모두 설치되어 있으나 6.0 버전은 이 항목에 나타나지 않습니다. 상황 재연을 위해 이 상태에서 “새 인스턴스 Visual Studio 2008”을 선택하

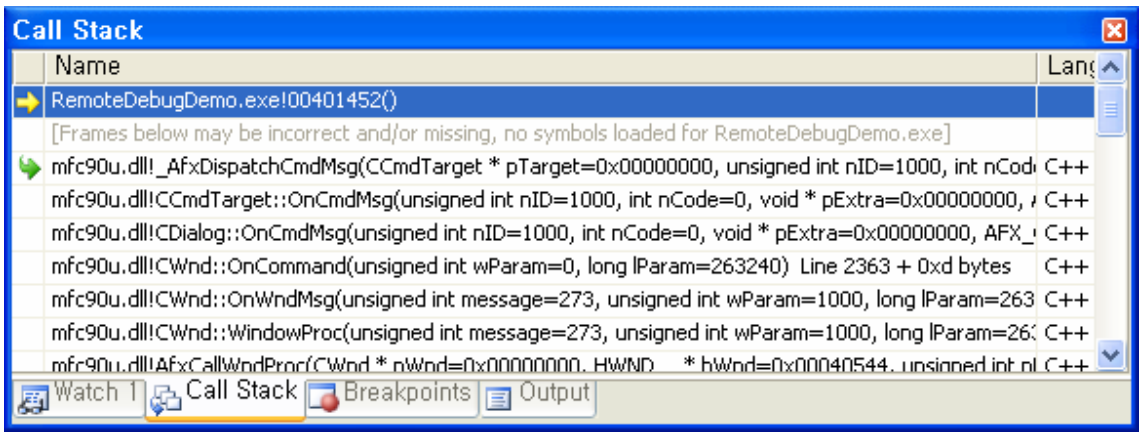
고 확인합니다. 그러면 Visual C++가 실행되고 다음과 같이 프로그램 실행이 중단된 원인을 보여줍니다.



Continue 버튼을 클릭하면 프로그램을 계속 다시 실행하도록 하는 것이며 Break 버튼을 클릭하면 브레이크 포인트가 동작하는 것처럼 프로그램의 실행을 멈추게 됩니다. 그러면 편집 윈도우가 MFC 소스 파일을 하나 열려서 다음과 같은 위치에서 에러가 발생했음을 가리킬 것입니다.

```
76
77      case AfxSigCmd_v:
78          // normal command or control notification
79          ASSERT(CN_COMMAND == 0);        // CN_COMMAND same as BN_CLICKED
80          ASSERT(pExtra == NULL);
81          (pTarget->*mmf.pfnCmd_v_v)();
82          break;
83
```

이 정보만 봐서는 구체적인 에러의 내용이나 코드 위치를 알 수 없습니다. 그러나 호출 스택 윈도우를 보시면 다음과 같이 에러가 발생한 위치(0x00401452)가 어디인지 구체적으로 알 수 있습니다.

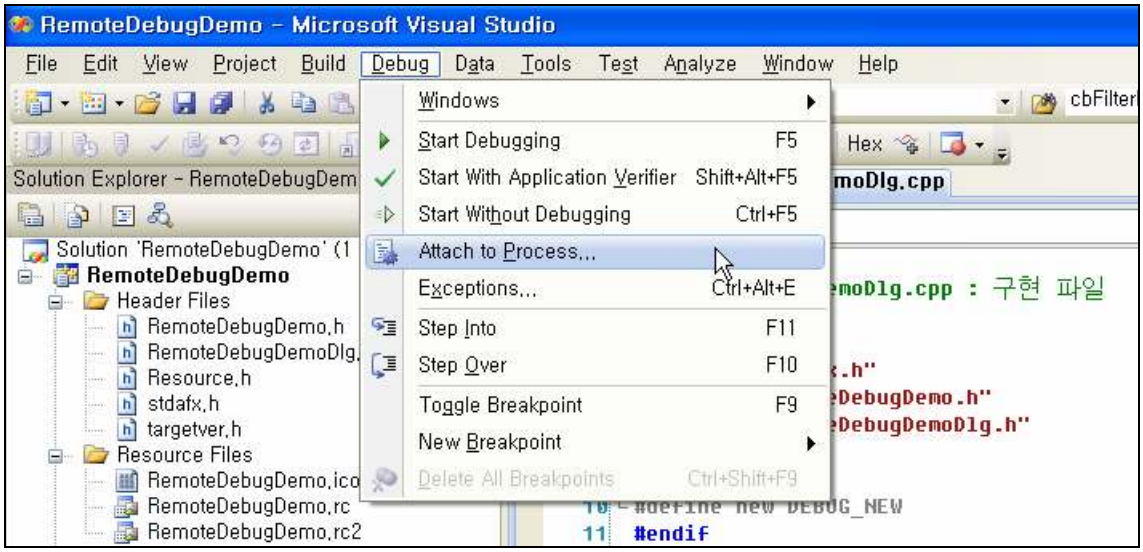


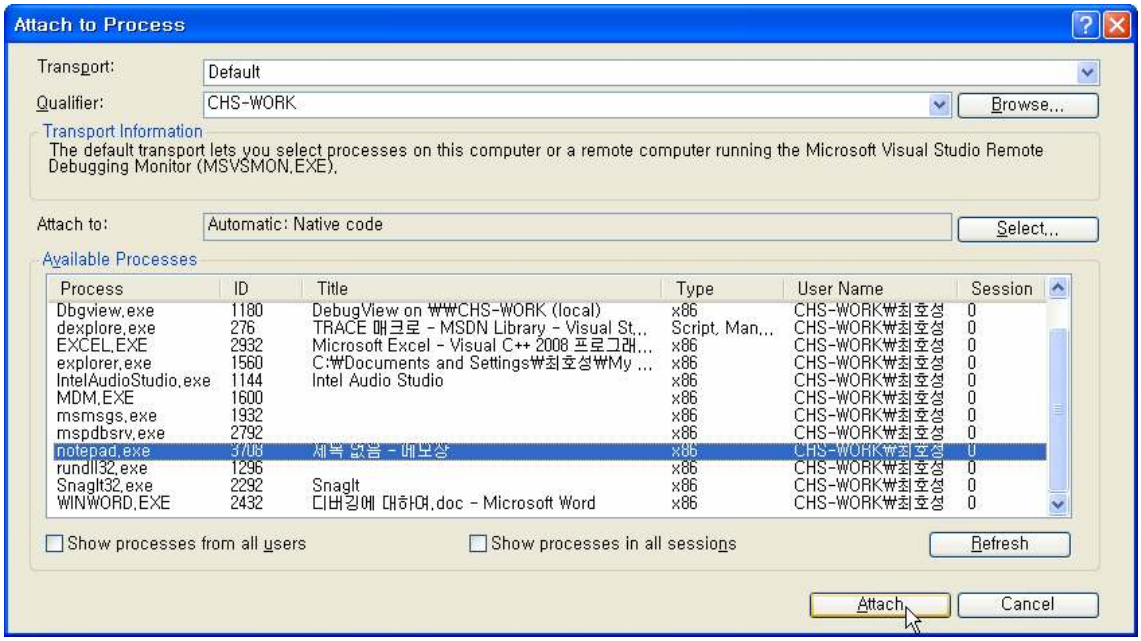
여기서 구체적인 함수 이름이 나오지 않고 주소로 표시된 것은 에러를 발생시킨 실행파일(RemoteDebugDemo.exe)에 대한 프로그램 디버그 데이터 베이스 파일이 없기 때문입니다.

재연을 위해 실행에 앞서 PDB 파일을 삭제하지 않았다면 분명히 구체적인 내용이 표시되었을 것입니다.

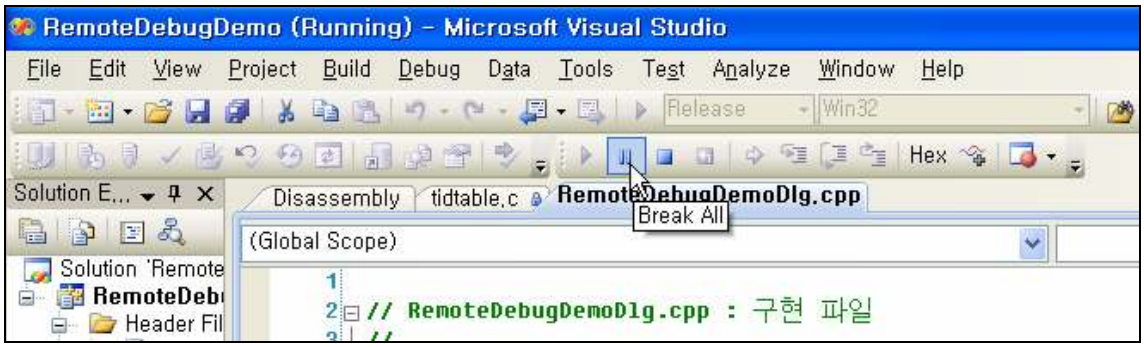
여기서 한 가지 중요한 사실을 알아야 합니다. **에러가 발생해서 실행이 중지된 프로세스는 말 그대로 실행이 일시 중지된 것이지 종료된 상태가 아니라는 것**입니다. 작업관리자를 통해 확인해 봐도 해당 프로세스는 아직 종료되지 않았음을 알 수 있습니다. 결국 실행중인 프로세스에 디버거가 연결되어 해당 프로세스를 디버깅하게 된 것입니다. 이것이 어떤 의미를 가지는지 좀더 자세히 살펴보겠습니다.

그러기 위해서 메모장을 실행 하고 Visual C++ 2008을 실행하고 아무 프로젝트나 열어 놓은 후, 다음과 같이 실행중인 메모장(Notepad.exe) 프로세스에 Visual C++ 디버거를 연결합니다.

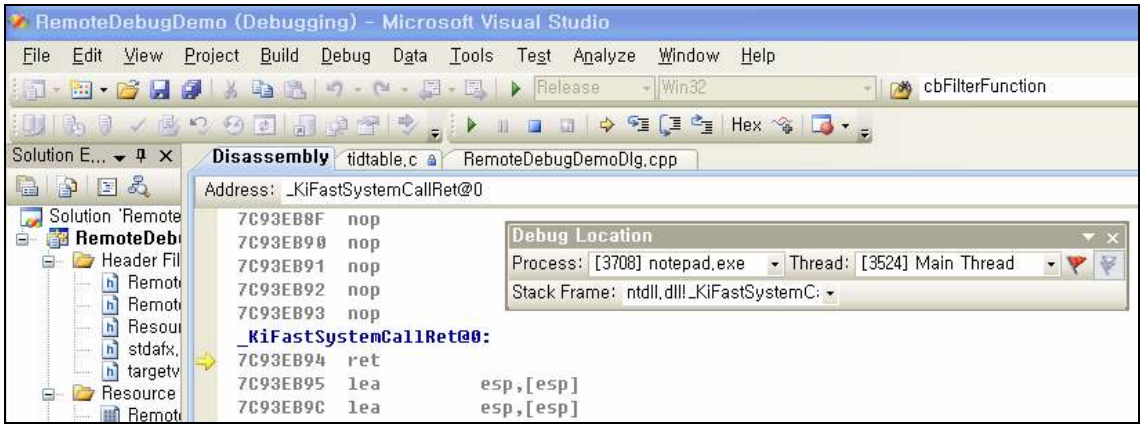




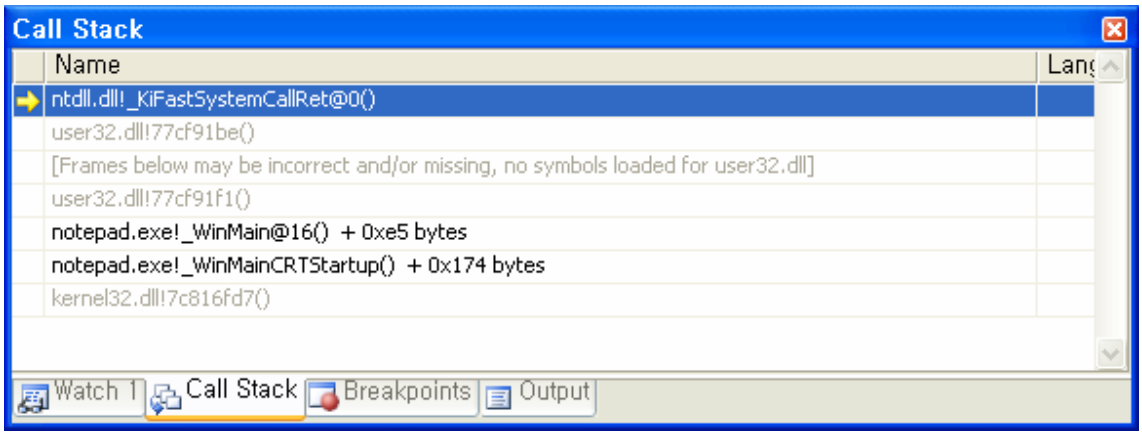
그리고 “Break All” 명령을 선택하여 메모장 프로세스를 일시 정시시킵니다.



그러면 메모장 프로세스를 디버그 할 수 있는 디버그 모드로 Visual C++가 전환됩니다.



만일 여러분이 메모장 프로그램의 PDB 파일을 가지 가지고 있다면 이 시점에서 호출 스택을 확인해서 좀더 구체적인 정보를 얻을 수 있을 것입니다.



이와 같이 실행중인 특정 프로세스에 디버거를 연결하면 실행을 일시 중지 시키고 해당 프로세스의 실행을 감시하는 것은 물론 사용중인 메모리의 내용까지 모두 확인할 수 있습니다. 만일 소스 코드까지 가지고 있다면 내가 개발하는 프로그램처럼 디버깅이 가능할 것입니다.

선배로써 한마디!

리버스 엔지니어링 이라는 것이 결국 실행중인 프로세스를 디버거를 통해 리어셈블한 코드를 분석하는 일입니다. 실제 소스 코드나 심볼(PDB)이 없더라도 어셈블리 코드를 이해할 수 있는 개발자라면 프로그램의 동작을 분석하고 원하는 대로 조작할 수가 있습니다. 스타크래프트의 맵핵과 같은 프로그램들은 이러한 방법으로 얻은 정보를 바탕으로 제작되는 크랙 소프트웨어 입니다. 공부를 목적으로 그러한 시도를 해보는 것은 상관없겠으나 악의적인 목적을 가진 소프트웨어를 제작 배포하는 것은 불법입니다. 그리고 게임가드나 핵쉴드와 같은 프로그램들은 이러한 디버거가 게임 프로세스에 연결되는 것을 막고 게임 프로세스의 메모리를 다른 프로세스가 읽지 못하도록 보호하는 역할을 합니다.

지금까지 설명된 내용을 바탕으로 개발자의 일상에 적용시켜보도록 하겠습니다. 개발자 철수는 자신이 개발중인 프로그램이 가끔씩 비정상 종료되는 버그가 있다는 사실을 통보 받고 1주일 이내에 이 문제를 해결하기로 하였다고 가정하겠습니다. 그러기 위해 철수는 문제가 발생한 시스템과 동일한 환경의 시스템을 구축하고 자신이 개발한 프로그램을 실행하고 에러가 발생할 때까지 방치합니다. 물론 이 시스템에는 그 실행파일을 빌드한 컴파일러와 소스코드 PDB 파일을 설치 해둡니다. 특히 PDB 파일은 실행파일이 있는 폴더에 복사해 둡

니다.

즉시 에러가 발생하지 않자 철수는 컴퓨터를 켜두고 프로그램을 실행시켜둔 상태에서 퇴근합니다. 그리고 다음날 운 좋게도 프로그램이 에러가 났다는 메시지를 보게 됩니다. 이미 대상 프로세스를 분석할 모든 준비가 끝났으므로 디버거를 실행하여 해당 프로세스에 연결하여 호출 스택 정보 및 에러를 유발한 코드를 확인합니다. 소스 코드까지 설치 해둔 덕분에 자신이 작성한 코드 중 어디에서 에러가 발생한 것인지 구체적으로 찾아내서 문제를 해결합니다.

지금 설명한 시나리오는 프로그램이 에러를 내주는 고마운 상황입니다. 이보다 더 머리 아픈 상황도 있습니다. 다음의 코드를 봅시다.

```
163 ////////////////////////////////////////////////////////////////////////////
164 int g_nData = 0;
165
166 ////////////////////////////////////////////////////////////////////////////
167 void CRemoteDebugDemoDlg::OnBnClickedButtonLoop()
168 {
169     g_nData = 0;
170     while(TRUE)
171     {
172         g_nData = rand() % 100;
173         Sleep(1);
174
175         if(g_nData == 100)      break;
176     }
177 }
178
```

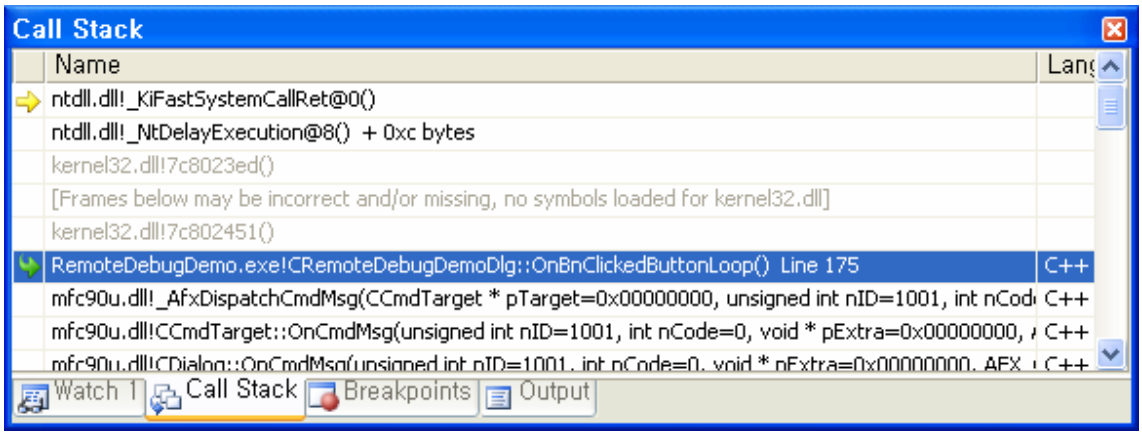
코드의 내용은 결국 임의의 값이 100이면 루프를 탈출하는 것입니다. 그런데 100으로 나눈 나머지가 100이 될 수는 없습니다. 따라서 OnBnClickedButtonLoop() 함수는 무한 루프를 돌게 됩니다. 결국 프로세스는 종료되지도 않고 아무런 에러도 내주지 않는 상태에서 마치 멈춘 것처럼 보이게 됩니다. 작업관리자를 이용해 프로세스의 상태를 확인해 보면 “응답 없음” 상태가 되어 있음을 볼 수 있습니다. 이 경우 프로세스 연결 기능을 사용하면 손쉽게 문제를 해결할 수 있습니다.

앞서 설명한 것처럼 해당 프로세스의 소스코드, PDB 파일을 준비하고 Visual C++ 2008을 이용하여 코드를 열고 “응답 없음” 상태에 빠진 프로세스에 연결합니다. 그리고 “Break All” 명령을 선택합니다. 그러면 Visual C++ 2008의 디버거는 프로세스의 실행을 일시 중지하고 다음과 같이 디버그 모드 화면으로 전환될 것입니다.

```
163 ////////////////////////////////////////////////////////////////////////////
164 int g_nData = 0;
165
166 ////////////////////////////////////////////////////////////////////////////
167 void CRemoteDebugDemoDlg::OnBnClickedButtonLoop()
168 {
169     g_nData = 0;
170     while(TRUE)
171     {
172         g_nData = rand() % 100;
173         Sleep(1);
174
175         if(g_nData == 100)      break;
176     }
177 }
178
```

Debug Location
Process: [3216] RemoteDebug[Thread: [364] Main Thread
Stack Frame: RemoteDebugDemo.exe!

이 때 호출 스택의 내용을 보면 다음과 같습니다.



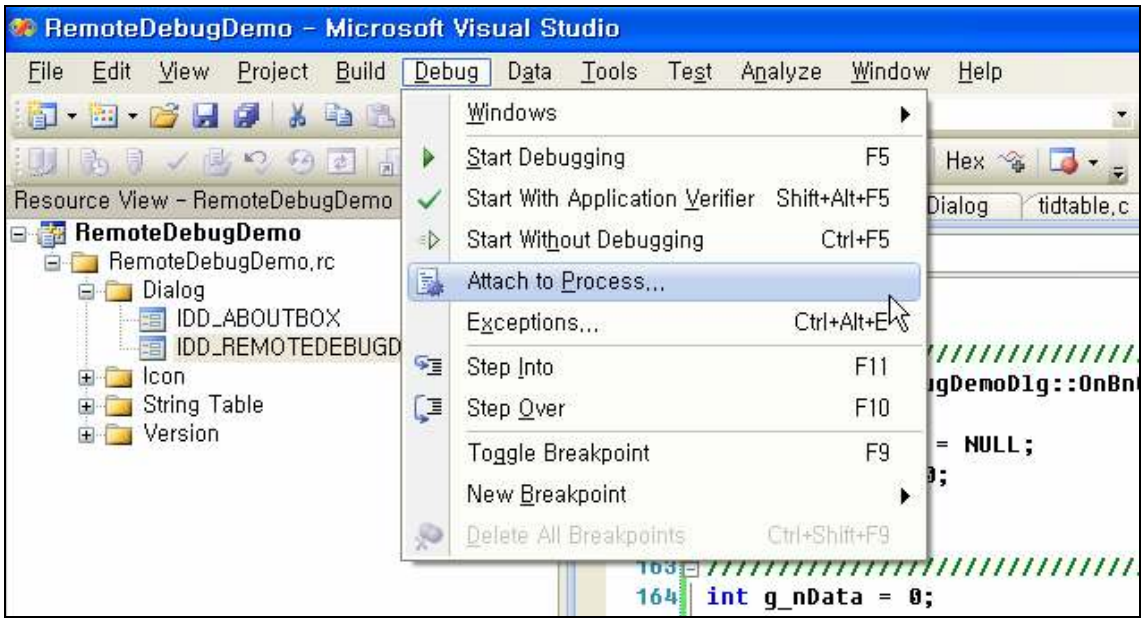
무한 루프를 돌고 있는 코드가 어디인지 직접 확인할 수 있게 됩니다. 이는 정말 편리한 기능이 아닐 수 없습니다. 이쯤 되면 디버깅에 어느 정도 자신감을 가져도 좋을 것입니다. 이와 같이 실행중인 프로세스에 Visual C++ 디버거를 연결하여 사용할 수 있는 기능은 로컬 컴퓨터뿐만 아니라 원격지의 컴퓨터에도 가능합니다. 이것을 가능하게 하는 것이 바로 **리모트 디버그** 기능입니다.

리모트 디버그

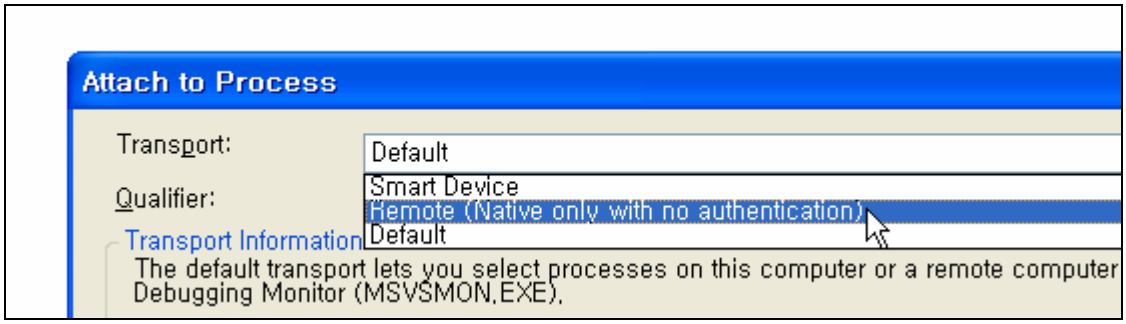
개발자 PC에서는 멀쩡히 잘도 돌아가는 프로그램이 어떤 사용자PC에서는 1시간에 한번꼴로 비정상 종료된다고 가정해 봅시다. 이 경우 개발자는 참 답답할 수 밖에 없습니다. 원인을 규명하기 위해 결국은 해당 사용자의 PC를 가져오던가 본인이 직접 가서 확인해야 합니다. 그나마 지리적으로 가까우면 좋겠으나 개발자는 서울에 있는 반면 사용자는 부산에 있는 경우 직접 가는 일도 쉬운 일은 아닙니다. 이러한 문제를 해결하는데 가장 좋은 방법은 리모트 디버그 기능을 사용하는 것입니다.

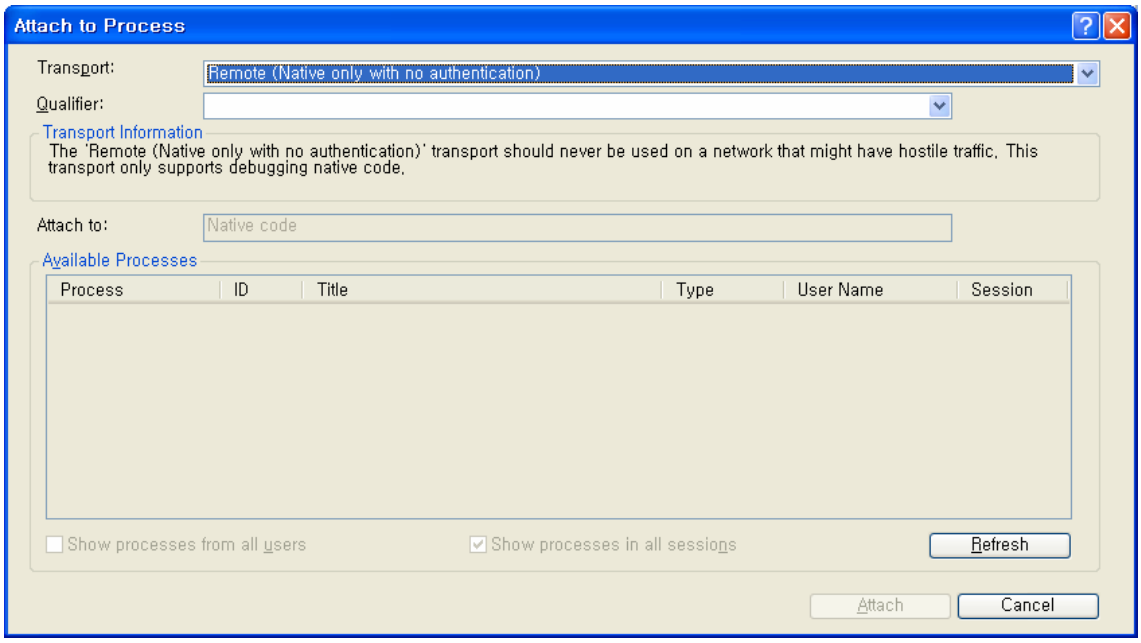
대한민국은 세계 최고를 다투는 네트워크 인프라를 가지고 있습니다. 초고속 인터넷 사용자 수나 인구대비 비율에서 세계 어느 곳에도 뒤지지 않습니다. 그러니 이 좋은 환경을 잘 활용하여야 할 것입니다.

우선 로컬 PC에서 리모트 디버그를 테스트 해보도록 하겠습니다. 이를 위해 앞서 다루었던 RemoteDebugDemo 예제 프로젝트를 열어 두고 윈도우 탐색기를 이용해 빌드된 RemoteDebugDeme.exe를 실행해 둡니다. 그리고 Visual C++의 디버그 메뉴에서 “Attch to Process” 메뉴를 선택합니다.



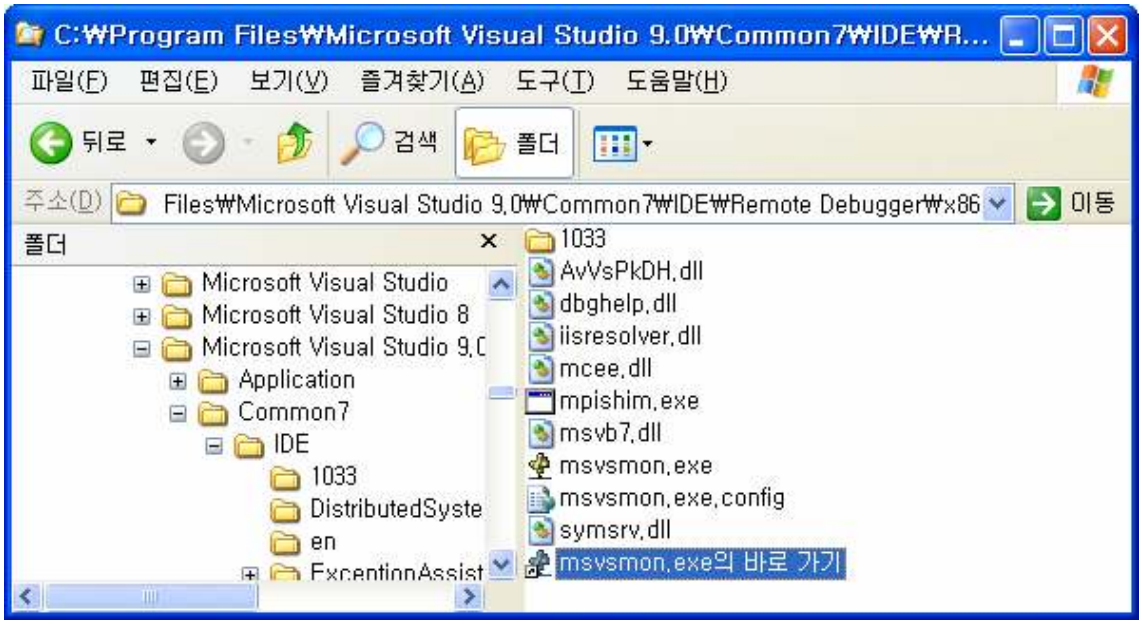
여기까지는 기존의 프로세스 연결과 동일합니다. 그러면 프로세스 연결을 위한 윈도우가 나타날 텐데 여기서 “Transport” 부분을 “Remote”로 변경합니다.





그러면 그림처럼 프로세스 연결화면이 변경될 것입니다. 이제는 Qualifier에 리모트 PC의 IP 주소를 입력하여 리모트 PC와 연결하여야 합니다. 그런데 아직 리모트 디버거를 실행하지 않았습니다. 그러므로 다음과 같은 방법으로 리모트 디버거를 실행합니다.

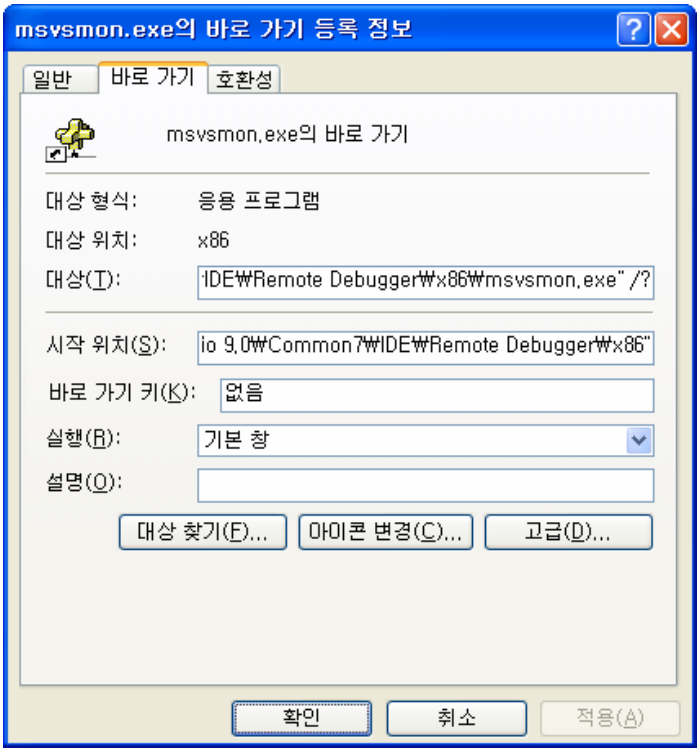
우선 윈도우 탐색기를 실행하여 Visual C++ 2008이 설치된 폴더를 열고 아래와 같은 경로에서 리모트 디버거 실행파일의 단축키를 만듭니다.



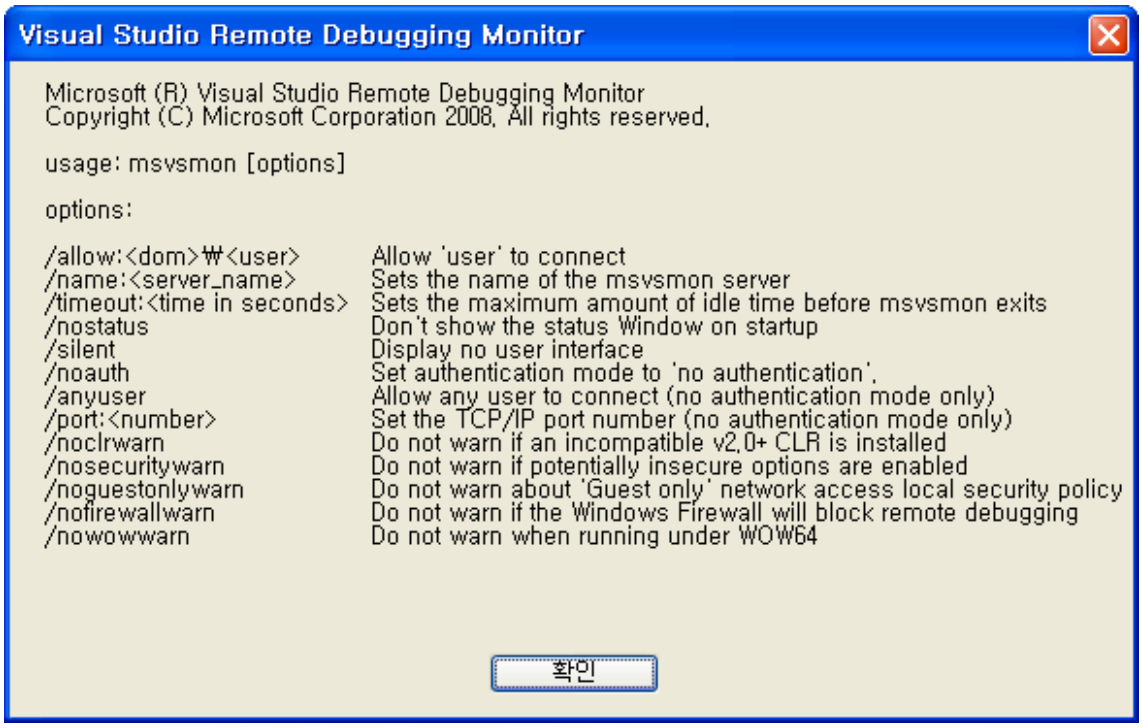
Visual Studio 설치 시 특별한 변화를 주지 않았다면 아래 경로에 해당될 것입니다.

C:₩Program Files₩Microsoft Visual Studio 9.0₩Common7₩IDE₩Remote Debugger₩x86

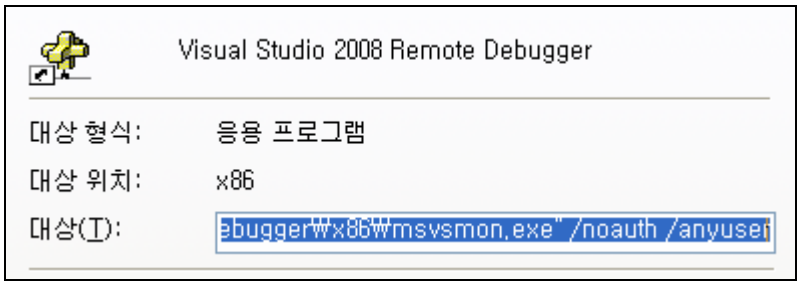
이 폴더에 보면 msvsmon.exe라는 파일이 있습니다. 그림처럼 이 파일에 대한 단축아이콘을 생성하고 속성을 열어 다음과 같이 “대상” 부분의 값을 수정합니다.



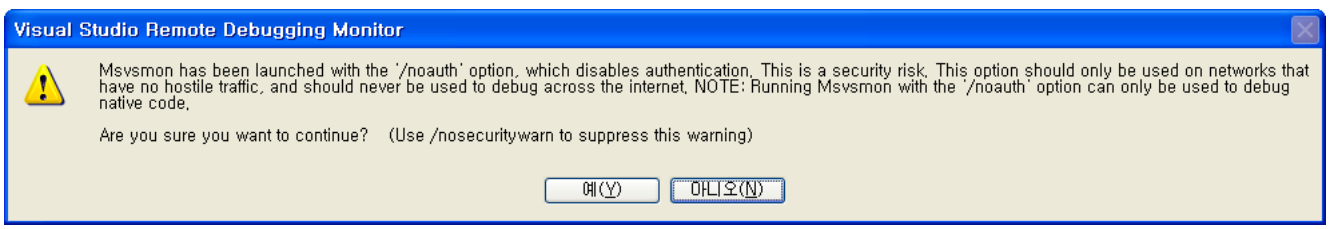
수정한 내용은 msvsmon.exe에 **/? 옵션**을 주어 실행하도록 변경한 것입니다. 그리고 실행을 해보면 다음과 같은 내용을 볼 수 있습니다.



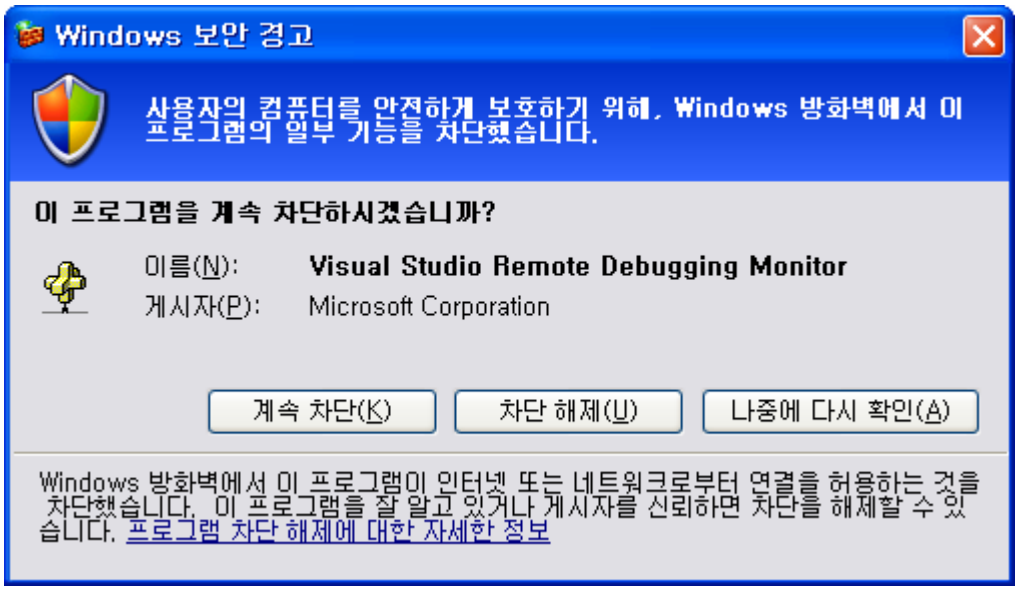
Visual Studio Remote Debugging Monitor의 옵션과 내용에 대해 열거하고 있습니다. 여기서 주의 깊게 볼 것은 **/noauth** 옵션과 **/anyuser** 옵션입니다. 이 두 옵션은 **아무런 인증 없이 아무나 리모트 디버거에 연결할 수 있도록 허용하는 것**입니다. 보안적으로 바람직하지는 않으나 편의를 위해 당장은 이 옵션을 사용하기로 하겠습니다. 다시 단축아이콘의 속성을 열어서 다음과 같이 실행 옵션을 변경합니다.



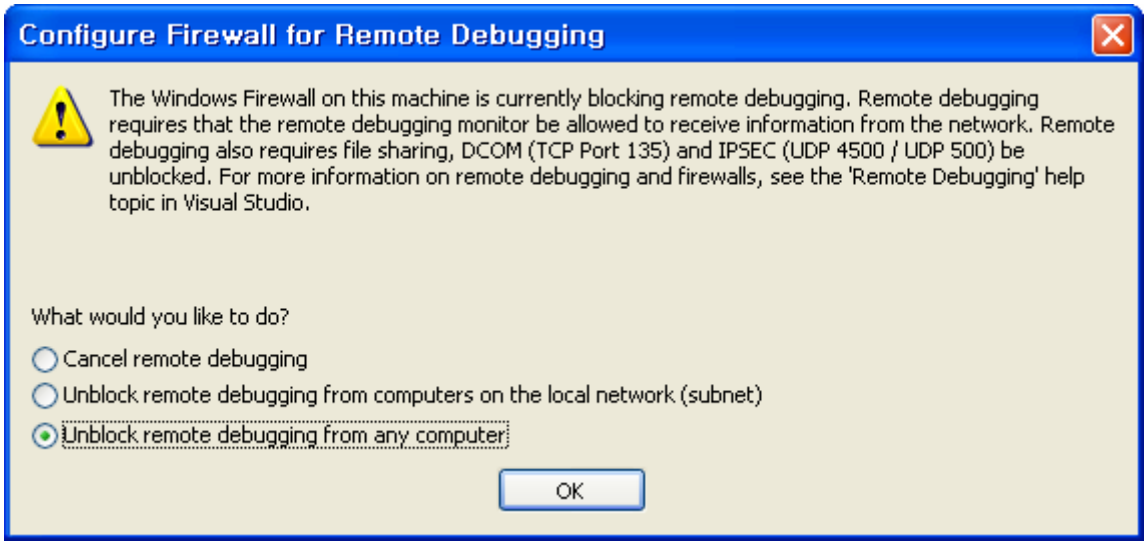
그리고 실행해보면 다음과 같은 확인 메시지를 볼 수 있습니다.



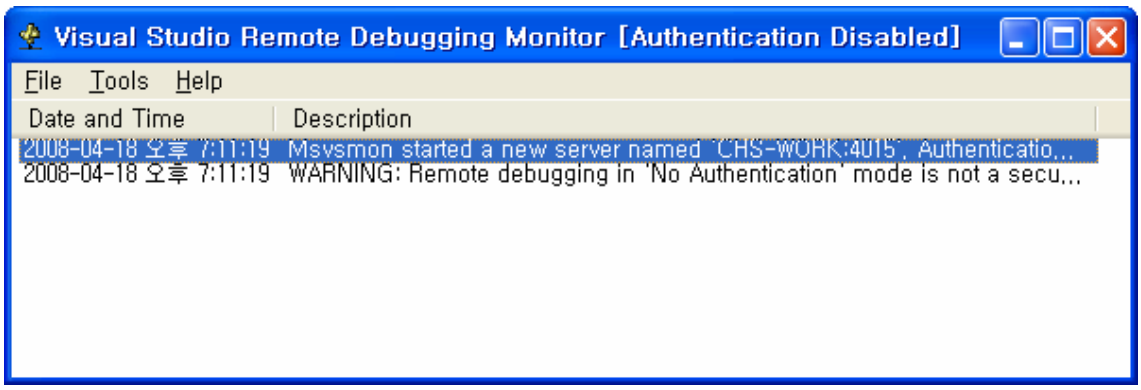
내용을 확인해보면 **보안적인 위험요소를 감수하고 실행할 것인지를 묻는 것**입니다. 우선은 "예"를 선택하여 실행합니다. 그러면 다음과 같이 윈도우 방화벽의 보안 경고를 볼 수 있습니다. 리모트 디버거 프로세스가 외부의 연결을 받을 수 있어야 하기 때문에 "차단 해제"를 선택하여 예외처리 해줍니다.



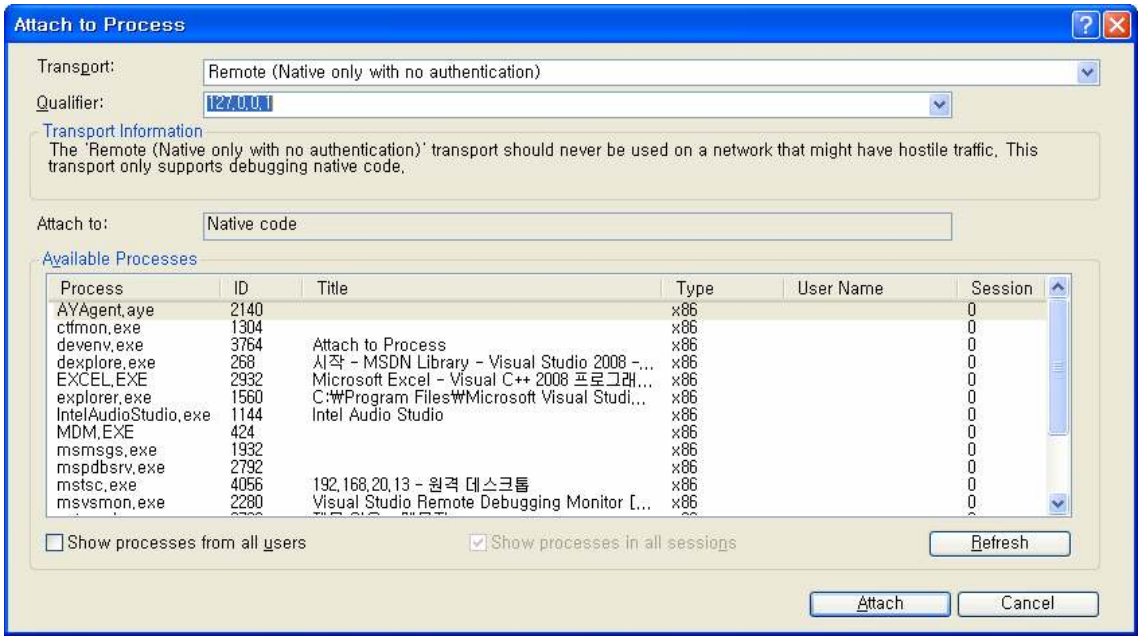
그러고 나면 리모트 디버거가 동작하기 위해 외부 연결을 아무런 조건 없이 받을 것인지 설정하는 창이 나옵니다. 그림처럼 모든 원격 컴퓨터들의 리모트 디버그 요청을 받도록 합니다. 보안을 위해 로컬 네트워크만 연결을 받도록 하여도 무방하겠습니다. 물론 외부 인터넷에 존재하는 PC를 리모트 디버그 할 때는 모든 컴퓨터들이 연결될 수 있도록 설정하여야 합니다.



확인하여 설정을 완료하면 다음과 같이 리모트 디버거 모니터링 윈도우가 나타납니다.



리모트 디버거가 실행되었고 아무런 인증 장치가 없다는 경고도 보입니다. 이제 내 PC에 리모트 디버거의 실행이 완료되었으니 Visual C++ 디버거를 연결해 보기로 하겠습니다. 다음 그림과 같이 Qualifier에 원격지 IP주소(현재 예에서는 나의 로컬 PC에 연결하므로 나 자신을 지칭하는 127.0.0.1 주소를 입력합니다. 그리고 엔터키를 치면 원격 컴퓨터의 실행중인 프로세스 리스트가 나타납니다.

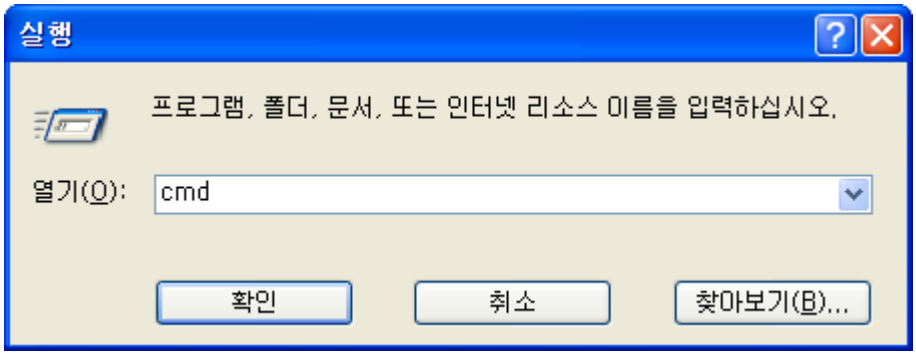


이제 이후의 처리는 로컬에서 실행중인 프로세스에 연결하는 방법과 같습니다.

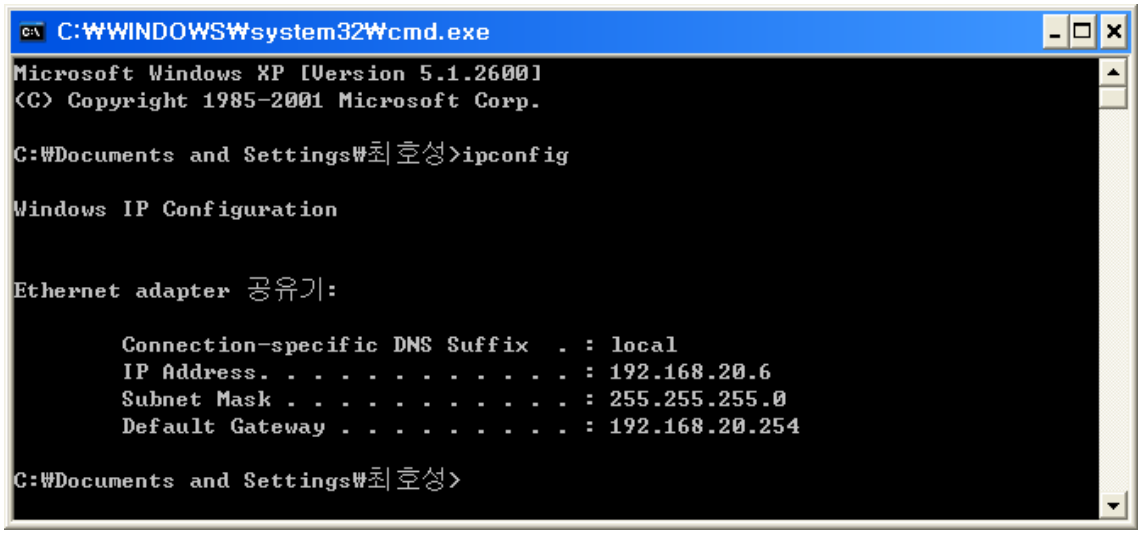
다시 우리 철수의 이야기로 돌아가겠습니다. 부산에 사는 사용자의 협조를 얻어 원격으로 에러가 발생한 PC에 연결하는 시나리오를 순서대로 열거하겠습니다.

Step 1 원격 디버그의 대상이 되는 PC의 IP주소를 알아냅니다.

상대 PC의 IP 주소를 알아내려면 **ipconfig** 명령을 이용하면 됩니다. 그러기 위해서 다음과 같이 **cmd.exe**를 실행합니다.



그리고 명령 프롬프트에 **ipconfig** 명령을 입력하여 실행합니다.



원격 사용자에게 이 방법으로 IP주소를 알아내도록 합니다. 만일 **원격 사용자가 공유기를 사용하고 있다면 직접 연결이 불가능하므로 주의**합니다. 그런 경우 공유기에 연결된 케이블을 PC에 직접 연결하여 공인 IP주소를 받도록 하여야 합니다.

Step 2 해당 PC에 리모트 디버거 프로그램을 전송합니다.

아마도 원격의 사용자는 리모트 디버거 프로그램을 가지고 있지 않을 것입니다. 그러므로 리모트 디버거 프로그램을 압축하여 해당 사용자게 메일로 보내줍니다. 물론, 다른 송수신 프로그램이나 FTP를 이용하여도 무방합니다. 참고로 리모트 디버거 프로그램이 있는 폴더(C:₩Program Files₩Microsoft Visual Studio 9.0₩Common7₩IDE₩Remote Debugger₩x86)를 압축하면 대략 **4.2MB** 정도 됩니다.

Step 3 리모트 디버거를 설정하고 실행합니다.

원격 사용자에게 메일을 확인하고 /noauth 옵션과 /anyuser 옵션으로 리모트 디버거를

실행하도록 합니다. 여기까지 진행되면 사실 대부분의 준비가 끝납니다.

Step 4 해당 소스를 열고 원격으로 연결합니다.

Visual C++ 2008을 실행하고 해당 프로젝트를 열고 대기합니다.

Step 5 에러 상황이 발생하면 Break All 명령을 선택하고 디버깅을 시작합니다.

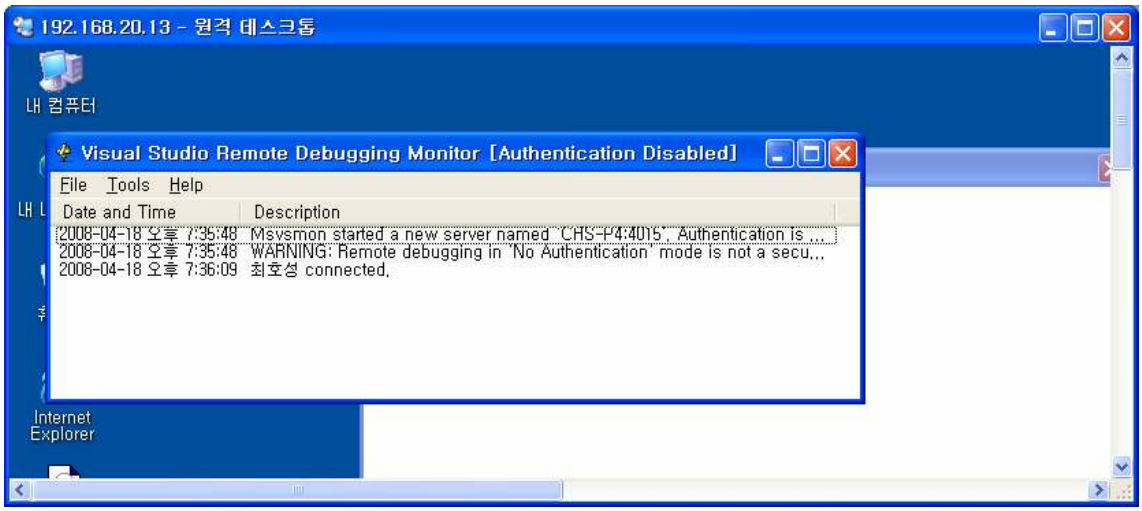
에러가 발생하지 않더라도 필요하다면 리모트 디버거에 연결하여 대상 프로세스를 살펴봅니다.

Step 6 원인을 찾고 리모트 디버거를 종료합니다.

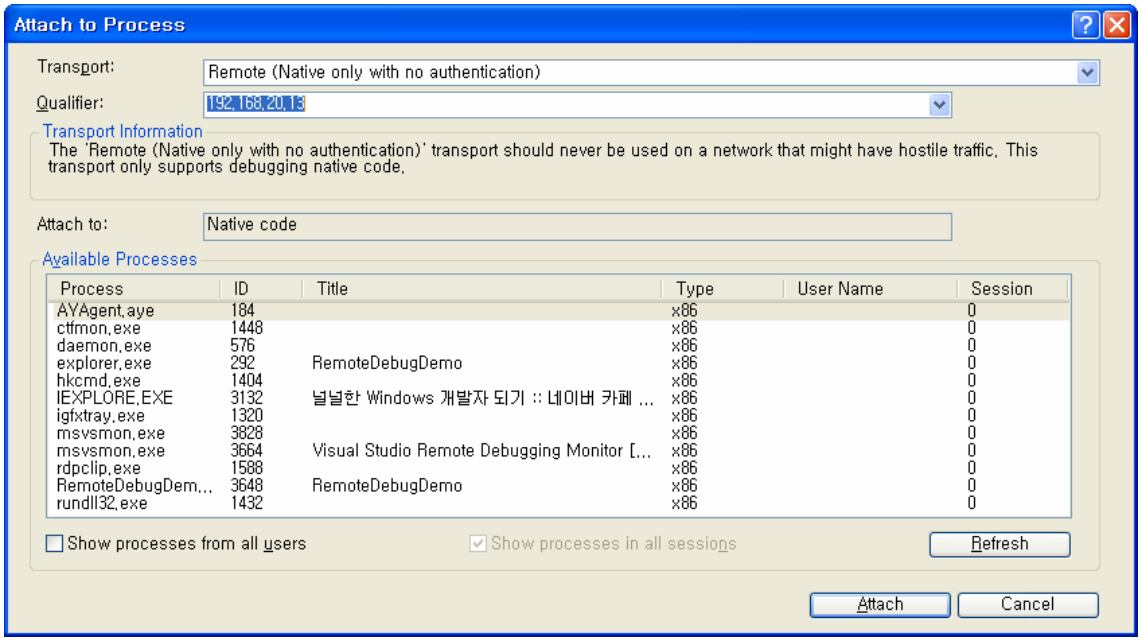
원격 PC의 리모트 디버거 프로그램을 종료하고 Visual C++ 2008을 닫습니다.

다음 그림들은 이와 같은 방법을 이용하여 원격 디버그를 수행한 예 입니다. 원격지의 IP 주소는 **192.168.20.13** 입니다. 이 주소는 사설 주소로써 공유기를 사용하고 있는 주소입니다. 여기에 연결한 PC는 같은 공유기에 연결된 PC로 IP주소가 **192.168.20.6** 입니다. 둘 다 사설 주소입니다만 같은 네트워크 세그먼트 내에 존재하므로 직접연결이 가능한 상황입니다.

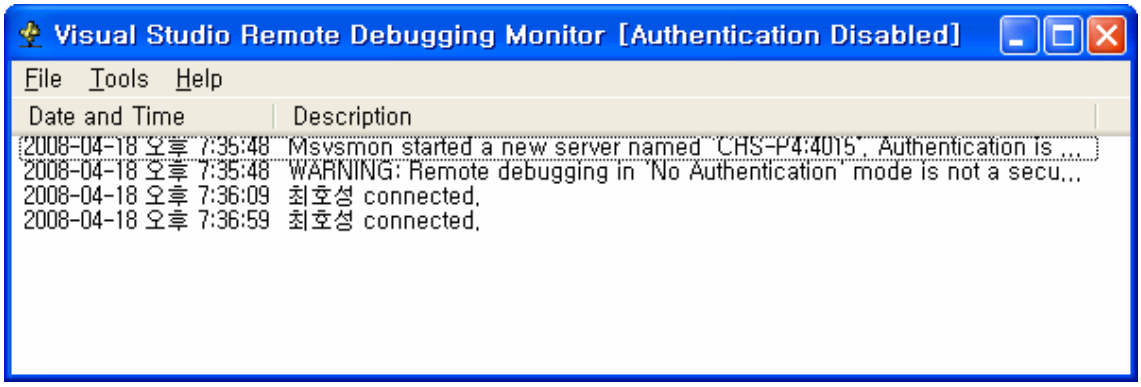
다음 그림은 원격 PC에 원격 데스크톱을 연결하여 리모트 디버거를 실행해둔 상태입니다. 오른쪽에 하얗게 보이는 것은 무한 루프에 빠진 RemoteDebugDemo.exe 프로세스 화면입니다.



다음 그림은 이 PC에 연결하기 위한 Visual C++의 화면입니다. Qualifier에 원격지 IP주소(192.168.20.13)가 들어있습니다.



다음 그림은 원격 PC의 리모트 디버거에 출력된 로그입니다. 원격으로 디버그를 수행하는 컴퓨터에 로그온 한 사용자 계정 이름이 출력되었습니다.



지금까지 설명한 디버그 방법들을 잘 활용하시면 거의 대부분의 경우에 대한 디버깅이 가능하리라 확신합니다. 특히 리모트 디버그 방법을 잘 활용하시면 큰 도움이 될 것입니다. 개인적으로는 이 기법을 이용하여 어떤 게임의 치명적 오류를 수정한 경험이 있습니다. 제대로 프로그램을 디버깅 할 수 있기 위해서는 시스템 프로그래밍 실력은 물론 필요하다면 일부 어셈블리 코드를 이해할 수도 있어야 하고 원격 디버깅을 위해 TCP/IP 네트워크에 대한 어느 정도의 지식도 가지고 있어야 합니다. 결국 많이 알면 그 만큼 버그를 만들 일이 줄어들고 혹시 발생한 버그가 있더라도 빠르게 해결할 수 있습니다.