

예전에 책으로 출판할 계획으로 작성한 글인데 책 출판 계획이 사라졌으므로 그냥 공개합니다.  
(책 집필은 너무 큰 인내를 필요로 해서 참 하기 어렵네요^^)

최흥배 ([jacking75@gmail.com](mailto:jacking75@gmail.com))

## 6장 디버깅

프로그램에 버그가 발생했을 때 이것을 잡는 행위를 우리는 디버깅이라고 한다.

프로그래밍에 있어서 디버깅은 필수이다. 아주 간단한 프로그램을 제외하고는 단 한번의 코딩으로 버그 없는 프로그램을 만드는 것은 불가능에 가깝다. 프로그래머는 완벽할 수 없고, 우리가 만드는 프로그램은 복잡하기 때문에 언제나 버그가 발생해서 이것을 잡기 위해서 프로그래밍에서 디버깅을 빼 놓을 수 없다.

그러나 디버깅은 프로그램을 직접적으로 만드는 것은 아니고, 디버깅을 해야 할 상황이 모든 사람에게 동일할 수 없기 때문에 디버깅을 다루는 책은 시중에 별로 없고, 디버깅을 주제로 가르치는 커리큘럼도 없어서 많은 초보 프로그래머들이 디버깅 기술이 부족하다.

보통 디버깅 기술은 프로그래밍을 자주 하면서 이때 발생한 버그를 잡아가면서 디버깅 기술을 쌓아가기 때문에 신입 프로그래머와 노련한 경력 프로그래머 간의 차이점의 하나로 디버깅 기술과 경험이라고 할 수 있다.

VC++가 다른 C++ 툴과 비교해서 뛰어난 점 중의 하나가 디버깅 기능이 뛰어나다는 점이다. 본인은 몇 년 전에 유닉스 플랫폼에서 프로그래밍을 몇 년간 한적이 있는데 이때 가장 힘들었던 점이 디버깅 하기가 너무 불편했던 것이었다. 프로그램을 만들 때 30% 이상의 시간을 디버깅에 소모되는데 디버깅 기능이 빈약한 경우 보통 보다 훨씬 더 많은 시간을 소모하고 스트레스도 많이 받는다.

빨리 프로그램을 만들고, 버그로 고통을 받지 않기 위해서는 디버깅 기술과 경험을 쌓는 것이 중요하다. 이 장에서는 VC++에 있는 가장 기초적인 디버깅 기능을 가르쳐주는 것을 시작으로 디버깅을 능숙하게 하기 위한 고급 디버깅 기술을 설명한다. VC++의 디버깅 기능은 사용하기 쉽고, 어려운 개념을 이해할 필요도 없고, 필자가 하는 설명을 꼭 따라가면 되니 가벼운 마음으로 따라오기 바란다.

### 1. 디버깅 기초

## 1.1 기초 중의 기초 - 중단점 설정과 디버깅하기

### 중단점 설정

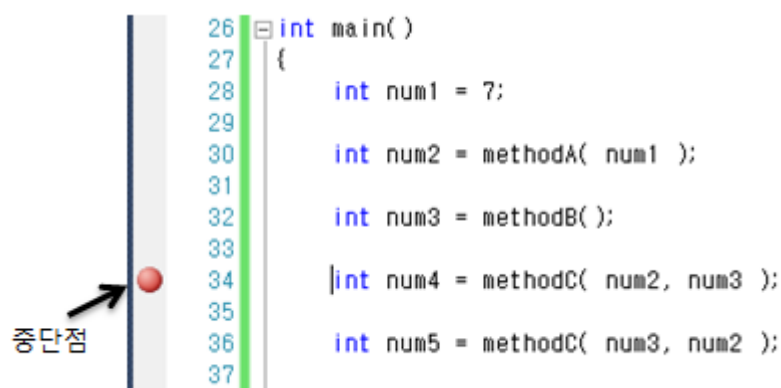
VC++에서 디버깅을 하기 위해서는 가장 먼저 해야 하는 작업이 '중단점(break point)'를 디버깅 하기 원하는 곳에 설정하는 것이다. 중단점은 프로그램 실행 도중 소스 중에서 멈추기 원하는 곳에 설명하면 디버깅을 시작하면 프로그램이 실행을 하다가 브레이크 포인트를 설정한 부분에서 멈춘다. 브레이크 포인트에 의해서 실행 중 멈추게 되면 그때 현재 프로그램의 상태를 파악할 수 있다.

브레이크 포인트 설정은 소스 중에서 프로그램이 실행 도중 멈추기 원하는 부분에서 키보드의 'F9' 단축키를 누르면 된다.

```
26 int main( )
27 {
28     int num1 = 7;
29
30     int num2 = methodA( num1 );
31
32     int num3 = methodB( );
33
34     int num4 = methodC( num2, num3 );
35
36     int num5 = methodC( num3, num2 );
37 }
```

< 그림 1. 브레이크 포인트를 설정하지 않은 코드 >

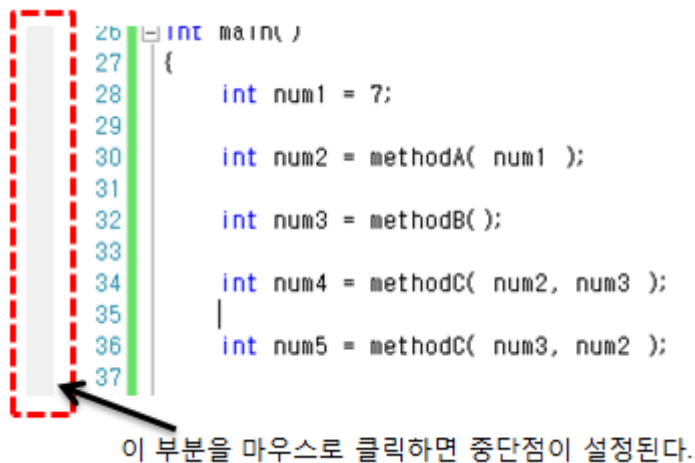
<그림 1>에서 34라인에 있는 코드를 실행 하기 전에 변수 num2와 num3이 어떤 값으로 되어 있는지 알기 위해서 34라인에 커서를 놓아 둔 후 단축키인 'F9' 키를 누르면 아래 그림과 같이 브레이크 포인트가 설정된다. 브레이크 포인트가 설정되면 그 위치에 빨간색 원이 표시된다.



```
26 int main( )
27 {
28     int num1 = 7;
29
30     int num2 = methodA( num1 );
31
32     int num3 = methodB( );
33
34     int num4 = methodC( num2, num3 );
35
36     int num5 = methodC( num3, num2 );
37 }
```

< 그림 2. 중단점 설정 >

중단점은 설정은 'F9' 키 이외에도 마우스로 클릭으로도 가능하다

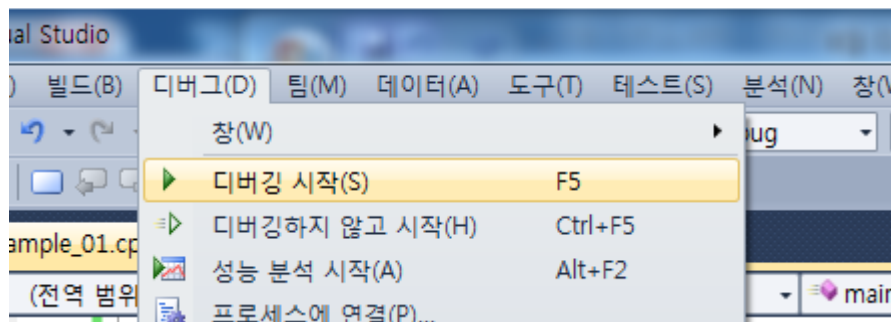


< 그림 3. 마우스 클릭으로 중단점 설정 >

중단점을 없을 때는 없애고 싶은 중단점이 있는 위치에 커서를 옮긴 후 단축키 'F9' 키를 누르면  
가 또는 중단점을 마우스로 클릭하면 없어진다.

## 디버깅 하기

중단점을 설정한 후 디버깅을 할 때는 단축키 'F5' 키를 누르던가 메뉴의 [디버깅] -> [디버깅 시작]'을 선택한다.



< 그림 4. 디버깅 시작 >

디버깅 시작을 하면 아래 그림과 같이 중단점을 설정한 곳에 프로그램이 멈춘다.

```

26 int main( )
27 {
28     int num1 = 7;
29
30     int num2 = methodA( num1 );
31
32     int num3 = methodB( );
33
34     int num4 = methodC( num2, num3 );
35
36     int num5 = methodC( num3, num2 );
37
38 }

```

< 그림 5. 디버깅 시 중단점에서 멈춤 >

중단점에서 멈추면 디버깅 관련 창 중 '자동'창을 통해서 중단점을 설정한 곳에서 사용되는 변수들의 값을 볼 수 있다.

```

29
30     int num2 = methodA( num1 );
31
32     int num3 = methodB( );
33
34     int num4 = methodC( num2, num3 );
35
36     int num5 = methodC( num3, num2 );
37
38
39     getchar( );
40     return 0;
41 }

```

100 %

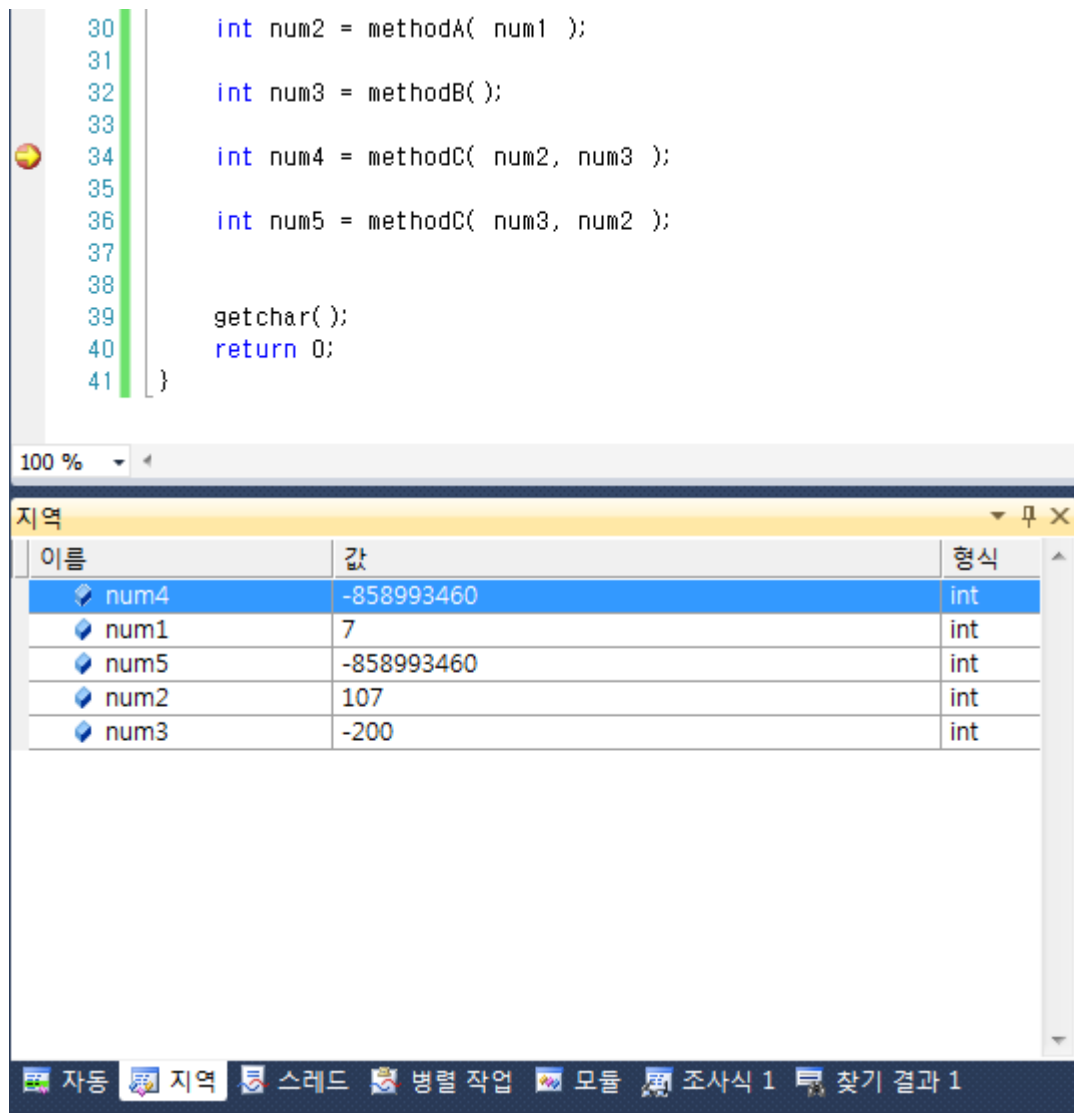
자동

이름	값	형식
num2	107	int
num3	-200	int
num4	-858993460	int

< 그림 6. 디버깅 시의 자동 창 >

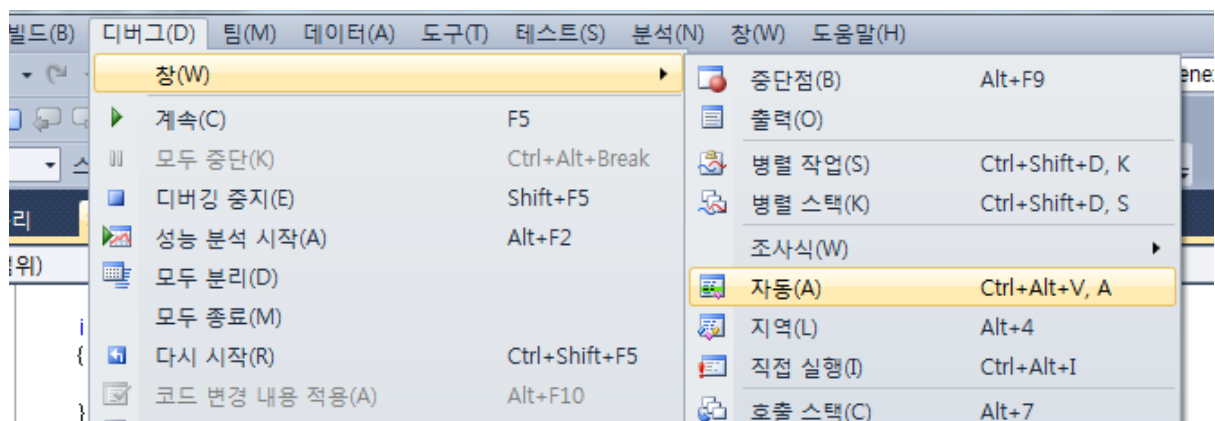
<그림 6>을 보면 중단점이 설정된 곳에서 변수 num2, num3, num4가 사용되고 있으므로 이들 변수의 내용이 표시되고 있다. num2의 값은 107, num3의 값은 -200, num4는 아직 함수 methodC()가 호출되지 않아서 -858993460 이라는 쓰레기 값을 가지고 있는 것을 알 수 있다.

디버깅 시 변수의 내용을 출력해주는 것은 자동 창 이외에도 지역 창을 통해서도 볼 수 있다. 지역 창은 현재 디버깅 시에 멈춘 곳에서 사용하는 모든 변수를 표시 해 준다.



< 그림 7. 디버깅 시의 지역 창 >

만약 자동 창이나 지역 창이 표시 되지 않을 때는 메뉴의 [디버그] -> [창]을 선택해서 보고 싶은 창을 선택하면 된다.



< 그림 8. 메뉴에서 디버그와 관련된 창 표시하기 >

중단점에서 멈춘 상태에서 프로그램을 계속 실행하기 위해서는 단축키 'F5' 키를 누르던가 아래와 같은 아이콘을 마우스로 누른다.



< 그림 9. 디버깅 시작, 계속, 종료 버튼 >

여기까지가 가장 간단하게 디버깅 하는 방법이다. 프로그램이 생각한대로 동작하지 않을 때 위에 설명한 방법처럼 디버깅을 하면 프로그램이 어떻게 동작하고 있는지 자세하게 알 수 있어서 버그가 발행한 위치와 어떤 것이 문제인지 쉽게 알 수 있다.

그럼 다음은 간단한 디버깅 방법에서 좀 더 디버깅을 편하게 할 수 있는 방법을 설명하겠다.

## 1.2 중단점 이후의 코드 디버깅

```

#include <iostream>

int methodA( int a )
{
    return a + 100;
}

int methodB()
{
    return -200;
}

int methodC( int a, int b )
{
    int c = 0;

    if( a < 0 )
    {
        a *= -1;
    }

    c = a * 1000;
    return c;
}

int main()
{
    int num1 = 7;

    int num2 = methodA( num1 );

    int num3 = methodB();

    int num4 = methodC( num2, num3 ); A. 여기에 중단점 설정

    int num5 = methodC( num3, num2 );

    getchar();
    return 0;
}

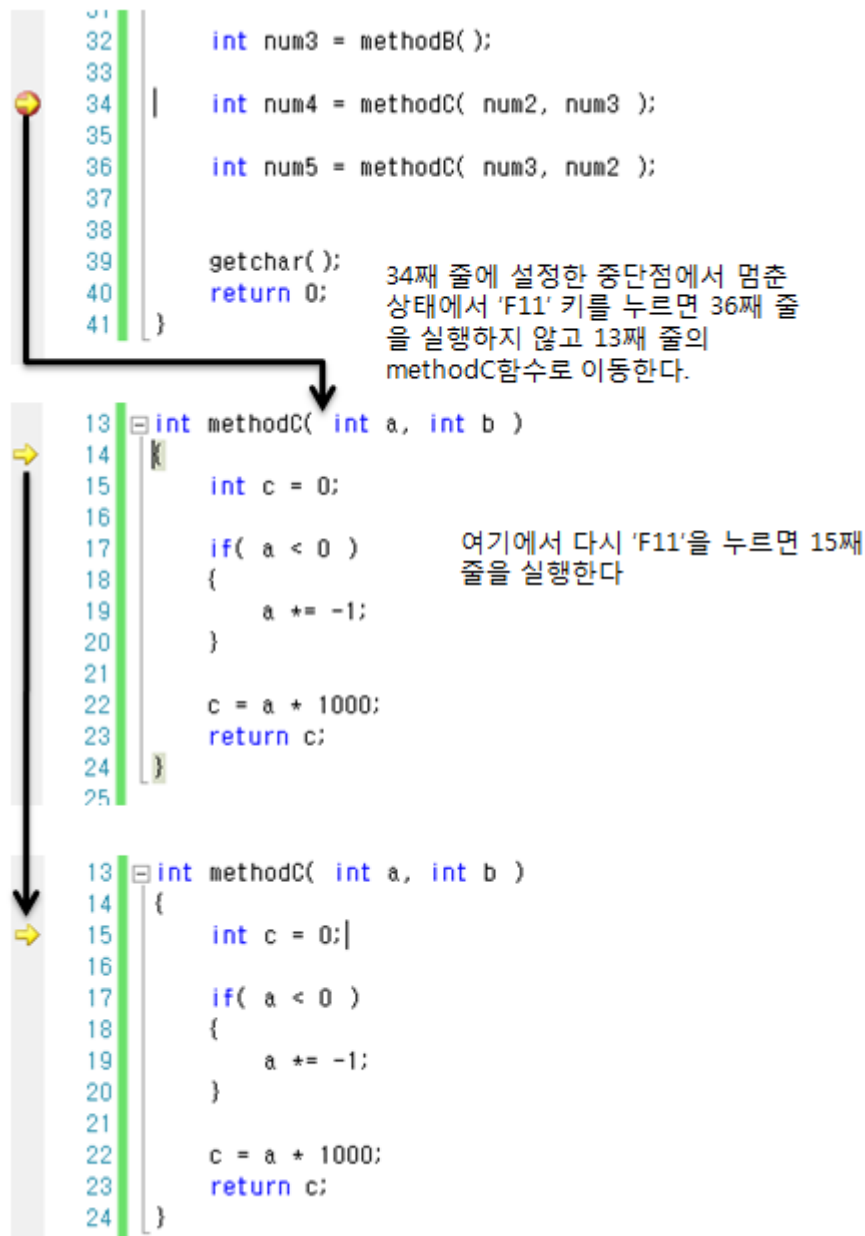
```

< 그림 10. >

디버깅 시 <그림 10>에서 설정한 중단점 이후 부분이 어떻게 동작하는지 세세하게 알고 싶어서 한 줄씩 프로그램을 실행하면서 변수의 내용을 보고 싶다면 어떻게 해야 할까? 디버깅을 멈추고 중단점을 더 추가해야 할까? 답은 그렇게 할 필요가 없다. 설정된 중단점 이후의 코드를 한 줄씩 실행하고 싶은 경우에는 중단점에 의해 멈춘 이후 'F5' 키로 계속하지 않고, '한 단계씩 코드 실행'(단축키 F11)나 '프로시저 단위 실행'(단축키 F10)을 선택하여 중단점 이후부터 소스코드를 한 줄씩 실행해 볼 수 있다.

### 1.3 한 단계씩 코드 실행

디버깅 시 중단점에서 중단된 이후 다음 코드를 한 줄씩 실행할 때 '한 단계씩 코드 실행'을 가리키는 'F11' 키를 누르면 중단점 이후에 함수를 호출하고 있으면 그 함수의 내부에 들어가서 한 줄씩 실행한다.



< 그림 11. '한 단계씩 코드 실행'의 흐름 >

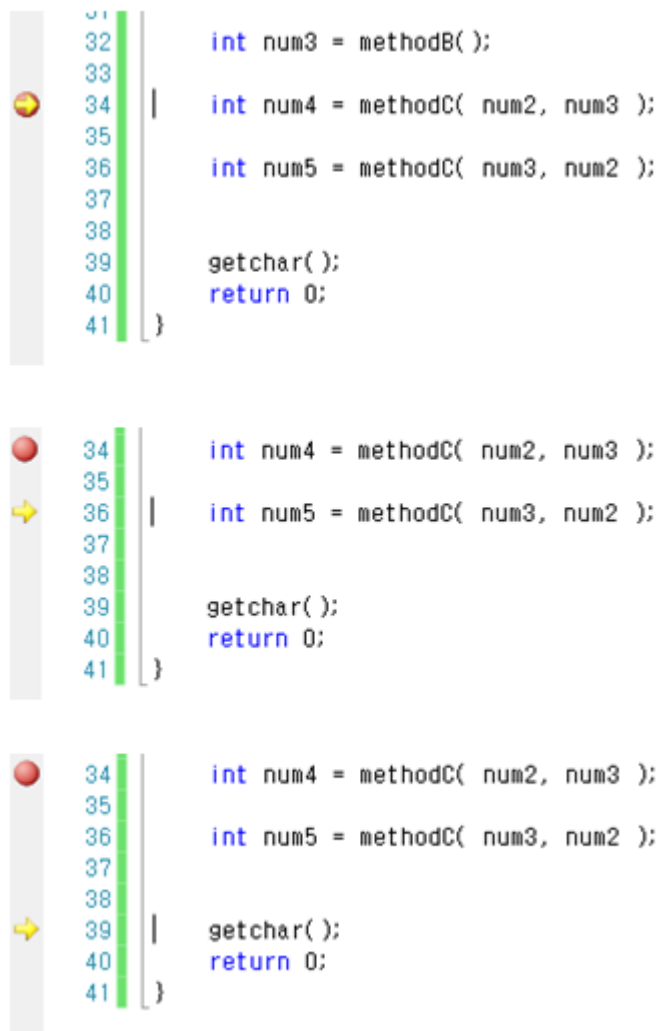
<그림 11>를 보면 34째 줄에 중단점이 설정되어 멈춘 후 F11 키를 누르면 34째 줄에서 호출하고 있는 methodC 함수로 이동하여 한 줄씩 실행되는 것을 볼 수 있다. 만약 methodC에서 나와서 34째 줄 다음에 있는 코드를 실행하고 싶다면 단축키 'shift + F11'을 누르면 된다.



이와 같이 '한 단계씩 코드 실행'은 눈에 보이는 한 줄씩 실행하지 않고 내부적으로 실행하는 모든 코드를 한 줄씩 실행하는 것이다. 그럼 <그림 10>의 34째 줄에서 멈춘 후 다음을 실행할 때 <그림 11>처럼 methodC로 이동하지 않고 보이는 대로 한 줄씩 실행하기 위해서는 어떻게 해야 할까? 이때는 '프로시저 단위 실행'을 하면 된다.

## 1.4 프로시저 단위 실행

중단점 이후의 코드를 실행할 때 보이는 줄 그대로 한 줄씩 실행하고 싶을 때는 F10 키를 눌러서 '프로시저 단위 실행'을 선택한다.



< 그림 12. '프로시저 단위 실행' >

<그림 12>를 보면 34째 줄에서 멈춘 후 F10을 누르면 36째 줄로 넘어가고 다시 F10을 누르면

다음째 줄인 39째 줄로 넘어가는 것을 볼 수 있다.

이것으로 VC++에서의 가장 기본적인 디버깅 방법을 다 배웠다. 여기까지만 알고 있으면 이제 디버깅을 할 수 있다고 자신 있게 말할 수 있다. 그러나 어디까지나 이것은 가장 기초적인 디버깅 방법으로 간단한 프로그램을 만들 때는 충분하지만 복잡한 프로그램에서는 이것만 알고 있으면 효율적으로 버그를 잡을 수 없다. 버그를 빨리 잡기 위해서는 VC++에서 제공하는 더 다양한 디버깅 기능을 배워야 한다.

그럼 지금까지 설명한 디버깅 방법을 머리에 확실하게 넣은 후 고급 디버깅 방법을 배워보자.

## 2. 디버깅 고급

혹시 고급이라고 해서 뭔가 어려운 것을 설명하지 않을까 라고 걱정하였다면 조금도 걱정하지 않아도 된다. 말이 고급이지 짧고 간단하여 정말 쉽다. VC++의 디버깅 기능이 좋은 이유는 디버깅을 쉽게 할 수 있기 때문이다. 그러나 대부분의 프로그래머는 기초적인 디버깅만 배우고, 그것만을 사용하고 있어서 VC++에서 제공하는 편리한 디버깅 기능을 낭비하고 있어서 많이 안타깝다.

다음에 설명하는 것을 잘 배운 후 디버깅 시에 필요하면 적극적으로 사용하기 바란다.

### 2.1 커서까지 실행

소스 코드에서 중단점을 설정하지 않고 특정 위치까지 실행하다가 멈추기를 원할 때가 있을 것이다. 보통 이런 경우는 코드를 분석하다가 특정 위치까지 프로그램이 어떻게 실행하는지 알고 싶을 때 유용하다.

사용 방법은 프로그램이 실행하다가 멈추기를 바라는 곳에 커서를 이동한 후 'Ctrl + F5' 키를 누르면 된다.

```

26 int main( )
27 {
28     int num1 = 7;
29
30     int num2 = methodA( num1 );
31
32     int num3 = methodB( );
33
34     int num4 = methodC( num2, num3 );
35
36     int num5 = methodC( num3, num2 );
37
38     getchar( );
39     return 0;
40 }
41

```

```

26 int main( )
27 {
28     int num1 = 7;
29
30     int num2 = methodA( num1 );
31
32     int num3 = methodB( );
33
34     int num4 = methodC( num2, num3 );
35
36     int num5 = methodC( num3, num2 );
37
38     getchar( );
39     return 0;
40 }
41

```

< 그림 13. 34째 줄에 커서를 놓은 후(왼쪽 그림) 'Ctrl + F10'을 누르면 오른쪽 그림처럼 34째 줄까지 실행하다가 멈춘다 >

일반적인 디버깅 방법으로는 일시적으로 조사하려는 부분이라도 중단점을 설정한 후 디버깅 실행을 하고, 그 부분을 다시 사용하지 않을 때는 중단점을 해제해야 하는데 '커서까지 실행'은 중단하기 원하는 부분에 커서를 이동한 후 'Ctrl + F10'으로 간단하게 디버깅 할 수 있다.

## 2.2 다음에 실행될 문 변경

디버깅 도중 현재 멈춘 곳의 앞 부분을 한 줄씩 실행해 보고 싶을 때는 디버깅을 종료하고 중단점을 추가해야 할까? 경우에 따라서는 디버깅을 끝내지 않고도 멈춘 곳의 앞 부분을 한 줄씩 실행할 수 있습니다. 바로 '다음에 실행될 문 변경' 기능을 사용하면 된다.

방법은 아래 그림과 같다.

```
9 int main()  
10 {  
11     int nNum1 = 100;  
12     nNum1 = CallA( nNum1 );  
13  
14     int Nums[100] = { 0, };  
15     다음에 실행될 문입니다. 다음에 실행될 문을 변경하려면 화살표를 끌어 오십시오. 이렇게 하  
16     {  
17         Nums[i] = i + nNum1;  
18     }  
19     ...
```

중단점에 마우스 커서를 이동시키면 중단점에 화살표와 위와 같은 메시지가 나옵니다. 이때 원하는 지점까지 마우스로 드래그 합니다.

```
9 int main()  
10 {  
11     int nNum1 = 100;  
12     nNum1 = CallA( nNum1 );  
13  
14     int Nums[100] = { 0, };  
15     for( int i = 0; i < 100; ++i )  
16     {  
17         Nums[i] = i + nNum1;  
18     }  
19
```

실행 위치가 14째 줄에서 12째 줄로 이동했다.

< 그림 14. 다음에 실행될 문 변경 >

다만 이 기능을 사용할 때는 주의해야 할 점이 있습니다. 이미 실행된 부분의 결과는 사라지지 않고 그대로 있다는 것이다.

```

3  int CallA( int nNum )
4  {
5      int nRetNum = nNum + 100;
6      return nRetNum;
7  }
8
9  int main()
10 {
11     int nNum1 = 100;
12     nNum1 = CallA( nNum1 );
13
14     int Nums[100] = { 0, };
15     for( int i = 0; i < 100; ++i )
16     {
17         Nums[i] = i + nNum1;
18     }
19
20     getchar();
21     return 0;
22 }

```

100 %

이름	값
nNum1	200
Nums	0x0054fc40

< 그림 15-1. 14째 줄에서 멈춘 상태에서 변수 nNum1의 값은 200 이다 >

```

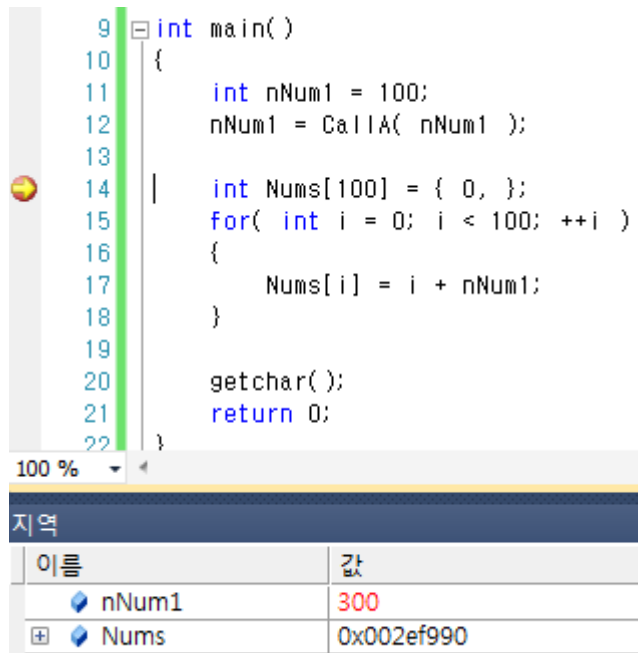
9  int main()
10 {
11     int nNum1 = 100;
12     nNum1 = CallA( nNum1 );
13
14     int Nums[100] = { 0, };
15     for( int i = 0; i < 100; ++i )
16     {
17         Nums[i] = i + nNum1;
18     }
19
20     getchar();
21     return 0;
22 }

```

100 %

이름	값
nNum1	200
Nums	0x002ef990

< 그림 15-2. 다음에 실행될 위치를 12째 줄로 이동. nNum1의 값은 200 이고, 함수 CallA는 실행 전 이다 >



< 그림 15-3. 12째 줄을 실행하면 nNum1의 값은 300이 된다 >

위 그림을 보면 알 수 있듯이 코드에서는 nNum1의 값은 100 이다. 그래서 처음 실행될 때 CallA 함수에 전달 되는 nNum1의 값은 100 이므로 12째 줄이 실행되면 nNum1의 값은 200이 된다. 그러나 <그림 15-2>처럼 실행 위치를 12째 줄로 이동하면 CallA 함수에 200 이라는 값을 가진 nNum1을 전달하여 12째 줄을 다시 실행하면 nNum1은 300 이 된다. 이처럼 한번 실행된 결과는 사라지지 않고 그대로 적용됨을 알 수 있다.

## 2.3 포인터 배열의 내용 보기

```

14     int Nums[100] = { 0, };
15     int* pNums = new int[100];
16
17     for( int i = 0; i < 100; ++i )
18     {
19         Nums[i] = i + nNum1;
20         pNums[i] = i + nNum1;
21     }
22
23     getchar();
24     delete[] pNums;
25     return 0;
26 }

```

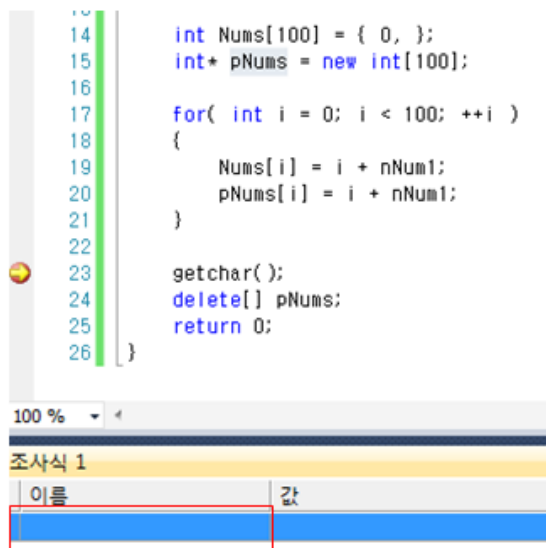
100 %

지역

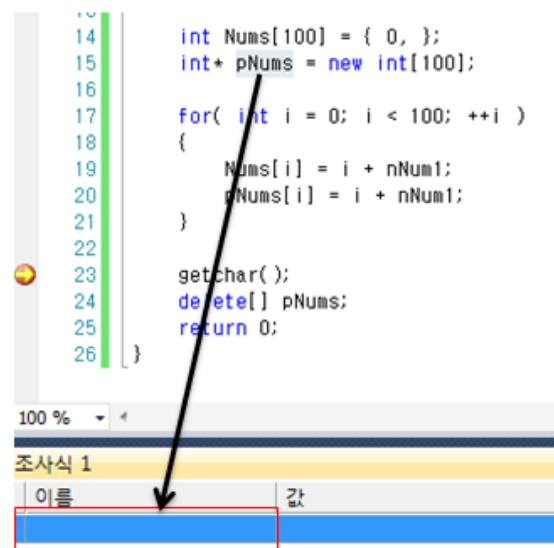
이름	값
i	100
pNums	0x005b1368
nNum1	200
Nums	0x0035f764
Nums[0]	200
Nums[1]	201
Nums[2]	202
Nums[3]	203
Nums[4]	204
Nums[5]	205
Nums[6]	206
Nums[7]	207

< 그림 16. >

<그림 16>을 보면 지역 창에 변수 Nums와 pNums의 값을 볼 수 있습니다. 그런데 Nums의 경우 배열의 각 요소의 값을 보여주지만 포인터인 pNums의 경우 첫 번째 요소의 값만 보여주고 있습니다. pNums의 각 요소의 값을 보고 싶을 때는 '조사식' 창을 사용해야 합니다. 조사식 창이 보이지 않는 경우 메뉴의 [디버그] -> [창] -> [조사식]을 선택합니다.



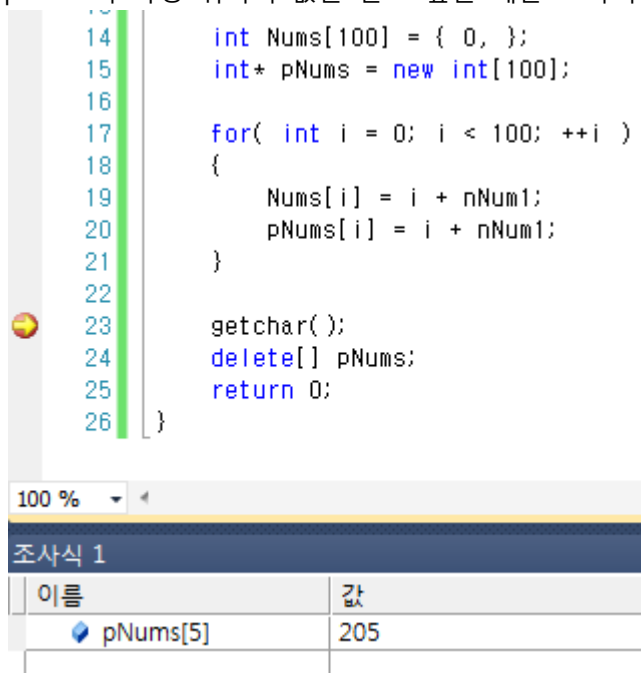
변수 pNums의 값을 보기 위해서는 이 부분을 마우스로 클릭한 후 pNums를 입력한다.



마우스로 pNums를 선택 후 드래그 한다.

< 그림 17. 조사식 사용 방법 >

pNums의 특정 위치의 값을 알고 싶을 때는 조사식에 'pNums[ 위치 ]'를 입력한다.



< 그림 18. pNums의 6번째 값 보기 >

pNums의 보고 싶은 행들의 값을 보고 싶을 때는 'pNums, 보고 싶은 행 개수'를 조사식에 입력한다.



```

14     int Nums[100] = { 0, };
15     int* pNums = new int[100];
16
17     for( int i = 0; i < 100; ++i )
18     {
19         Nums[i] = i + nNum1;
20         pNums[i] = i + nNum1;
21     }
22
23     getchar();
24     delete[] pNums;
25     return 0;
26 }

```

100 %

조사식 1

이름	값
pNums, 100	0x005b1368
[0]	200
[1]	201
[2]	202
[3]	203
[4]	204
[5]	205
[6]	206
[7]	207

< 그림 19. pNums의 모든 값을 보기 >

위의 경우는 pNums의 복수 개의 행의 값을 볼 수 있어서 좋지만 너무 많이 보여줘서 불편한 점도 있다. 만약 pNums의 11번째 위치 이후에서 10개만 보고 싶을 때는 'pNums + 위치, 보고 싶은 행 개수'를 조사식에 입력한다.

```

14     int Nums[100] = { 0, };
15     int* pNums = new int[100];
16
17     for( int i = 0; i < 100; ++i )
18     {
19         Nums[i] = i + nNum1;
20         pNums[i] = i + nNum1;
21     }
22
23     getchar();
24     delete[] pNums;
25     return 0;
26 }

```

100 %

조사식 1

이름	값
pNums+10, 10	0x005b1390
[0]	210
[1]	211
[2]	212
[3]	213
[4]	214
[5]	215
[6]	216
[7]	217
[8]	218
[9]	219

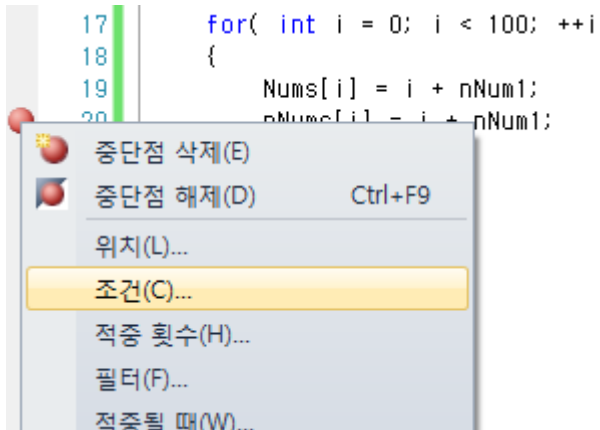
< 그림 20. pNums의 11번째 위치부터 10개의 값 보기 >

## 2.4 중단점 조건

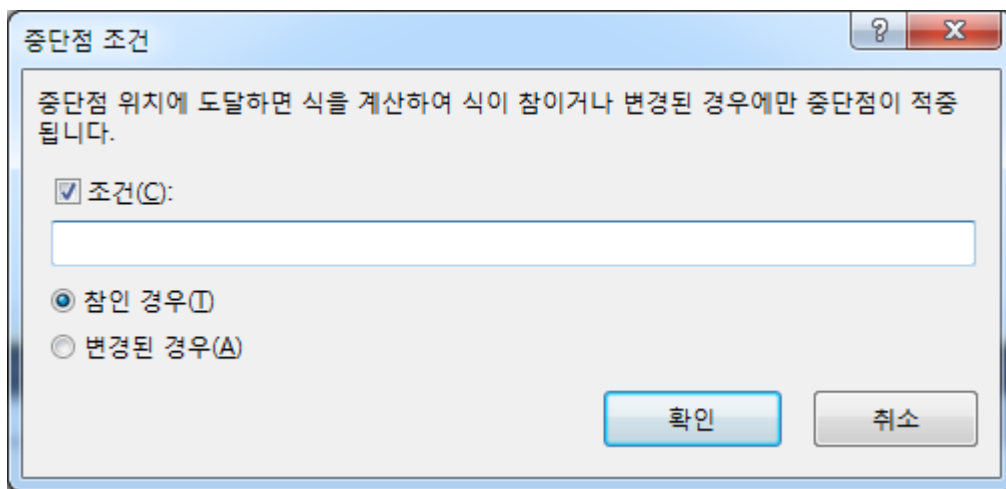
디버깅 시에 중단점이 설정된 곳이 매번 멈추지 않고 변수의 값이 변경되는 경우나 변수의 값이 특정 값이 될 때에만 멈추기를 원할 때는 중단점에 '조건'을 설정한다.

조건은 특정 변수의 값이 변하였던가 또는 특정 값이 될 때의 두 가지이다.

조건을 설정하는 방법은 중단점을 설정한 후 중단점을 마우스 오른쪽 버튼으로 클릭하여 나오는 팝업 메뉴 중 '조건'을 선택하면 조건을 설정하는 창이 나온다.



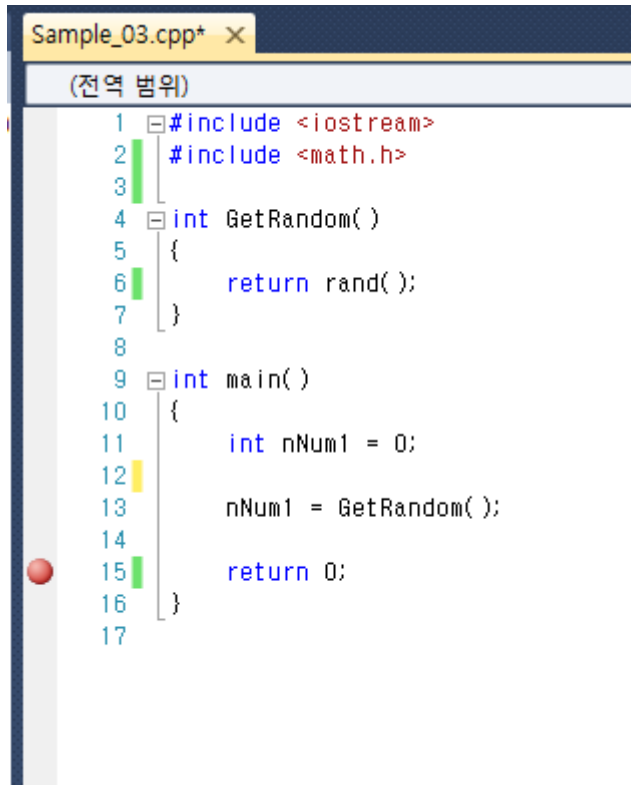
< 그림 21. 중단점에 조건 설정을 위해 팝업 메뉴에서 선택 >



< 그림 22. 중단점 조건 >

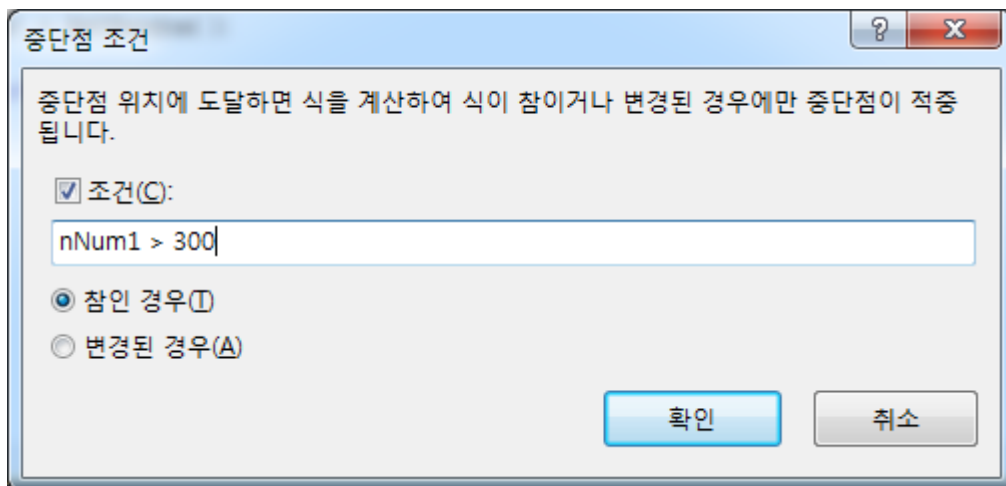
중단점 조건 중 '참인 경우'와 '변경된 경우'가 어떤 것인지 아직 이해가 안 되는 사람을 위해 예를 들어 보이겠다.

## 중단점 조건 – 참인 경우



< 그림 23. 중단점 조건 - '참인 경우' 예제 코드 >

<그림 23>에는 15째 줄이 중단점이 설정 되어있다. 디버깅 시에 13째 줄이 실행되어 nNum1의 값이 300이 되는 경우에만 멈추고 싶은 경우에는 아래와 같이 조건을 설정한다.



< 그림 24. 중단점 조건 - 참인 경우 설정 >

이제 디버깅 실행을 하면 nNum1이 300 보다 큰 경우에만 중단점을 설정한 부분을 실행할 때 멈추고 300 이하라면 멈추지 않게 된다.

중단점 조건이 설정되면 중단점의 모습이 아래 그림처럼 바뀐다.

```

14 |
15 | | return 0;
16 | }

```

< 그림 25. 조건이 설정된 중단점 모습 >

## 중단점 조건 - 변경된 경우

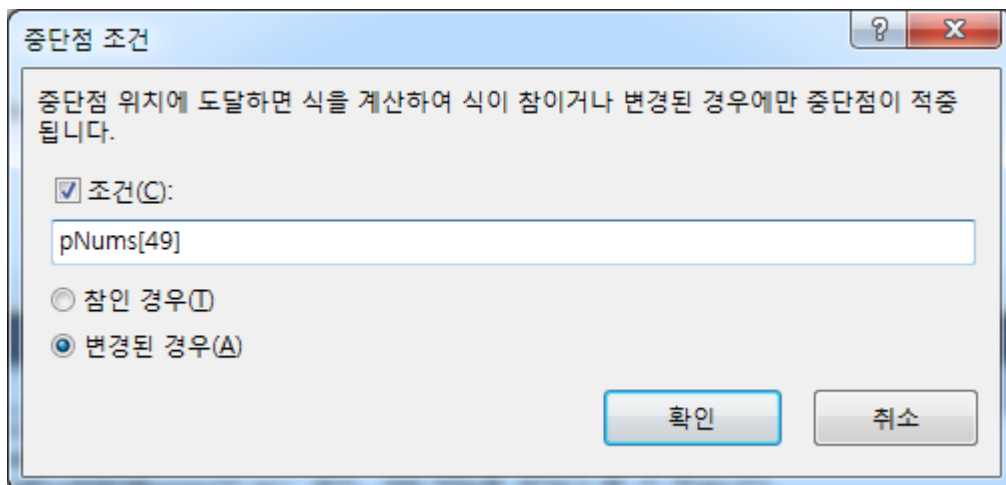
```

13 |
14 | int Nums[100] = { 0, };
15 | int* pNums = new int[100];
16 |
17 | for( int i = 0; i < 100; ++i )
18 | {
19 |     Nums[i] = i + nNum1;
20 |     pNums[i] = i + nNum1;
21 | }
22 |

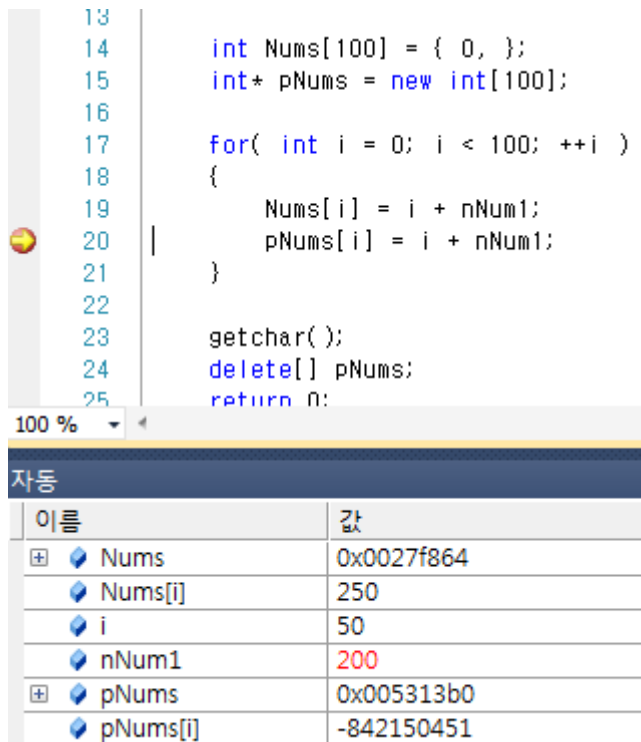
```

< 그림 24. 중단점 조건 - 변경된 경우 코드 >

<그림 24>에는 20째 줄에 중단점이 설정되어 있다. 여기서는 pNums의 50번째 위치의 값이 변경된 경우 중단 되도록 조건을 설정하겠다.



< 그림 25. pNums[49]의 값이 변경된 경우 중단되도록 설정 >

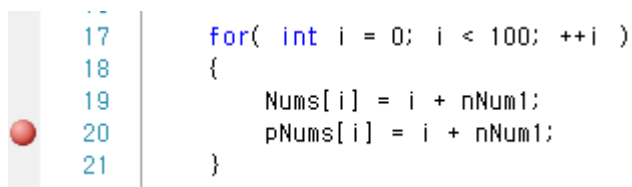


< 그림 26. <그림 24>의 코드를 디버깅 >

<그림 26>을 보면 20째줄에서 pNums[50]에 값을 대입할 때 pNums[49]의 값이 변경된 것을 감지하여 중단 되었다.

이렇게 중단점 조건을 사용하면 특수한 상황을 디버깅할 때 편리하다. 만약 중단점 조건을 사용하지 않으면 특수한 상황이 될 때까지 '디버깅 시작' -> '디버깅 중단'을 반복하게 되어 디버깅에 많은 시간을 소비하게 될 것이다.

## 2.5 적중 횟수

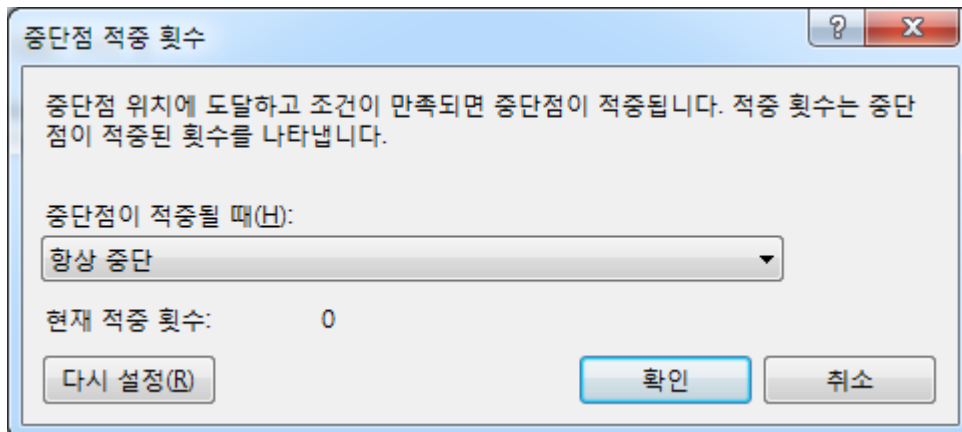


< 그림 27. 반복문에서 디버깅 >

<그림 27>과 같이 반복문 안에 중단점을 설정하여 디버깅 할 때 매번 중단되지 않고 특정 횟수 일 때 중단 되기를 원할 때에는 '적중 횟수'를 사용한다.

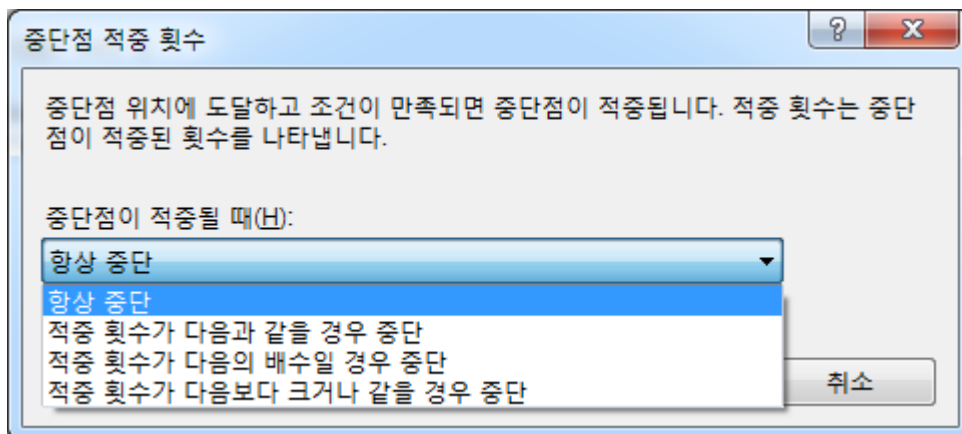
적중 횟수는 <그림 21>처럼 중단점의 팝업 메뉴에서 선택할 수 있다.

적중 횟수 메뉴를 선택하면 아래와 같은 창이 표시된다.



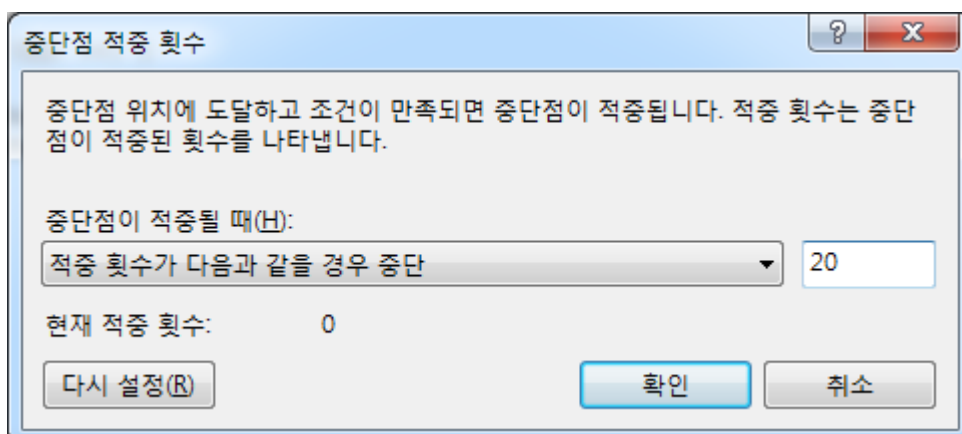
< 그림 28. 종단점 적중 횟수 창 >

기본은 항상 중단이지만 아래와 같이 옵션을 설정할 수 있다.



< 그림 29. 종단점 적중 횟수 옵션 >

만약 '적중 횟수가 다음과 같은 경우 중단'을 선택하면 아래와 같이 텍스트 박스가 표시되고 여기에 횟수를 입력하면 디버깅 시에 설정한 적중 횟수 일 때만 중단 된다.



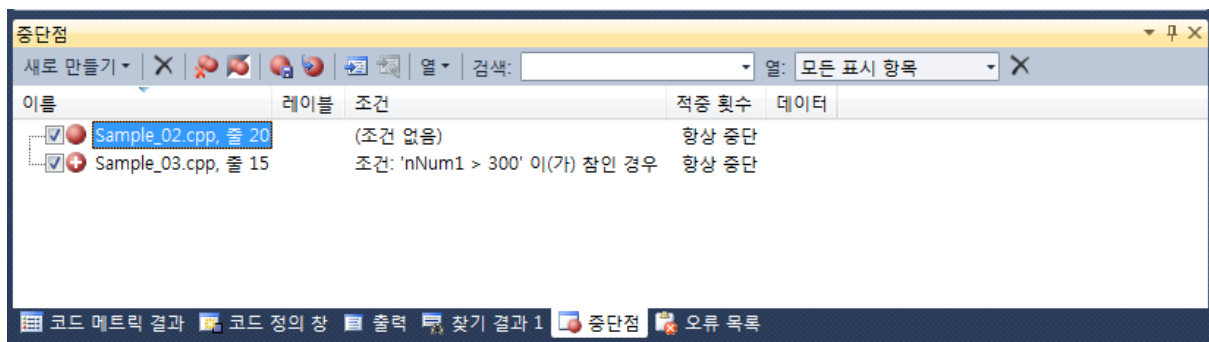
< 그림 30. 종단점 횟수 설정 >

적중 횟수는 반복문 안에서 디버깅할 때 아주 유용한 기능이다.

## 2.6 중단점 창

현업에서 만드는 프로그램들은 대부분 규모가 크고 복잡하다. 그래서 디버깅을 하는 경우 코드의 많은 부분에 중단점을 설정해야 하는 경우가 생긴다. 중단점이 많아지면 이것들을 관리하는 것도 중요해진다. 중단점을 어디 어디에 설정했는지, 중단점에 조건은 있는지? 등등

설정된 중단점 정보들은 '중단점 창'을 통해서 볼 수 있다.



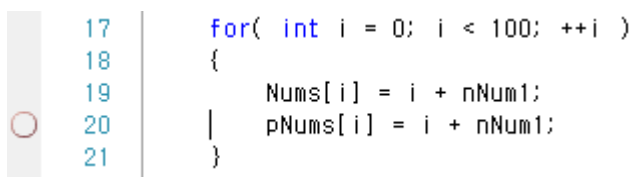
< 그림 31. 중단점 창 >

중단점 창이 보이지 않을 때는 메뉴의 [디버그] -> [창] -> [중단점]을 선택한다.

## 2.7 중단점 비 활성화 시키기

중단점을 삭제하지 않고 일시적으로 중단점을 비활성화 시키고 싶을 때가 있다. 방법은 해당 중단점 위치에 커서를 이동 시킨 후 단축키 'Ctrl+ F9'를 누르면 된다.

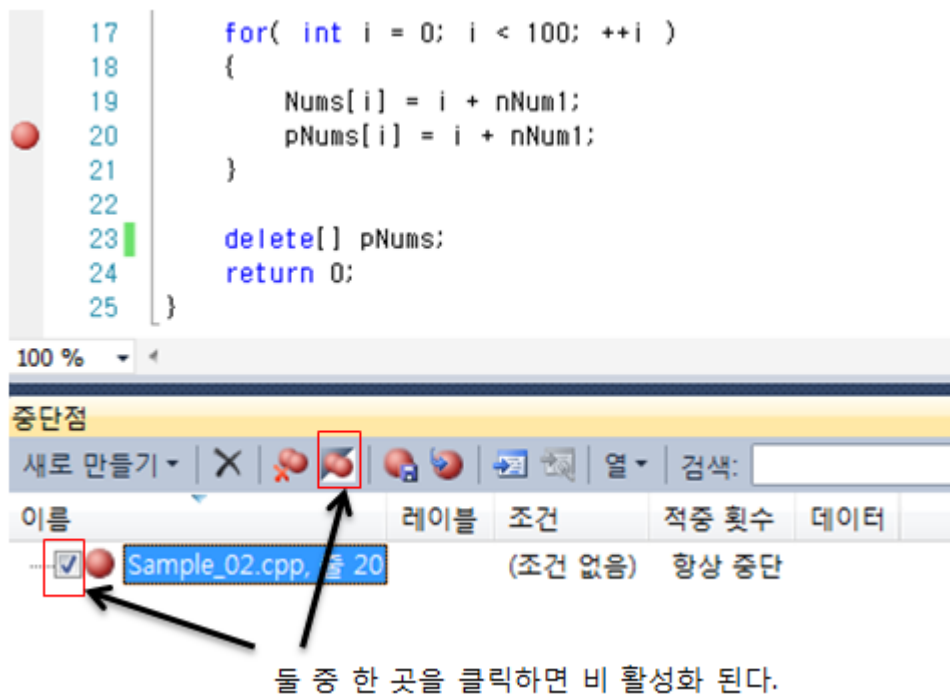
중단점이 비활성화 되면 중단점이 색이 없는 원이 된다.



< 그림 32. 비 활성화된 중단점 >



다시 활성화 시키고 싶다면 다시 'Ctrl + F9'를 누르면 활성화 된다.  
중단점 비 활성화는 중단점 창에서도 할 수 있다.

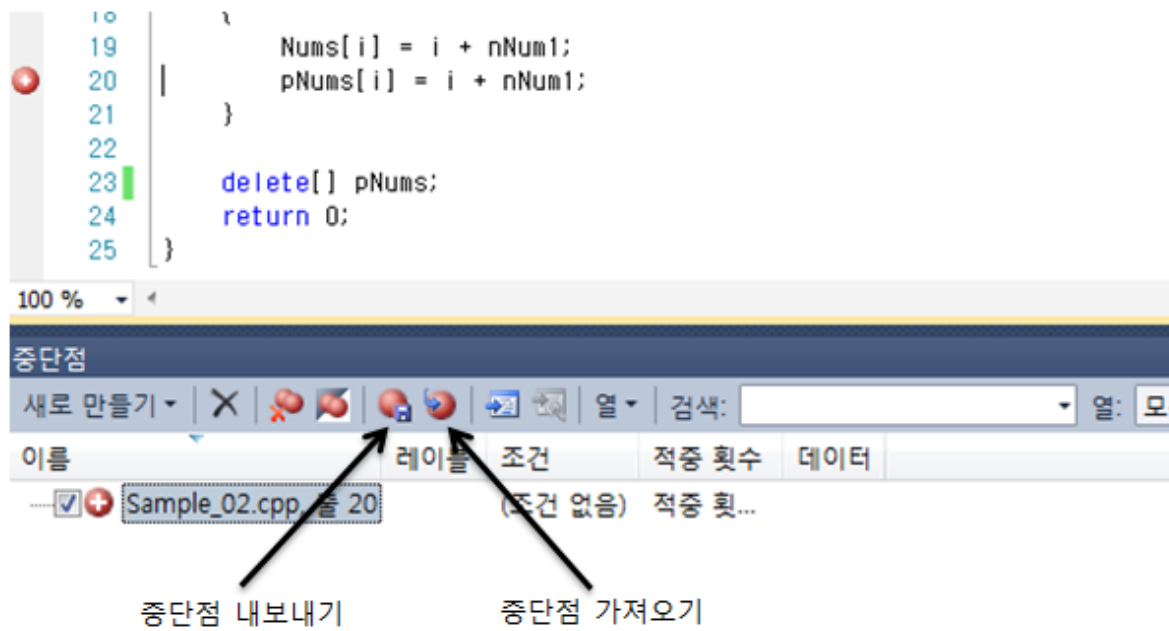


< 그림 33. 중단점 창을 통한 중단점 비 활성화 >

## 2.8 중단점 내보내기/가져오기

중단점을 지금은 사용하지 않지만 다음에 사용할 확률이 높은 경우, 또는 같이 작업하는 동료의 컴퓨터의 VC++에 자신이 디버깅에 사용했던 중단점을 설정해야 하는 경우 등에는 중단점 내보내기/가져오기 기능을 사용한다.

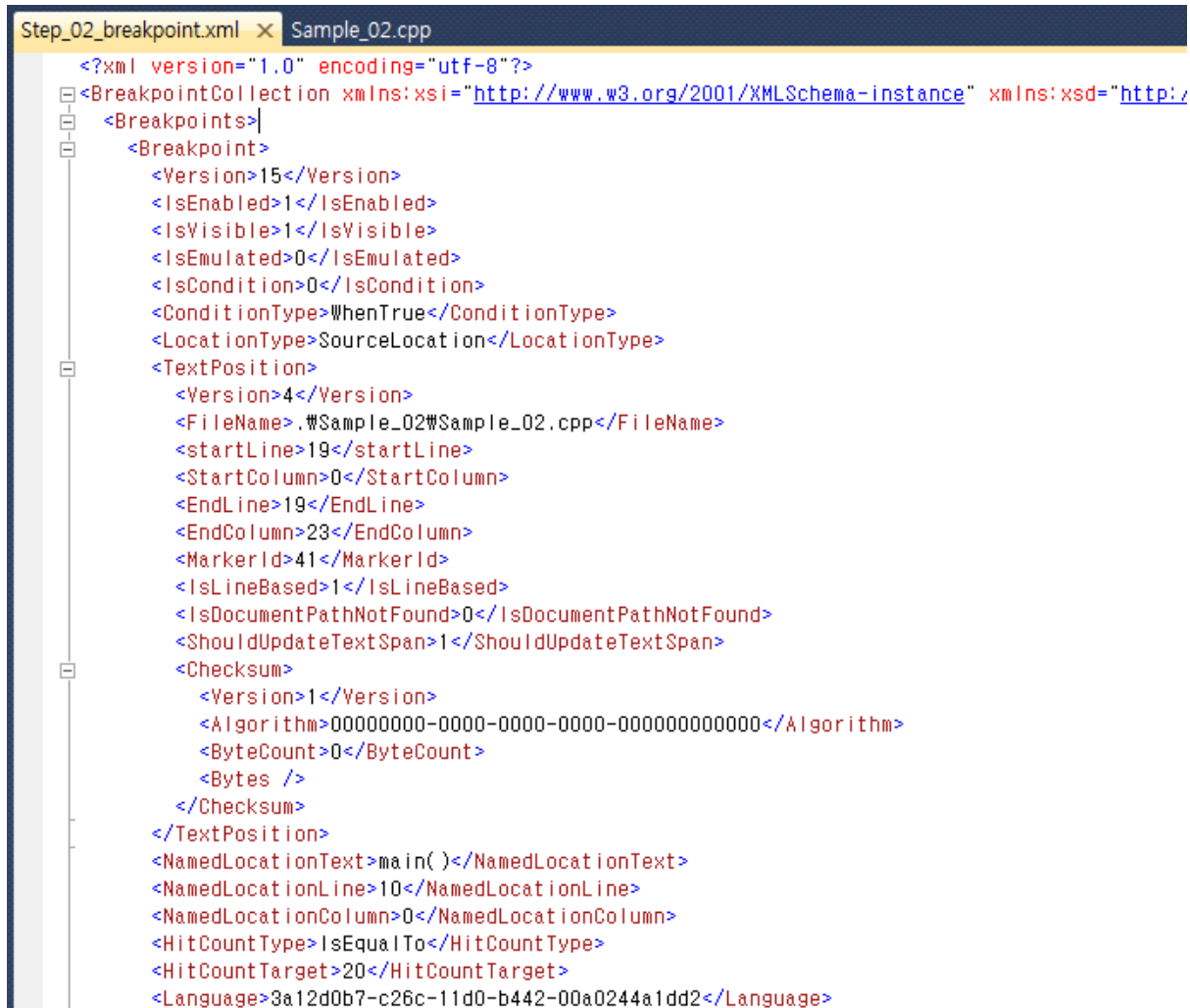
중단점 정보는 xml 파일 포맷으로 내보낼 수 있고 가져올 수 있다.  
중단점 내보내기/가져오기는 중단점 창을 통해서 쉽게 할 수 있다.



< 그림 34. 중단점 창을 통해 중단점 내보내기/가져오기 >

중단점 창을 통해서 중단점을 내보낼 때는 중단점 창에 표시된 중단점들만을 내보낸다(뒤에 설명할 레이블을 사용하면 특정 중단점만 중단점 창에 표시할 수 있다).

아래 그림은 내보낸 중단점 xml 파일의 내용의 일부이다.



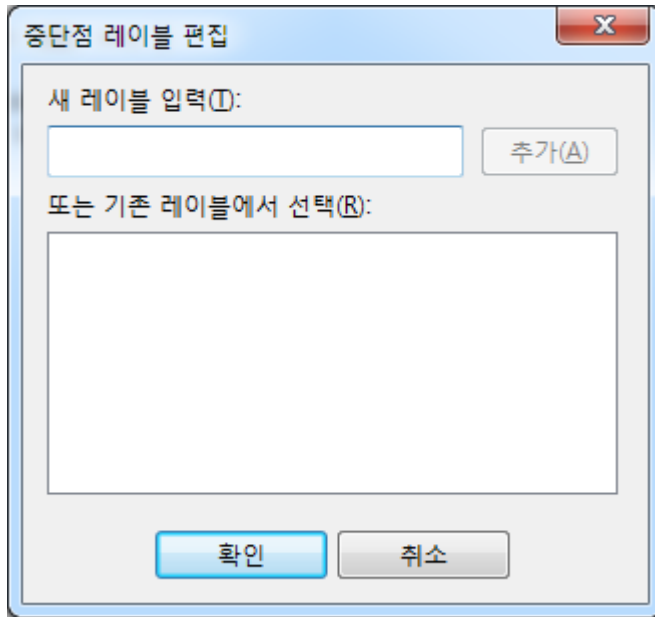
< 그림 35. 내보내진 중단점의 xml 파일 정보 >

## 2.9 레이블

중단점에는 '레이블'(이름)을 지정할 수 있다. 레이블을 지정하면 어떤 목적을 가진 중단점인지 또는 비슷한 목적의 중단점을 그룹별로 관리할 수 있다.

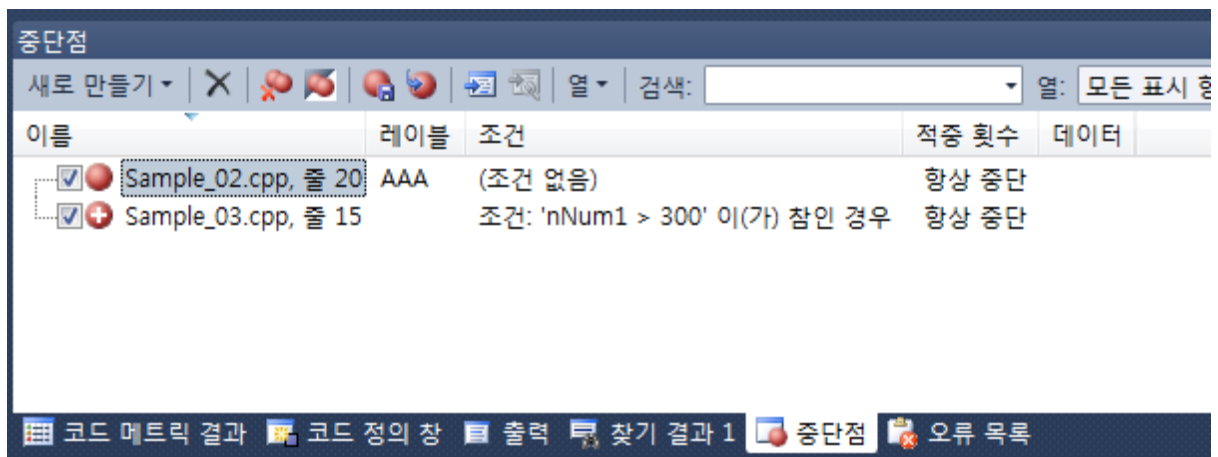
### 레이블 설정

중단점을 마우스 오른쪽 클릭하면 표시되는 팝업 메뉴에서 '레이블 편집'을 선택하면 '중단점 레이블 편집' 창이 표시됩니다. 여기서 새로운 레이블 이름을 입력한다.



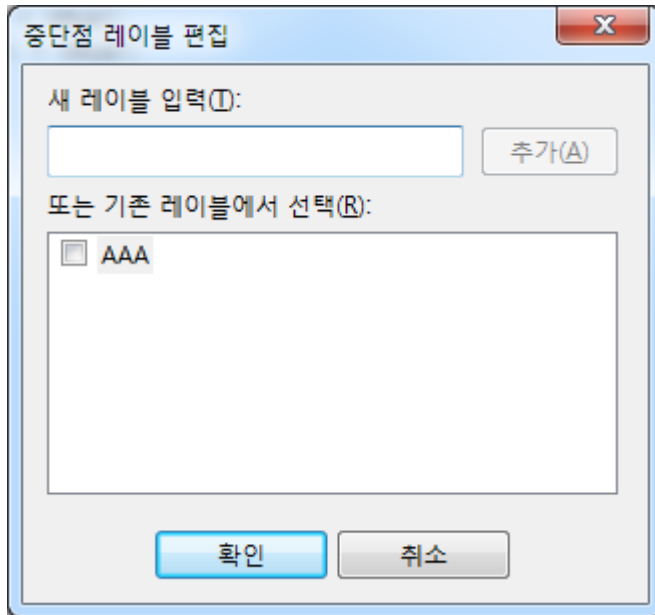
< 그림 36. 중단점 레이블 편집 창 >

레이블을 입력하고 '확인' 버튼을 누르면 중단점 창에 레이블이 표시됩니다.



< 그림 37. 레이블이 표시된 중단점 창. <그림 36>에서 레이블을 'AAA'라고 입력 >

만들어 놓았던 레이블은 재사용할 수 있다. 다른 중단점에 같은 레이블을 사용할 때는 기존의 것을 선택하면 된다.



< 그림 38. 앞에서 'AAA' 레이블을 만들어 놓아서 선택할 수 있다 >

## 그룹별 관리

서로 중단점끼리 비슷한 목적을 가진 경우는 같은 레이블을 지정하여 그룹별로 관리할 수 있다.

```

36   tstring DBIP = configReader.GetValue<tstring>(ACE_TEXT("IP")); //직접 입력할 때
37   tstring DBUserID = configReader.GetValue<tstring>(ACE_TEXT("ID"));
38   tstring DBUserPW = configReader.GetValue<tstring>(ACE_TEXT("PASSWORD"));
39   tstring DATABASE = configReader.GetValue<tstring>(ACE_TEXT("DATABASE"));
40   INT32 nConnectTimeOut = configReader.GetValue<INT32>(ACE_TEXT("CONNECT_TIMEOUT"));
41   bool IsRetryConnection = configReader.GetValue<INT32>(ACE_TEXT("IS_RETRY_CONNE"));
42   INT32 nMaxConnectPoolCount = configReader.GetValue<INT32>(ACE_TEXT("MAX_CONNEC"));
43
44   //
45   SAdoConfig adoconfig;
46   adoconfig.SetIP( DBIP.c_str() );

```

중단점

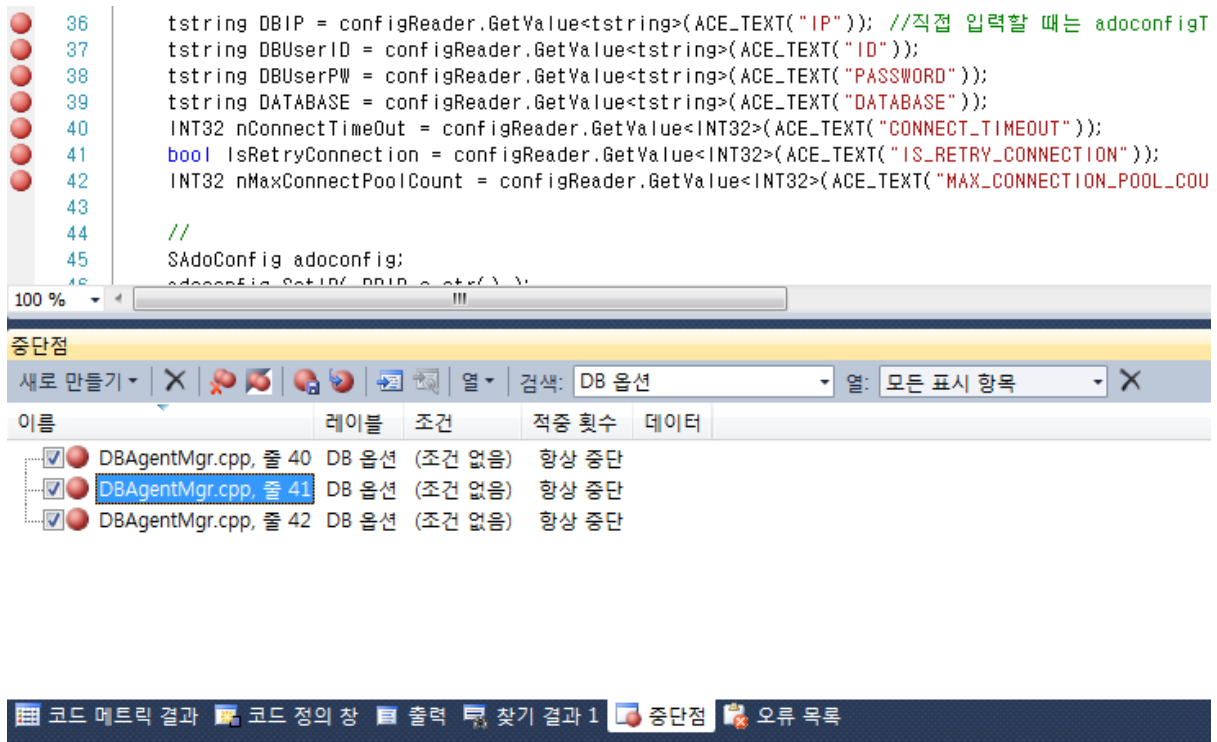
새로 만들기 | [Icons] | 열 | 검색: | 열: 모든 표시 항목

이름	레이블	조건	적중 횟수	데이터
Client.cpp, 줄 76		(조건 없음)	항상 중단	
DBAgentMgr.cpp, 줄 36	DB 주소	(조건 없음)	항상 중단	
DBAgentMgr.cpp, 줄 37	DB 주소	(조건 없음)	항상 중단	
DBAgentMgr.cpp, 줄 38	DB 주소	(조건 없음)	항상 중단	
DBAgentMgr.cpp, 줄 39	DB 주소	(조건 없음)	항상 중단	
DBAgentMgr.cpp, 줄 40	DB 옵션	(조건 없음)	항상 중단	
DBAgentMgr.cpp, 줄 41	DB 옵션	(조건 없음)	항상 중단	
DBAgentMgr.cpp, 줄 42	DB 옵션	(조건 없음)	항상 중단	

코드 메트릭 결과 | 코드 정의 창 | 출력 | 찾기 결과 1 | 중단점 | 오류 목록

< 그림 39. 같은 목적의 중단점들은 같은 레이블을 설정 >

<그림 39>를 보면 36~39째 줄까지의 중단점은 DB 주소 디버깅을 위한 것, 40 ~ 42째 줄까지의 중단점은 DB 옵션 디버깅을 위한 중단점이라는 것을 쉽게 알 수 있다. 레이블을 통해서 중단점들을 그룹화 시키면 중단점 창에서 특정 중단점만 표시하여 중단점 정보를 쉽게 파악할 수 있다. 특정 레이블의 중단점만 표시하고 싶을 때는 '검색' 창을 통해서 원하는 레이블을 입력한다.

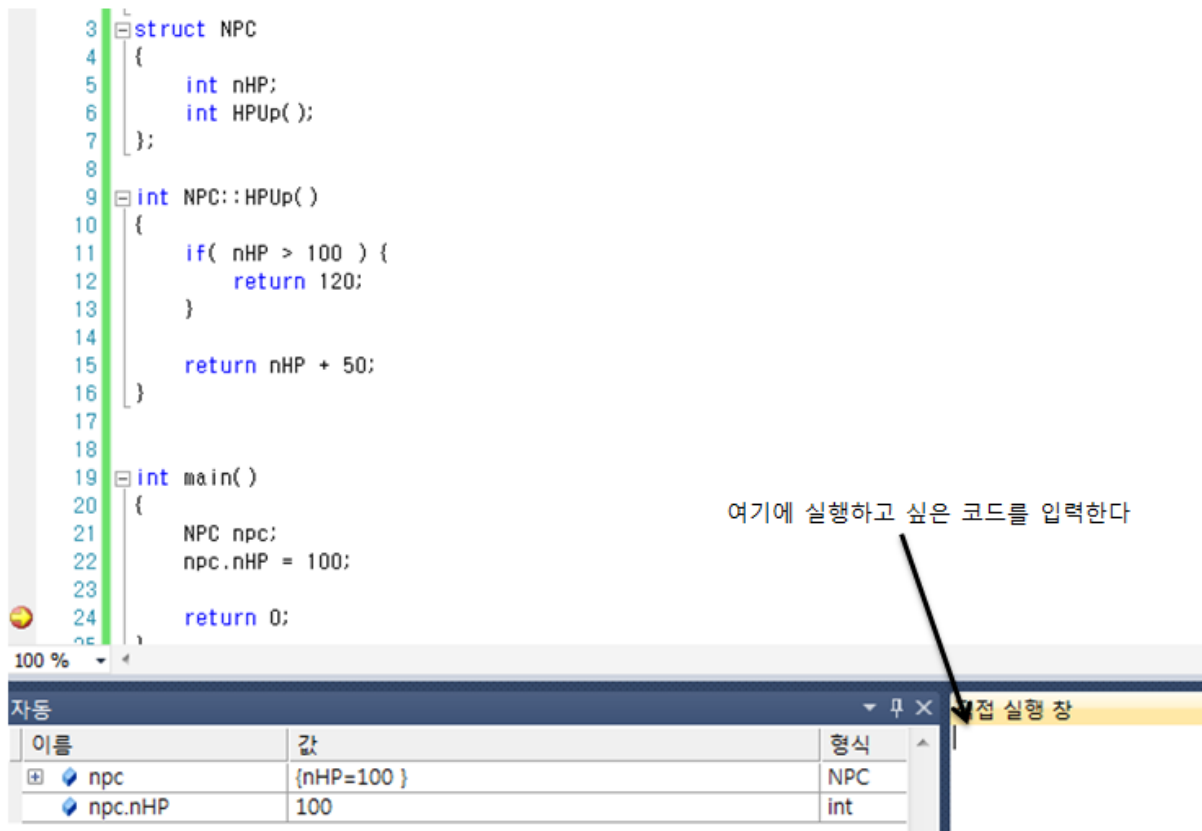


< 그림 40. DB 옵션 레이블만 표시 >

## 2.10 직접 실행

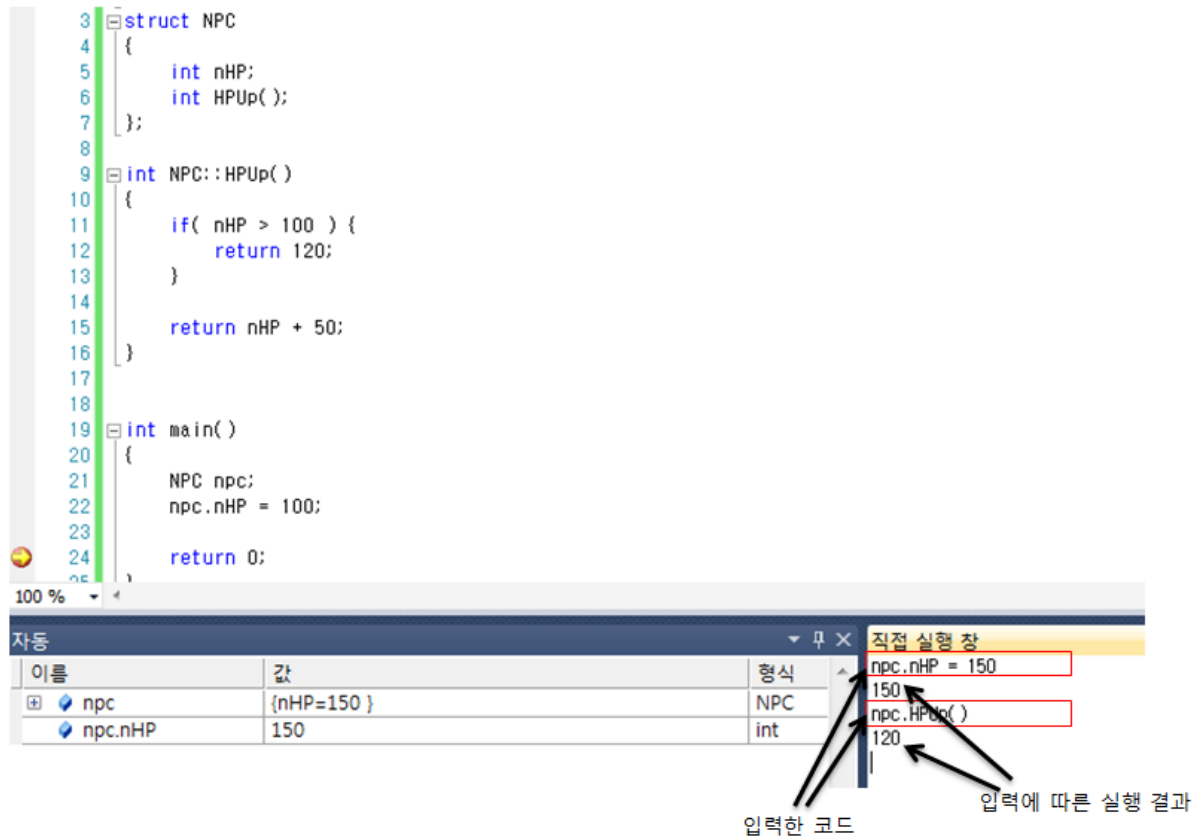
디버깅 중 코드에 코딩 하지 않고 어떤 결과가 나오는지 테스트해 봐야 할 때가 있는데 이럴 때 '직접 실행' 기능을 사용하면 좋다.

직접 실행 창이 보이지 않는 경우는 메뉴의 [디버그] -> [창] -> [직접 실행]을 선택하면 된다.



< 그림 41. 디버깅 중의 직접 실행 창 모습 >

디버깅 중 임의로 특정 코드가 어떤 결과가 나오는지 알고 싶을 때는 <그림 41>에 보이는 '직접 실행 창'에 코드를 입력해 본다.



< 그림 42. 직접 실행 창에 코드 입력 >

<그림 42>는 디버깅 중 직접 실행 창에 npc 객체의 nHP 멤버에 실제 코드와는 다른 값을 설정한 후 npc 객체의 HPU() 멤버를 실행하여 어떤 결과가 나오는지 확인하였다.

참고로 < 그림 42>를 보면 알듯이 직접 실행 창에서 현재 사용 중인 변수의 값을 변경하면 실제로 적용된다는 것을 알 수 있다( <원래 코드에서는 npc의 nHP 멤버에 100을 설정했는데 직접 실행 창에서 150을 설정하여 <그림 42>의 오른쪽 밑의 자동 창을 보면 npc의 nHP가 150으로 되어 있다. 직접 실행을 하기 전인 <그림 41>을 보면 nHP는 코드대로 100으로 되어 있다).

직접 실행을 할 때 하나 조심해야 할 것이 있다. <그림 41>에 있는 NPC 구조체의 멤버 중 int HPU() 멤버가 인라인으로 되어 있으면 직접 실행에서 HPU() 멤버를 호출할 수 없다.

<코드>

```
struct NPC
```

```
{
```

```
    int nHP;
```

```
    int HPU()
```

```
{
```



```

        if( nHP > 100 ) {
            return 120;
        }

        return nHP + 50;
    }
};
</코드>

```

위 코드는 NPC 선언 안에 HPUp() 멤버를 정의해서 암묵적으로 HPUp() 멤버는 인라인화 된다

## 2.11 DataTips

회사에서 일을 할 때 기억해야 할 것이 있으면 그 내용이 길지 않다면 보통 메모지를(포스트잇 같은) 모니터나 책상에 붙여 놓는다. 메모지는 우리가 눈에 잘 띄는 곳에 놓아두기 때문에 잊어버리지 않고 볼 수 있다. 디버깅을 할 때도 중요한 사항은 메모를 하고 그것을 눈에 잘 띄는 곳에 놓아두면 좋을 것이다. 디버깅과 관련된 정보를 눈에 잘 띄는 곳에 둔다면 그것은 VC++에 두면 될 것이다. 이럴 때 사용하는 것이 DataTips 기능이다.

DataTips는 디버깅 할 때에는 필요한 것을 메모하고 이것은 디버깅할 때만 보이고 디버깅이 끝나도 사라지지 않는다. 말보다는 직접 보는 것이 쉽게 이해가 갈 테니 다음 그림을 보기 바란다.

```

32 void HBPlayerMgr::Release( )
33 {
34     size_t nCount = m_Players.size();
35     for( size_t i = 0; i < nCount; ++i )
36     {
37         SAFE_DELETE( m_Players[i] );
38     }
39 }
40

```

디버깅 중 중단점에서 멈춘 상태에서 마우스 커서를 화살표가 가리키는 변수에 위치 시키면 위처럼 **nCount 0**가 표시된다.

```

32 void HBPlayerMgr::Release( )
33 {
34     size_t nCount = m_Players.size();
35     for( size_t i = 0; i < nCount; ++i )
36     {
37         SAFE_DELETE( m_Players[i] );
38     }
39 }
40

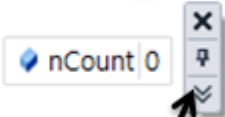
```

누르면 아래 그림처럼 된다.

```

32 void HBPlayerMgr::Release( )
33 {
34     size_t nCount = m_Players.size();
35     for( size_t i = 0; i < nCount; ++i )
36     {
37         SAFE_DELETE( m_Players[i] );
38     }
39 }

```

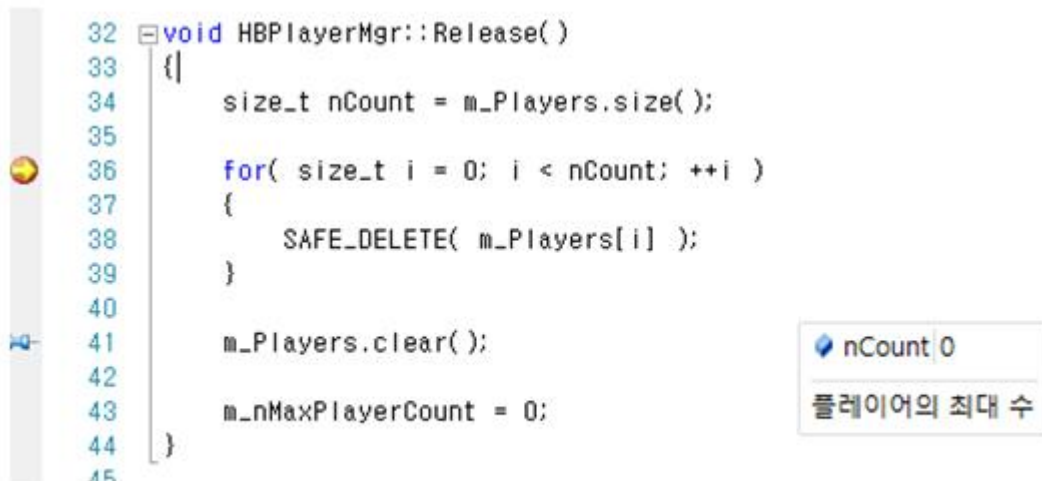
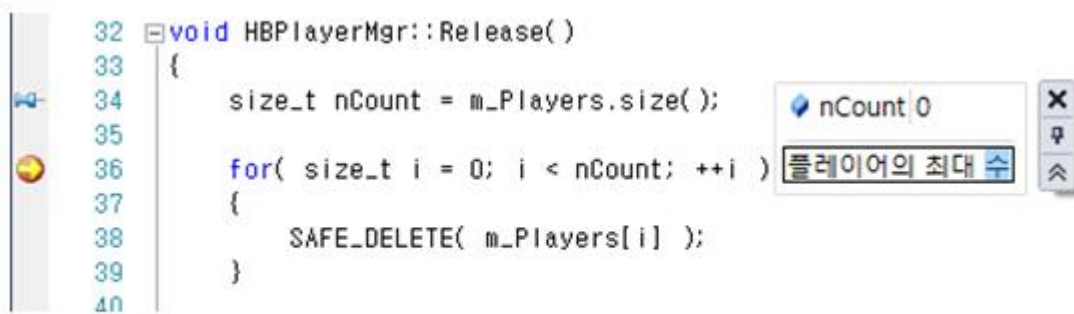


누르면 메모 창 이 보인다

DataTips가 설정되었다는 표시

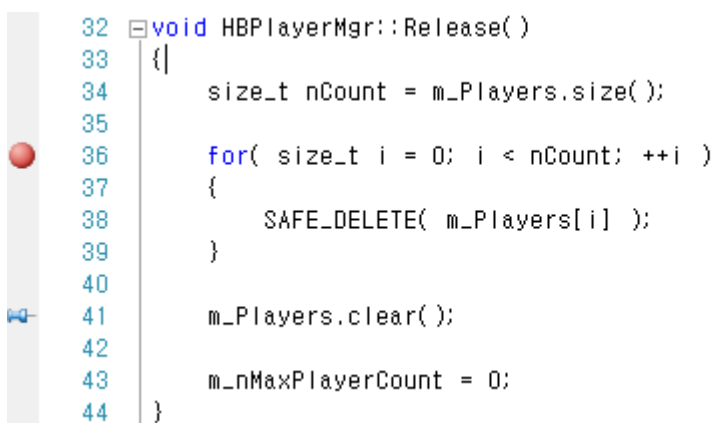
< 그림 43. DataTips 설정 >

위와 같이 DataTips가 설정시킨 후 디버깅과 관련된 메모를 입력한다. 그리고 DataTips는 에디터 내에서 마음대로 이동이 가능하다.



< 그림 44. DataTips에 메모 추가 및 이동 모습 >

디버깅이 끝나면 DataTips는 모습을 숨긴다.



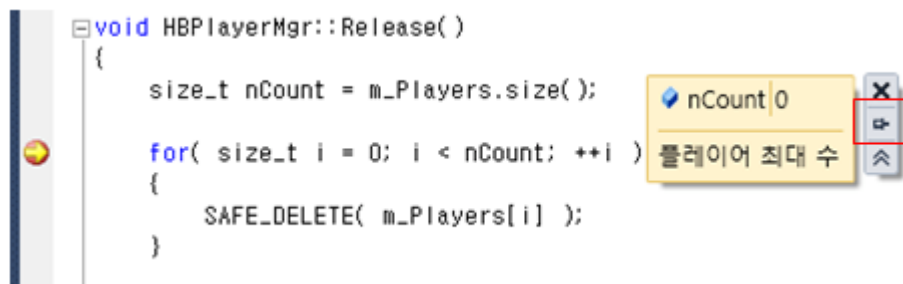
< 그림 45. 디버깅이 끝난 후의 모습 >

DataTips의 또 하나 편리한 점은 디버깅이 끝난 후 DataTips 마크에 마우스 커서를 가져가면 앞선 디버깅에서 어떤 값을 가졌는지 표시된다.



< 그림 46. 디버깅이 끝났지만 DataTips의 내용을 볼 수 있다 >

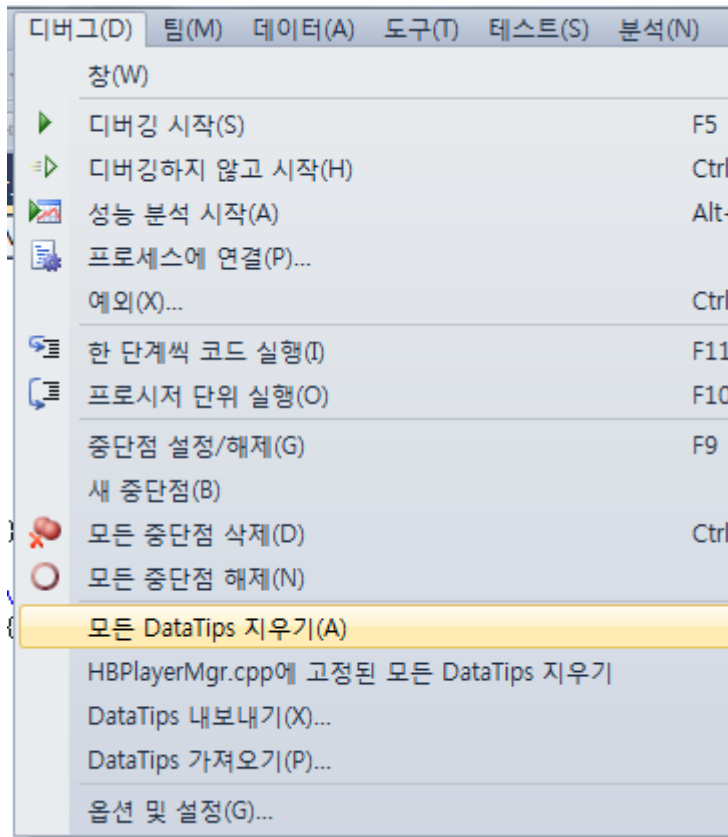
참고로 <그림 46>을 보면 디버깅 상태가 아닌 경우에는 DataTips의 마크가 표시되는데 만약 이 마크가 표시되지 않는다면 그것은 아래와 같은 상황일 테니 잘 확인해야 한다.



< 그림 47. 디버깅이 끝난 후 DataTips 마크가 표시 안됨 >

## DataTips 가져오기/내보내기

중단점처럼 DataTips도 xml 파일 형식으로 가져오기와 내보내기를 할 수 있다. 방법은 메뉴의 [디버깅]를 선택한 후 [DataTips 가져오기]나 [DataTips 내보내기]를 선택하면 된다.



< 그림 48. 메뉴의 DataTips 내보내기/가져오기 >

또 <그림 48>을 보면 알 수 있듯이 이외에도 모든 DataTips 지우기와 현재 선택된 파일의 모든 DataTips 지우기 기능도 있다.

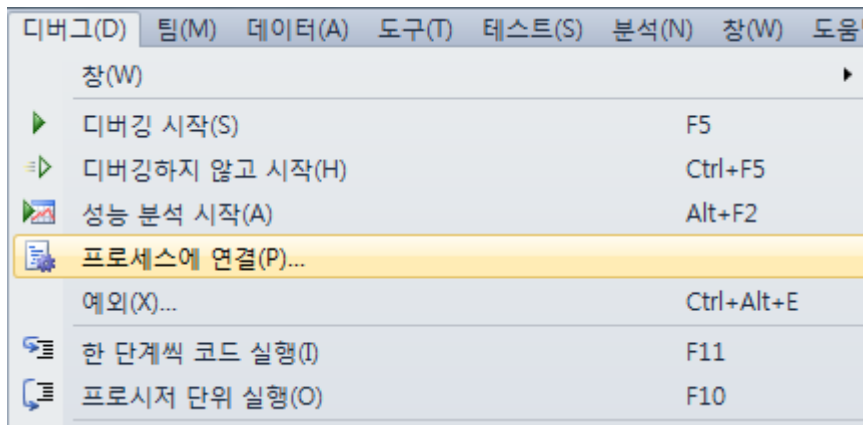
## 2.12 실행 중인 프로그램 디버깅

디버깅 실행이 아닌 일반적인 방법으로 실행을 했는데 실행 후 프로그램이 이상한 행동을 해서 디버깅을 해봐야 정확한 문제를 알 수 있을 때 어떻게 해야 할까? 프로그램을 종료하고 VC++에서 디버깅 실행을 한 후 디버깅을 해야 할까? 대부분의 골치 아픈 버그는 프로그램이 매번 실행할 때마다 발생하지 않고 불특정 하게 버그가 발생하는 것이다. 그래서 버그를 잡기 위해 프로그램을 종료하고 VC++로 디버깅 실행을 하면 버그가 나타나지 않아서 버그 잡는데 많은 시간이 소모될 수 있다.

이런 경우를 대비해서 VC++은 실행 중인 프로그램을 디버깅 할 수 있는 기능을 지원한다.

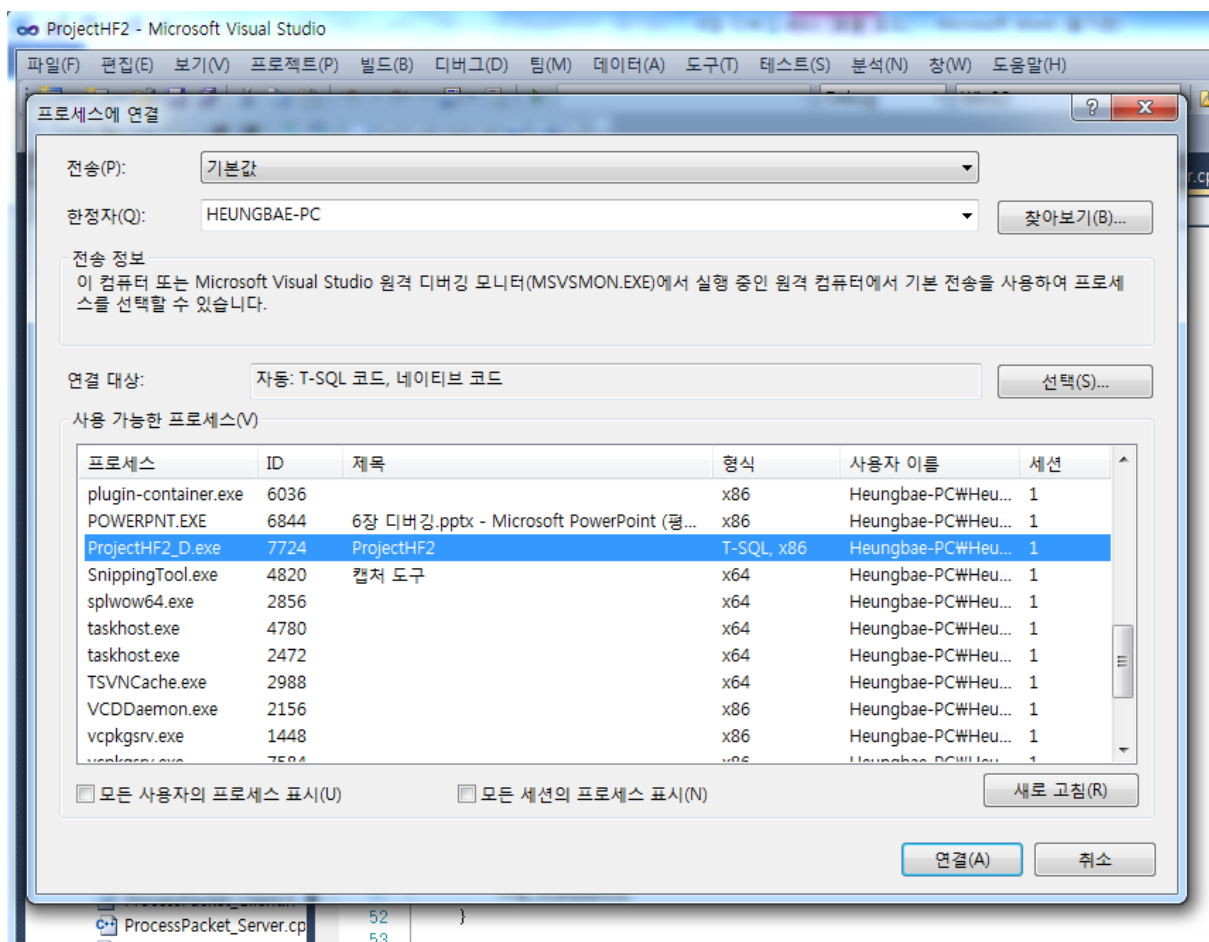
바로 디버그 기능 중 '프로세스에 연결' 이라는 것을 사용하면 된다.

이 기능은 메뉴의 [디버그] -> [프로세스에 연결]을 선택하면 된다.



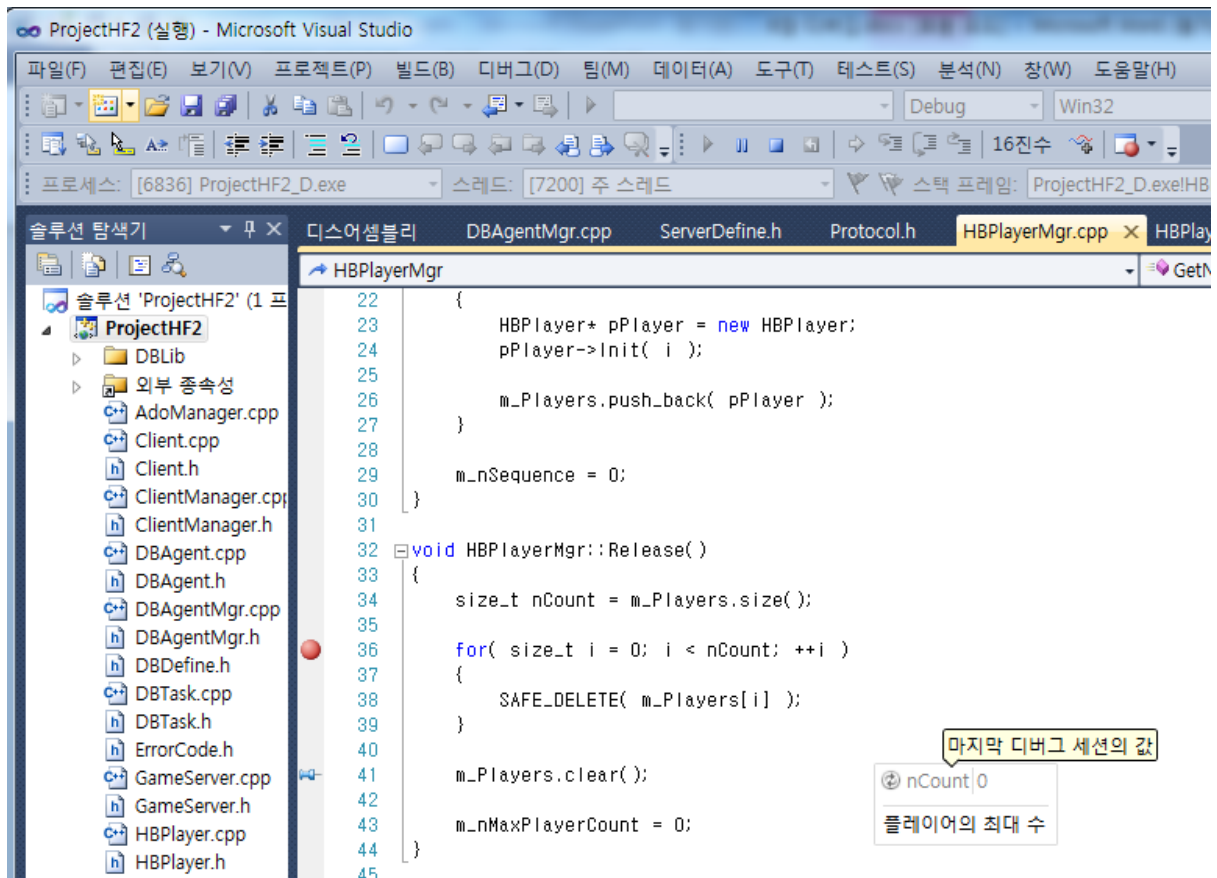
< 그림 49. 디버그 메뉴의 프로세스에 연결 >

프로세스에 연결 기능을 이용해서 디버깅을 하기 위해서는 먼저 디버깅할 프로그램의 프로젝트를 연 후 메뉴에서 [프로세스에 연결]을 선택한다.



< 그림 50. 프로세스에 연결 실행 >

<그림 50>은 'ProjectHF2\_D.exe'라는 프로그램을 실행한 후 이 프로그램의 VC++ 프로젝트를 VC++에서 연 후, '프로세스에 연결' 메뉴를 실행한 모습입니다. 여기서 아래의 '연결' 버튼을 누르면 이제 프로젝트와 실행중인 프로그램이 서로 연결되면서 아래와 같이 디버깅 실행이 된다.



< 그림 51. '프로세스에 연결'을 사용하여 디버깅 >

이 후는 일반적인 디버깅 실행처럼 디버깅을 하면 된다. 이 기능은 프로그램의 소스만 가지고 있다면 대단히 유용한 기능이니 꼭 잊지 말기를 바란다.

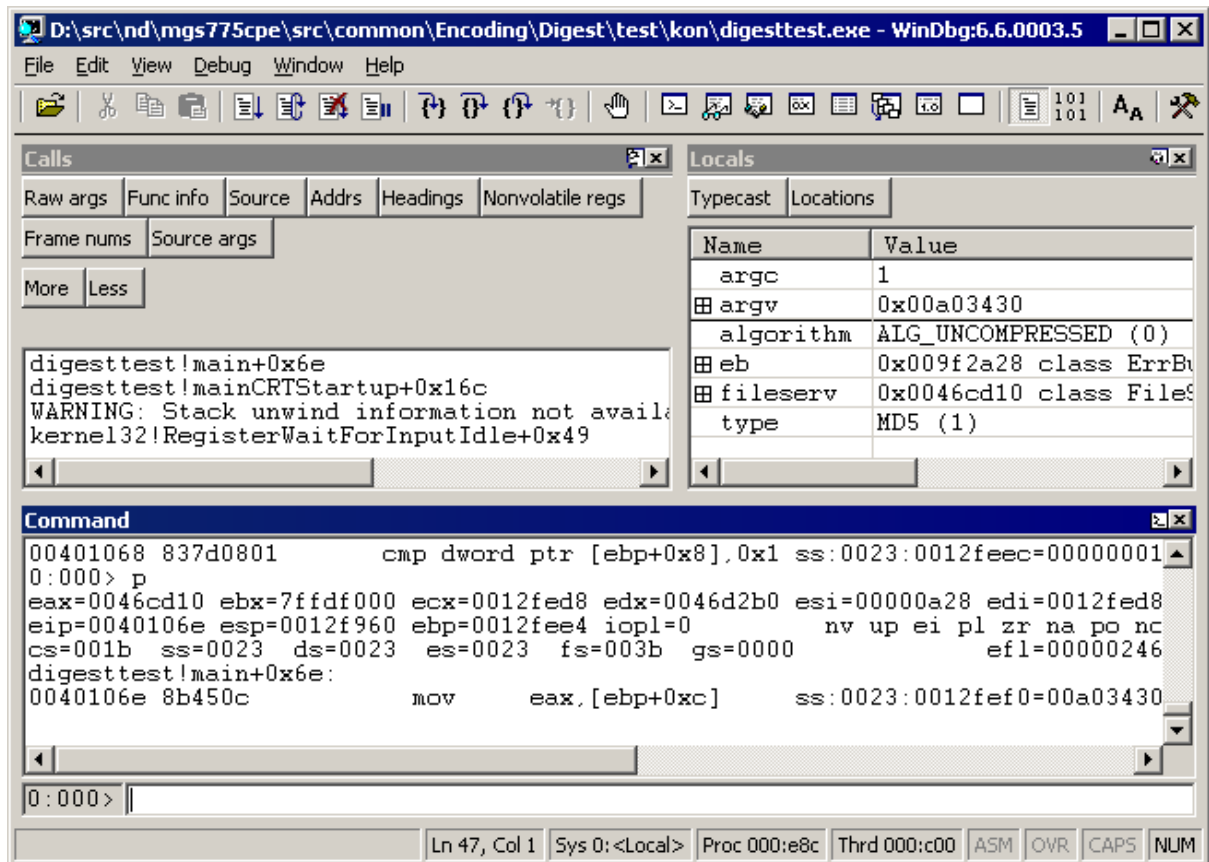
## 2.13 덤프 파일 디버깅

프로그램이 실행 중 갑자기 죽어버리는 경우는 문제를 어떻게 해결해야 할까? 특히 자신이 만든 프로그램이 자신의 눈 앞에서 죽어버리면 그나마 이유를 추측할 수 있지만 다른 곳에서 죽어버린다면 이유를 추측하기가 쉽지 않을 것이다.

프로그램에 어떠한 설정을 해서 갑자기 죽어버릴 때 '덤프 파일'이라는 것을 남길 수가 있다. 이 덤프 파일만 있으면 프로그램이 죽었을 때의 상황을 알 수 있다. 일종의 '블랙 박스'라고 생각해도 좋다. 덤프 파일의 확장자는 '.dmp' 이다.

덤프 파일을 입수하면 이제 이 파일로 디버깅을 해야 한다. 예전에는 덤프 파일 디버깅을 위해서는 'WinDbg'라는 사용하기 불편한 디버그 툴을 사용해야 했으나 VC++ 9 이후부터는 VC++에서

도 덤프 파일 디버깅을 할 수 있다.



< 그림 52. WinDbg. 보통 커널 개발 프로그래머가 많이 사용한다. VC++로도 디버깅이 힘든 경우에는 이 툴을 사용해야 하는 경우도 있다 >

<코드>

```

#include <stdio>
#include <string.h>
#include <wchar.h>
#include "client/windows/handler/exception_handler.h"

```

```

namespace {
    static bool callback(const wchar_t *dump_path, const wchar_t *id,
        void *context, EXCEPTION_POINTERS *exinfo,
        MDRawAssertionInfo *assertion,
        bool succeeded) {
        if (succeeded) {
            printf("dump guid is %ws\n", id);
        } else {
            printf("dump failed\n");
        }
    }
}

```



```

    }
    fflush(stdout);

    return succeeded;
}

static void CrashFunction()
{
    int *i = reinterpret_cast<int*>(0x45);
    *i = 5; // crash!
}

} // namespace




int main(int argc, char **argv)
{
    google_breakpad::ExceptionHandler eh(
        L".", NULL, callback, NULL,
        google_breakpad::ExceptionHandler::HANDLER_ALL);
    CrashFunction();
    printf("did not crash?\n");
    return 0;
}

```

</코드>

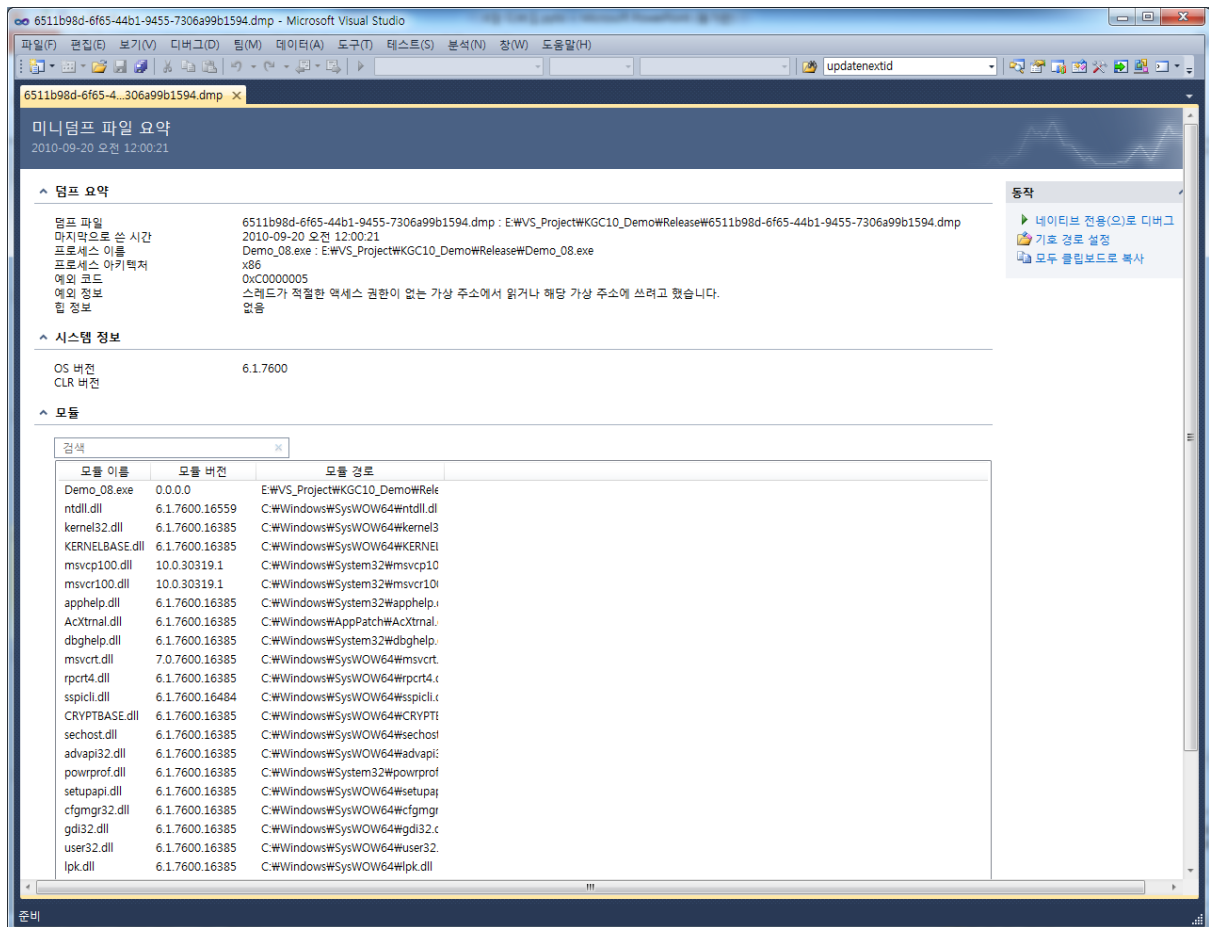
(이 코드의 출처는 <http://zhangyafeikimi.javaeye.com/blog/384351> 이다)

위 코드는 google-breakpad라는 라이브러리를 사용하여 프로그램이 비정상 종료를 할 때 덤프 파일을 남기도록 한다. 코드를 잘 보면 CrashFunction() 함수에서 무조건 프로그램이 죽도록 되어 있다. 이것을 빌드해서 실행하면 덤프 파일 발생한다.

급기 새 폴더		
이름	수정한 날짜	유형
 6511b98d-6f65-44b1-9455-7306a99b1594.dmp	2010-09-20 오전 12:00	Crash D
 Demo_08.exe	2010-09-19 오후 11:53	응용 프
 Demo_08.pdb	2010-09-19 오후 11:53	Program

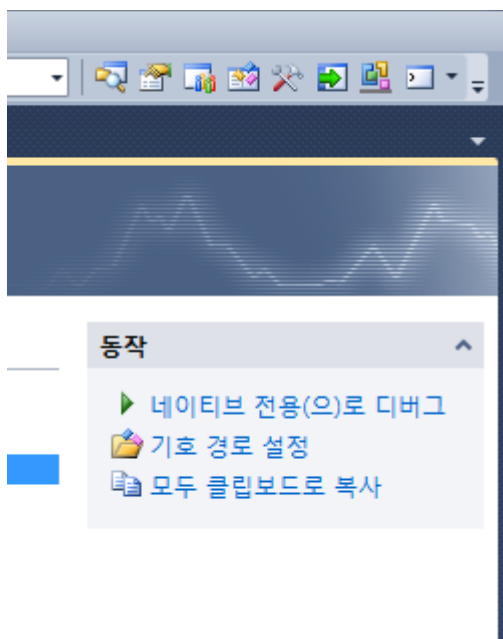
< 그림 53. 덤프 파일이 만들어졌다 >

덤프 파일을 실행하면 VC++이 자동으로 실행된다.



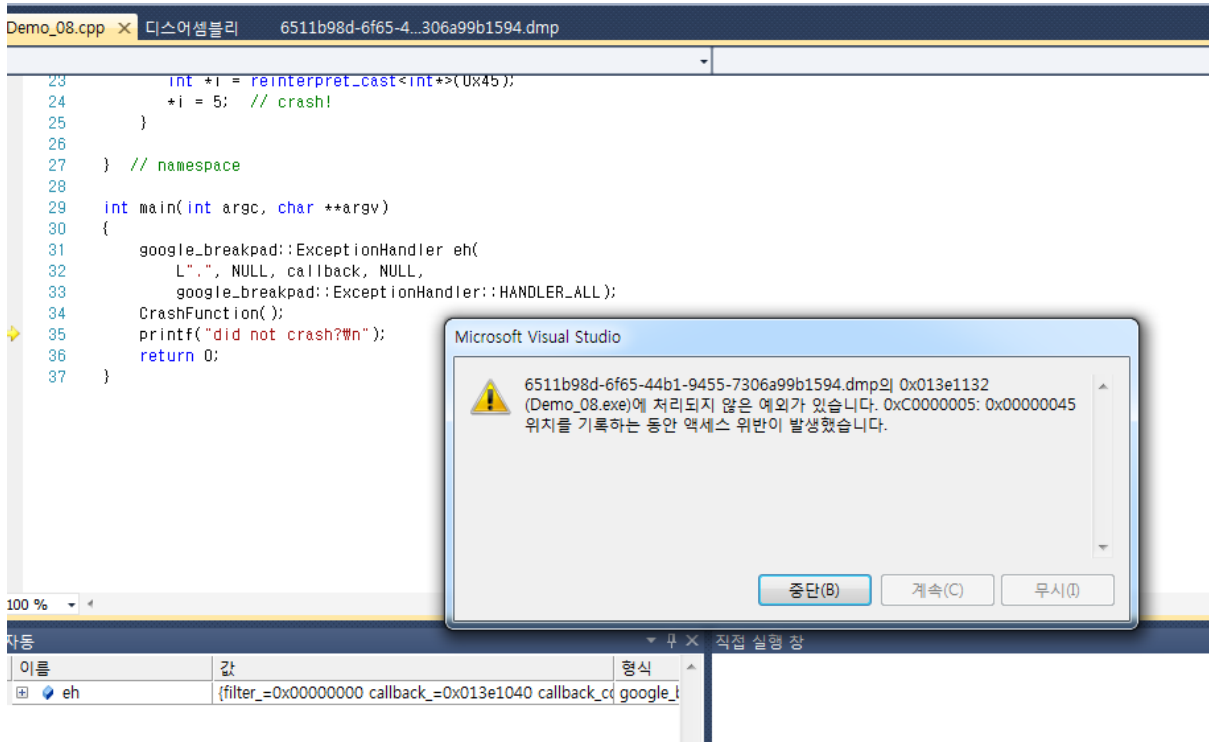
< 그림 54. 덤프 파일을 통해서 VC++이 실행된 모습 >

<그림 54>에서 오른쪽 상단에 있는 '네이티브 전용(으)로 디버그'를 마우스로 클릭한다



< 그림 55 >

그러면 아래와 같이 디버깅 모드가 되면서 프로그램이 죽은 위치를 가르킨다.



< 그림 56. 덤프 파일 디버깅 모습 >

덤프 파일을 디버깅 하기 위해서는 해당 프로그램의 소스 코드와 심볼 파일(.pdb 확장자를 가진 파일)이 필요하다.

VC++에서 덤프 파일 디버깅을 지원해 주기 전까지는 WinDbg라는 프로그램을 따로 설치해야 하고 사용 방법도 쉽지 않았는데(WinDbg에서 디버깅을 할 때는 커맨드 라인 명령어로 해야한다) 이제는 덤프 파일 디버깅도 무척 쉬워졌다.

이것으로 VC++에 있는 대부분의 디버깅 기능은 다 배웠다고 할 수 있다. 지금까지 설명한 것들 잘 기억한다면 어려운 디버깅을 좀 더 쉽고 빠르게 할 수 있을 것이다.