

Hanbit eBook

Realtime 52

7

당신의 프로그래밍에 디버깅 더하기

Visual C++ 디버깅 기초에서 고급까지

최홍배 지음

당신의 프로그래밍에 디버깅 더하기

Visual C++ 디버깅 기초에서 고급까지

당신의 프로그래밍에 디버깅 더하기 Visual C++ 디버깅 기초에서 고급까지

초판발행 2014년 11월 21일

지은이 최홍배 / 펴낸이 김태현

펴낸곳 한빛미디어(주) / 주소 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부

전화 02-325-5544 / 팩스 02-336-7124

등록 1999년 6월 24일 제10-1779호

ISBN 978-89-6848-658-6 15000 / 비매품

총괄 배용석 / 책임편집·기획 김창수 / 편집 김상민

디자인 표지 여동일, 내지 스튜디오 [밈], 조판 김상민

영업 김형진, 김진불, 조유미 / 마케팅 박상용, 김옥현

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.

한빛미디어 홈페이지 www.hanbit.co.kr / 이메일 ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea

Copyright © 2014 최홍배 & HANBIT Media, Inc.

이 책의 저작권은 최홍배와 한빛미디어(주)에 있습니다.

저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

지금 하지 않으면 할 수 없는 일이 있습니다.

책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.

한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 서문

프로그램에 버그가 발생했을 때 이것을 잡는 행위를 디버깅이라고 한다.

프로그래밍에서 디버깅은 필수다. 아주 간단한 프로그램이 아니고서야 단 한 번의 코딩만으로 버그 없는 프로그램을 만드는 것은 거의 불가능에 가깝기 때문이다. 프로그래머는 완벽할 수 없고, 우리가 만드는 프로그램은 복잡하기 때문에 언제나 버그가 발생한다. 그리고 이것을 잡기 위한 디버깅은 프로그래밍에서 절대 빼놓을 수 없다.

하지만 디버깅은 프로그램을 직접 만드는 과정이 아니고 디버깅해야 할 상황이 모든 사람에게 동일하지 않기 때문에 디버깅을 다루는 책은 시중에 많지 않다. 또한, 디버깅을 주제로 하여 가르치는 커리큘럼도 부족하여 많은 초보 프로그래머가 디버깅 기술이 부족한 실정이다.

보통 프로그래밍을 자주 하다 보면 버그가 수시로 발생하는데, 디버깅 기술은 이런 버그를 잡아가면서 향상시킬 수밖에 없다. 따라서 신입 프로그래머와 노련한 경력 프로그래머의 차이점 중 하나로 디버깅 기술과 경험을 꼽곤 한다.

Visual C++이 다른 C++ 툴보다 뛰어난 점 중 하나가 바로 디버깅 기능이 우수하다는 점이다. 필자는 유닉스 플랫폼에서 수년간 프로그래밍한 경험이 있는데 이때 가장 힘들었던 점이 디버깅하기가 너무 불편하다는 것이었다. 프로그램을 만들 때 보통 평균 30% 이상의 시간이 디버깅에 소모되는데 디버깅 기능이 빈약한 툴의 경우 훨씬 더 많은 시간을 소모함은 물론 스트레스도 많이 받게 된다.

프로그램을 좀 더 빠르게 완성하고 버그로 고통받지 않으려면 디버깅 기술과 경험을 쌓는 것이 매우 중요하다. 이 책에서는 Visual C++이 제공하는 가장 기초적인

디버깅 기능부터 고급 디버깅 기술까지 설명한다. Visual C++의 디버깅 기능은 사용하기 쉽고 어려운 개념을 이해할 필요도 없으며 필자가 하는 설명을 그저 쭉 따라가기만 하면 되니 가벼운 마음으로 책장을 넘기기 바란다.

저자 소개

저자_최홍배

2003년부터 현재에 이르기까지 PC 보드 게임부터 MMORPG, 모바일 게임을 아우르는 다양한 온라인 게임 서버 프로그램을 만들어온 개발자다. 게임 개발자로서 프로그래밍 언어 중 C++를 주 언어로, C#을 보조 언어로 사용하고 있다(그러나 최근에는 모바일 게임 서버 개발에 C#을 더 많이 사용하고 있다). 요즘은 C++11/14 프로그래밍과 심도 있는 .NET 기술, 유명 백엔드 오픈 소스 라이브러리 및 프로그램, 프로그래밍 언어 Ruby와 Scala에 대해 공부하고 있다. 기술과 개발 경험을 여러 사람과 나누는 것을 좋아하여 게임 개발자 커뮤니티나 세미나 강연을 통해 다른 프로그래머와 활발히 교류하고 있다. 웹이 대중화되기 전부터 프로그래밍 공부를 해 와서 그런지 여전히 새로운 기술을 배울 때는 책을 선호하여 지금도 매달 새로운 프로그래밍 관련 책을 읽으며 연구하고 있다. 현재 T3엔터테인먼트(<http://www.t3.co.kr>) 모바일 1팀에서 모바일 게임 서버 플랫폼을 개발 중이다.

- 블로그: <http://jacking.tistory.com/>
- 트위터: [@jacking75](https://twitter.com/@jacking75)

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. 무료로 업데이트되는, 전자책 전용 서비스입니다.

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위해 DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 편리한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 있을 가능성이 높습니다. 이 경우 저작권법에 의해 불이익을 받으실 수 있습니다.

다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횟수에 관계없이 다운받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오탈자 교정이나 내용의 수정·보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구입하신 독자에게만 제공되며, 다른 사람에게 양도나 이전은 허락되지 않습니다.

차례

1 디버깅 기초	1
1.1 기초 중의 기초 – 중단점 설정과 디버깅하기.....	1
1.2 중단점 이후 코드 디버깅.....	5
1.3 한 단계씩 코드 실행.....	7
1.4 프로시저 단위 실행.....	8
2 디버깅 고급	10
2.1 커서까지 실행.....	10
2.2 다음에 실행될 문 변경.....	11
2.3 포인터 배열의 내용 보기.....	14
2.4 중단점 조건.....	19
2.5 적중 횟수.....	23
2.6 중단점 창	25
2.7 중단점 비활성화하기.....	25
2.8 중단점 내보내기/가져오기.....	26
2.9 레이블.....	29
2.10 직접 실행.....	33
2.11 DataTips.....	35
2.12 실행 중인 프로그램 디버깅.....	40
2.13 덤프 파일 디버깅.....	42

1 | 디버깅 기초

1.1 기초 중의 기초 – 중단점 설정과 디버깅하기

1.1.1 중단점 설정

Visual C++에서 디버깅을 하려면 가장 먼저 디버깅하려는 위치에 중단점^{break point}을 설정해야 한다. 디버깅하려는 코드 부분에 중단점을 설정하고 디버깅을 시작하면 프로그램이 실행되다가 중단점을 설정한 부분에서 멈추게 되는데, 이때 현재 프로그램의 상태를 파악할 수 있다.

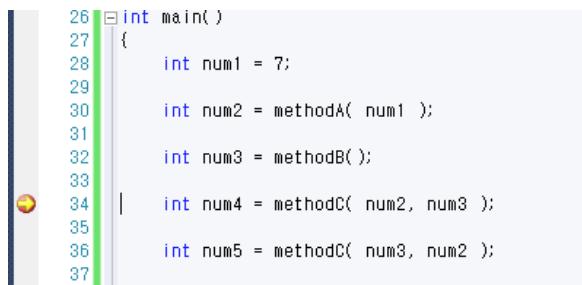
중단점은 프로그램이 실행되는 도중 멈추고자 하는 코드 부분에 단축키 F9를 눌러 설정한다.

[그림 1-1] 중단점을 설정하지 않은 코드

```
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 1-1]에서 변수 num2와 num3이 어떤 값으로 되어 있는지 알고 싶다면 코드의 34번째 줄에 커서를 놓아둔 후 단축키 F9를 누르면 [그림 1-2]와 같이 중단점이 설정된다. 중단점이 설정되면 그 위치에 빨간색 원이 표시된다.

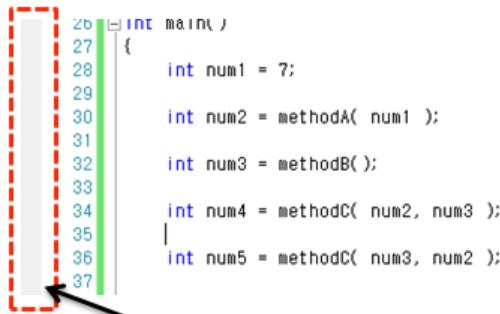
[그림 1-2] 중단점 설정



```
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 }
```

중단점은 단축키 F9 또는 마우스 클릭으로 설정할 수 있다.

[그림 1-3] 마우스 클릭으로 중단점 설정



```
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 }
```

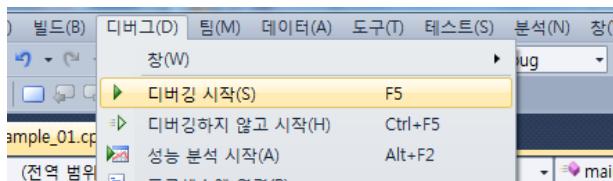
이 부분을 마우스로 클릭하면 중단점이 설정된다.

중단점을 없앨 때는 없애려는 중단점 위치에 커서를 옮긴 후 단축키 F9를 누르거나 중단점을 마우스로 클릭하면 된다.

1.1.2 디버깅하기

중단점을 설정한 후 디버깅을 할 때는 단축키 F5를 누르거나 메뉴의 [디버그] → [디버깅 시작]을 선택하면 된다.

[그림 1-4] 디버깅 시작



디버깅을 시작하면 [그림 1-5]와 같이 중단점을 설정한 위치에서 프로그램이 멈춘다.

[그림 1-5] 중단점 실행

```
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
40
41 }
```

The code editor shows a Java program with several lines of code. A red circular icon with a white dot, representing a breakpoint, is positioned next to the vertical margin line at line 34. The code uses standard Java syntax with methods like methodA, methodB, and methodC.

중단점에서 멈추면 디버깅 관련 창 중에서 자동 창을 이용해 중단점이 설정된 위치에서 사용되는 변수값을 확인할 수 있다.

[그림 1-6] 디버깅 시 자동 창

```
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
40
41 }
```

100 %

자동

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

The code editor shows the same Java program as in [그림 1-5]. Below it, the 'Autos' view of the debugger is open, displaying variable values. The variable 'num2' has a value of 107, 'num3' has a value of -200, and 'num4' has a value of -858993460. All variables are of type int.

[그림 1-6]을 보면 중단점이 설정된 위치에 변수 num2, num3, num4가 사용되고 있으므로 이들의 변수값이 표시된다. 변수 num2, num3은 각각 107, -200의 값을 갖고 변수 num4는 methodC() 함수가 아직 호출되지 않아 -858993460이라는 쓰레기 값을 갖는다.

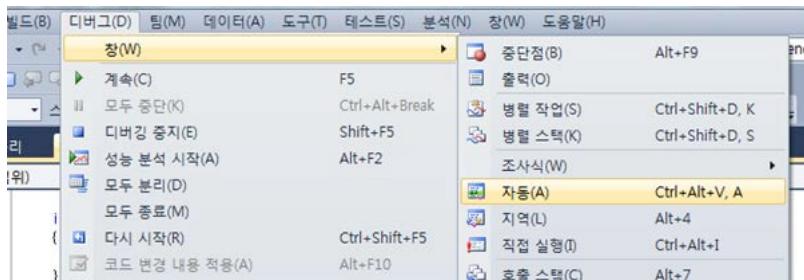
디버깅 시 변수값은 자동 창 외 지역 창에서도 출력된다. 지역 창은 디버깅 중 현재 멈춘 위치의 모든 변수를 표시해 준다.

[그림 1-7] 디버깅 시 지역 창

이름	값	형식
num4	-858993460	int
num1	7	int
num5	-858993460	int
num2	107	int
num3	-200	int

자동 창이나 지역 창이 표시되지 않을 때는 메뉴의 [디버그] → [창]을 선택하여 보고 싶은 창을 클릭하면 된다.

[그림 1-8] 메뉴에서 디버그와 관련된 창 표시하기



중단점에서 멈춘 프로그램을 계속 실행하려면 단축키 F5를 누르거나 다음과 같이 해당 아이콘을 마우스로 클릭한다.

[그림 1-9] 디버깅 시작 및 계속/종료 버튼



여기까지 가장 간단하게 디버깅하는 방법을 소개했다. 프로그램이 생각한 대로 동작하지 않을 때 앞에서 설명한 방법처럼 디버깅을 하면 프로그램이 어떻게 동작하고 있는지 자세히 알 수 있어 버그가 발생한 위치와 문제점을 쉽게 파악할 수 있다.

1.2 중단점 이후 코드 디버깅

[그림 1-10]처럼 디버깅할 때 설정한 중단점 이후 부분이 어떻게 동작하는지 세세하게 알고자 한 줄씩 프로그램을 실행하면서 변수의 내용을 살펴보고 싶다면 어떻게 해야 할까? 디버깅을 멈추고 중단점을 더 추가해야 할까? 그렇게 할 필요는 없다.

설정된 중단점 이후의 코드를 한 줄씩 실행하고 싶다면 중단점에 의해 멈춘 상황에서 F5 키로 실행하지 않고 ‘한 단계씩 코드 실행(단축키 F11)’이나 ‘프로시저 단위 실행(단축키 F10)’을 선택하여 중단점 이후의 코드를 한 줄씩 실행하면 된다.

[그림 1-10] 디버깅 예제 코드

```
#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;
}
```

1.3 한 단계씩 코드 실행

디버깅 시 중단점에서 중단된 이후 다음 코드를 한 줄씩 실행하는 ‘한 단계씩 코드 실행’의 단축키인 F11을 누르면 중단점 이후 함수를 호출하고 그 함수 내부로 들어가서 한 줄씩 실행하게 된다.

[그림 1-11] 한 단계씩 코드 실행의 흐름

The screenshot shows a debugger interface with two columns of code. The left column shows the main program code, and the right column shows the content of the methodC function being executed. A yellow dot marks a breakpoint at line 34. A yellow arrow points from the line 34 annotation to the start of the methodC function in the right column. Another yellow arrow points from the end of the methodC function back to the main program code. A callout box with a yellow arrow points to the line 34 annotation, containing the following text:

34째 줄에 설정한 중단점에서 멈춘 상태에서 'F11' 키를 누르면 36째 줄을 실행하지 않고 13째 줄의 methodC함수로 이동한다.

Below the code columns, there are two identical snippets of the methodC function:

```
13 int methodC( int a, int b )
14 {
15     int c = 0;
16
17     if( a < 0 )      여기에서 다시 'F11'을 누르면 15째 줄을 실행한다
18     {
19         a *= -1;
20     }
21
22     c = a * 1000;
23
24 }
```

```
13 int methodC( int a, int b )
14 {
15     int c = 0;
16
17     if( a < 0 )
18     {
19         a *= -1;
20     }
21
22     c = a * 1000;
23
24 }
```

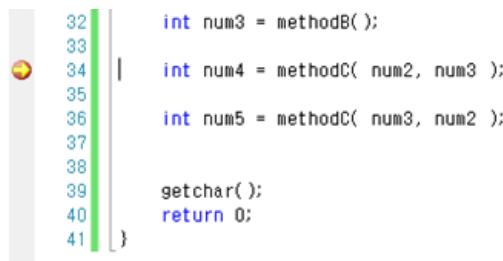
[그림 1-11]를 보면 34번째 줄의 중단점에 의해 프로그램이 멈추고 F11을 눌러 34번째 줄에서 호출하는 methodC 함수로 이동하여 한 줄씩 실행되는 것을 볼 수 있다. 만약 methodC 함수에서 나와 34번째 줄 다음의 코드를 실행하고 싶다면 단축키 Shift + F11을 누르면 된다.

이와 같이 ‘한 단계씩 코드 실행’은 눈에 보이는 코드를 한 줄씩 실행하는 것이 아니라, 내부적으로 실행하는 모든 코드를 한 줄씩 실행하게 된다. 그럼 [그림 1-10]의 34번쨰 줄에서 멈춘 후 다음을 실행할 때 [그림 1-11]처럼 methodC 함수로 이동하지 않고 보이는 대로 한 줄씩 실행하게 하려면 어떻게 해야 할까? 이때는 ‘프로시저 단위 실행’을 이용하면 된다.

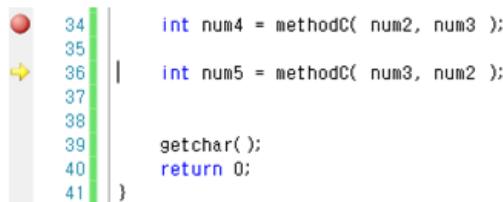
1.4 프로시저 단위 실행

중단점 이후의 코드를 실행할 때 보이는 줄 그대로 한 줄씩 실행하고 싶을 때는 F10을 눌러 ‘프로시저 단위 실행’을 선택한다.

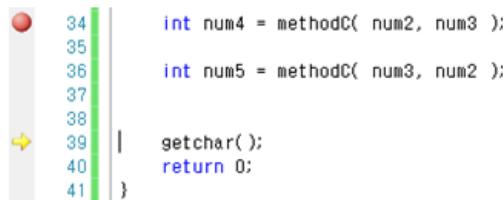
[그림 1-12] 프로시저 단위 실행



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 }



34 | int num4 = methodC(num2, num3);
35
36 | int num5 = methodC(num3, num2);
37
38
39 getchar();
40 return 0;
41 }



34 | int num4 = methodC(num2, num3);
35
36 | int num5 = methodC(num3, num2);
37
38
39 | getchar();
40 return 0;
41 }

[그림 1-12]를 보면 34번째 줄에서 멈춘 후 F10을 누르면 36번째 줄로 넘어가고 다시 F10을 누르면 39번째 줄로 넘어가는 것을 볼 수 있다.

이것으로 Visual C++에서의 가장 기본적인 디버깅 방법을 다 배웠다. 여기까지 알고 있다면 이제 디버깅을 할 수 있다고 자신 있게 말할 수 있다. 그러나 어디까지나 이것은 가장 기초적인 디버깅 방법으로 간단한 프로그램을 만들 때는 충분할 수 있지만, 복잡한 프로그램에서는 이것만으로는 효율적으로 버그를 잡을 수 없다. 복잡한 프로그램에서 버그를 좀 더 빨리 잡기 위해서는 Visual C++에서 제공하는 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 잠시라도 걱정했다면 그럴 필요는 없다. 기본과 구분하기 위하여 고급이라는 표현을 사용했지만 실제로는 짧고 간단하여 습득하기에 정말 쉽다. Visual C++의 디버깅 기능이 좋은 가장 큰 이유가 바로 디버깅을 쉽게 할 수 있다는 것이다. 그러나 대부분의 프로그래머가 기초적인 디버깅만 배우고 그것만을 사용하고 있어, Visual C++에서 제공하는 편리한 디버깅 기능을 제대로 활용하지 못하는 것 같아 많이 안타깝다.

다음에 설명하는 것을 잘 배운 후 디버깅을 할 때 적극적으로 활용하기 바란다.

2.1 커서까지 실행

소스 코드에서 중단점을 설정하지 않고 특정 위치까지 실행하다 멈추려 할 때가 있을 것이다. 코드를 분석하다가 특정 위치까지 프로그램이 어떻게 실행하는지 알고 싶은 경우가 이에 해당될 것이다.

사용 방법은 프로그램을 실행하다 멈추려는 위치에 커서를 이동한 후 Ctrl + F10 을 누르면 된다.

[그림 2-1] 커서까지 실행 전(왼쪽 그림)과 후(오른쪽 그림)

The screenshot shows two side-by-side code editors in Visual Studio. On the left, labeled '전' (before), the cursor is at line 34, which contains the statement 'int num4 = methodC(num2, num3);'. On the right, labeled '후' (after), the cursor has moved to line 34, and a yellow arrow points to the start of the line, indicating the current execution position. Both versions of the code are identical, showing five integer assignments followed by a getchar() call and a return 0; statement.

```
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
39     getchar();
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
39     getchar();
40
41 }
```

앞서 배운 기본 디버깅 방법으로는 중단점을 설정한 후 디버깅을 실행하고 그 부분을 다시 사용하지 않을 때는 중단점을 해제해야 한다. 그에 반해 ‘커서까지 실행’은 중단하려는 부분에 커서를 이동한 후 Ctrl + F10을 누르면 간단하게 디버깅할 수 있게 된다.

2.2 다음에 실행될 문 변경

디버깅 도중 현재 부분의 앞부분을 한 줄씩 실행하려고 할 때는 디버깅을 종료하고 중단점을 추가해야 할까? 경우에 따라서는 디버깅을 끝내지 않고도 멈춘 부분의 앞부분을 한 줄씩 실행할 수 있다. ‘다음에 실행될 문 변경’ 기능을 사용하면 된다. 방법은 [그림 2-2]와 같다.

[그림 2-2] 다음에 실행될 문 변경

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

중단점에 마우스 커서를 이동시키면 중단점에 화살표와 위와 같은 메시지가 나온다. 이때 원하는 지점까지 마우스로 드래그 한다.

```
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.

단, 이 기능을 사용할 때 주의해야 할 점이 있다. 이미 실행된 부분의 결과는 사라지지 않고 그대로 있다는 점이다.

[그림 2-3] 14번째 줄에서 멈춘 상태

The screenshot shows a debugger interface with assembly code and a local variable table. The assembly code is as follows:

```
3 int CallIA( int nNum )
4 {
5     int nRetNum = nNum + 100;
6     return nRetNum;
7 }
8
9 int main()
10 {
11     int nNum1 = 100;
12     nNum1 = CallIA( 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 }
22 }
```

The local variable table (지역) shows the following values:

이름	값
nNum1	200
Nums	0x0054fc40

[그림 2-4] 다음 실행될 위치를 12번째 줄로 이동

The screenshot shows a debugger interface. The code editor displays the following C code:

```
9 int main()
10 {
11     int nNum1 = 100;
12     nNum1 = CallIA( 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 }
22 }
```

The cursor is at line 12, indicated by a yellow arrow icon in the margin. A red dot icon is also present in the margin. The local variables window below shows:

이름	값
nNum1	200
Nums	0x002ef990

[그림 2-5] 12번째 줄 실행

The screenshot shows the debugger after executing the statement at line 12. The code editor now shows:

```
9 int main()
10 {
11     int nNum1 = 100;
12     nNum1 = CallIA( 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 }
22 }
```

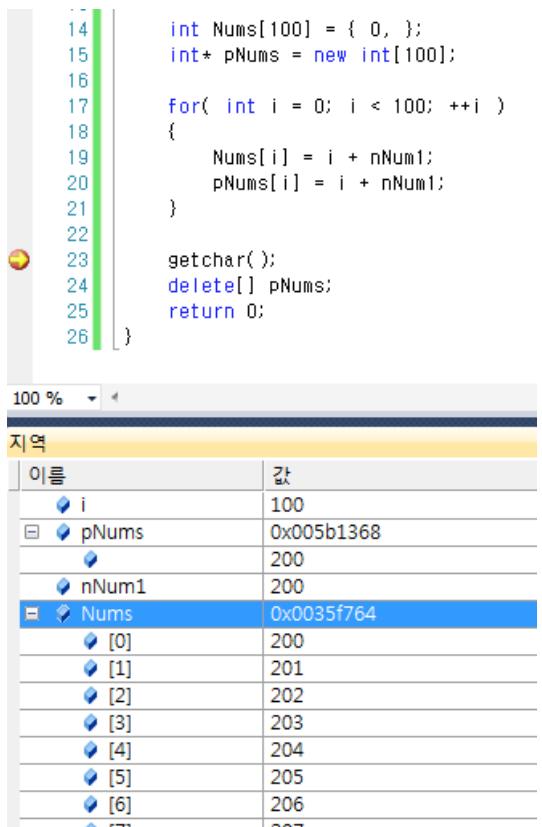
A red dot icon is now in the margin at line 12, indicating it is the current instruction. The local variables window shows:

이름	값
nNum1	300
Nums	0x002ef990

[그림 2-3]를 보면 알 수 있듯이 코드에서 변수 nNum1의 값은 100이다. 따라서 처음 실행될 때 CallA 함수에 전달되는 변수 nNum1의 값은 100이므로 12번째 줄이 실행되면 변수 nNum1의 값은 200이 된다. 그러나 [그림 2-4]처럼 실행 위치를 12 번째 줄로 이동하면 CallA 함수에 200이라는 값을 가진 변수 nNum1을 전달하여 12번째 줄을 다시 실행하면 변수 nNum1의 값은 300이 된다([그림 2-5]). 이처럼 한번 실행된 결과는 사라지지 않고 그대로 적용됨을 알 수 있다.

2.3 포인터 배열의 내용 보기

[그림 2-6] 지역 창 변수 Nums와 pNums의 값



The screenshot shows a debugger interface with two main sections: a code editor and a variable viewer.

Code Editor:

```
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 }
```

Variable Viewer (Watch View):

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

The variable _nums is highlighted in blue, indicating it is selected. The memory dump shows the first few elements of the Nums array starting at address 0x0035f764, with values increasing from 200 to 206.

[그림 2-6]을 보면 지역 창에 변수 Numbs와 pNumbs의 값을 확인할 수 있다. 그런데 변수 Numbs의 경우 배열 각 요소의 값을 보여주지만, 포인터인 변수 pNumbs의 경우 첫 번째 요소의 값만 보여준다. 변수 pNumbs의 각 요소의 값을 보고 싶다면 조사식 창을 사용하면 된다. 조사식 창이 보이지 않는 경우에는 메뉴의 [디버그] → [창] → [조사식]을 선택한다.

[그림 2-7] 조사식 창 띄우기

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

조사식 1	이름	값

조사식 2	이름	값

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

마우스로 pNums를 선택 후 드래그 한다.

변수 pNums의 특정 위치 값을 알고 싶을 때는 조사식에 ‘pNums[위치]’를 입력한다.

[그림 2-8] pNums의 6번째 값 보기

The screenshot shows a code editor and a debugger interface. The code editor displays a C++ program with a breakpoint at line 23. The debugger window has a zoom level of 100% and shows a memory dump titled '조사식 1' (Inspection 1). A table lists memory locations and their values, with the entry for pNums[5] highlighted.

```
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
26 }
```

이름	값
pNums[5]	205

변수 pNums의 여러 행 값을 알고 싶을 때는 ‘pNums, 알고 싶은 행 개수’를 조사식에 입력한다.

[그림 2-9] pNums의 모든 행 값 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
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

앞의 경우는 변수 pNums의 여러 행 값을 볼 수 있어 유용하지만 너무 많은 값을 보여줘 불편하기도 하다. 만약 변수 pNums의 11번째 위치 이후부터 10개의 값만을 보고 싶다면 ‘pNums + 위치, 보고 싶은 행 개수’를 조사식에 입력한다.

[그림 2-10] pNums의 11번째 위치부터 10개의 값만 보기

The screenshot shows a debugger interface. At the top, there is assembly code:

```
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
26 }
```

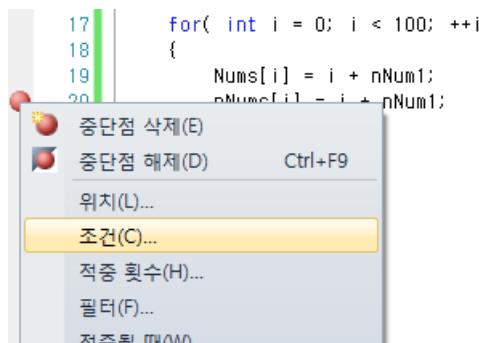
Below the code is a zoomed-in view of the memory dump titled "조사식 1". It shows a table with two columns: "이름" (Name) and "값" (Value). The table has 10 rows, indexed from [0] to [9]. The first row is expanded to show its contents:

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

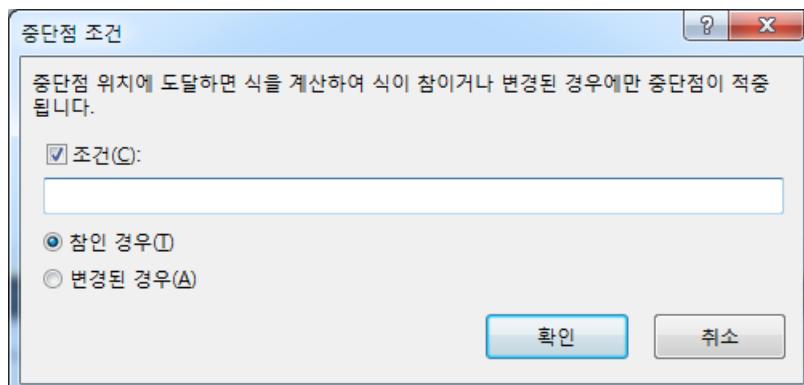
2.4 중단점 조건

디버깅할 때 중단점이 설정된 곳에 매번 멈추지 않고 변수값이 변경되거나 변수값이 특정값이 될 때만 멈추기를 원한다면 중단점에 ‘조건’을 설정하면 된다. 조건은 특정 변수값이 변하거나 특정값이 되는 두 가지 경우에 설정할 수 있다. 조건을 설정하는 방법은 중단점을 설정한 후 중단점에서 마우스 우클릭하여 나오는 팝업 메뉴 중 ‘조건’을 선택하면 조건을 설정하는 창이 나온다.

[그림 2-11] 중단점에 조건 설정을 위해 팝업 메뉴에서 선택



[그림 2-12] 중단점 조건



중단점 조건에서 ‘참인 경우’와 ‘변경된 경우’를 선택할 수 있는데 두 경우를 예로 설명하겠다.

2.4.1 중단점 조건 - 참인 경우

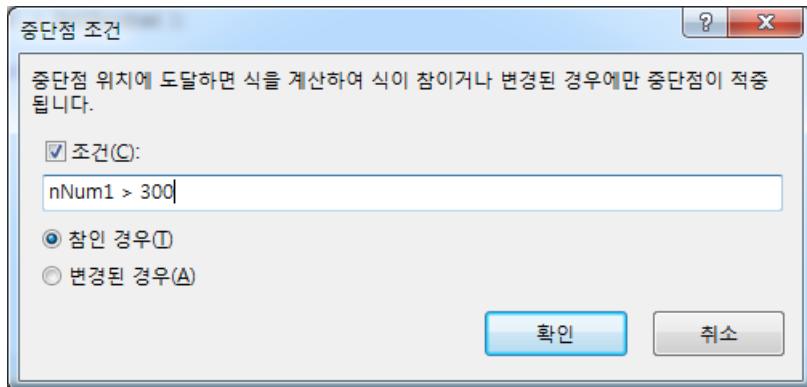
[그림 2-13] 중단점 조건 - 참인 경우 예제 코드

The screenshot shows a code editor window titled "Sample_03.cpp". The code is written in C++ and defines a function "GetRandom" that returns a random integer. It also contains a main function that calls "GetRandom" and returns 0. A red dot, representing a breakpoint, is placed on the line "return 0;" (line 15). The code is color-coded with syntax highlighting: comments are in green, strings in yellow, and keywords in blue. Braces and operators are in black. The line numbers are on the left side of the code area.

```
1 #include <iostream>
2 #include <math.h>
3
4 int GetRandom( )
5 {
6     return rand( );
7 }
8
9 int main( )
10 {
11     int nNum1 = 0;
12
13     nNum1 = GetRandom( );
14
15     return 0;
16 }
17
```

[그림 2-13]을 보면 15번째 줄이 중단점으로 설정되어 있다. 디버깅할 때 13번째 줄이 실행되어 변수 nNum1의 값이 300이 될 경우에만 멈추고 싶다면 다음과 같이 조건을 설정한다.

[그림 2-14] 중단점 조건 - 참인 경우 설정



이제 디버깅을 실행하면 변수 nNum1이 300보다 큰 경우에만 중단점 설정에 의해 멈추고 300 이하면 멈추지 않게 된다. 중단점 조건이 설정되면 중단점은 [그림 2-15]처럼 바뀌게 된다.

[그림 2-15] 조건이 설정된 중단점



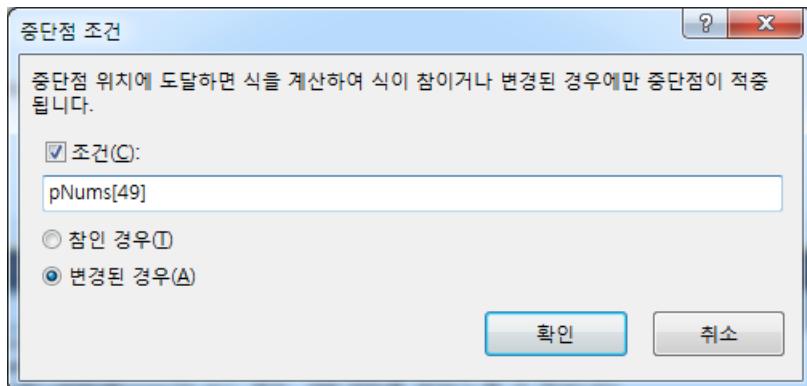
2.4.2 중단점 조건 - 변경된 경우

[그림 2-16] 중단점 조건 - 변경된 경우 예제 코드

```
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 2-16]을 보면 20번째 줄이 중단점으로 설정되어 있다. 여기서는 변수 pNums의 50번째 위치 값이 변경된 경우 중단되도록 조건을 설정했다.

[그림 2-17] pNums[49]의 값이 변경된 경우 중단되도록 설정



[그림 2-18] [그림 2-16]의 코드를 디버깅

```
13     int Nums[100] = { 0, };
14     int* pNums = new int[100];
15
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;
```

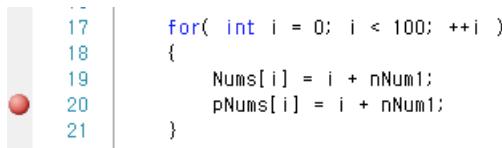
자동	
이름	값
_nums	0x0027f864
_Nums[i]	250
i	50
nNum1	200
pNums	0x005313b0
pNums[i]	-842150451

[그림 2-18]을 보면 20번째 줄에서 pNums[50]에 값을 대입할 때 pNums[49]의 값이 변경된 것을 감지하고 중단되었다.

이렇게 중단점 조건을 사용하면 특수한 상황을 디버깅할 때 편리하다. 만약 중단점 조건을 사용하지 않으면 특수한 상황이 될 때까지 ‘디버깅 시작 → 디버깅 중단’을 반복한다면 디버깅에 매우 많은 시간을 소비하게 될 것이다.

2.5 적중 횟수

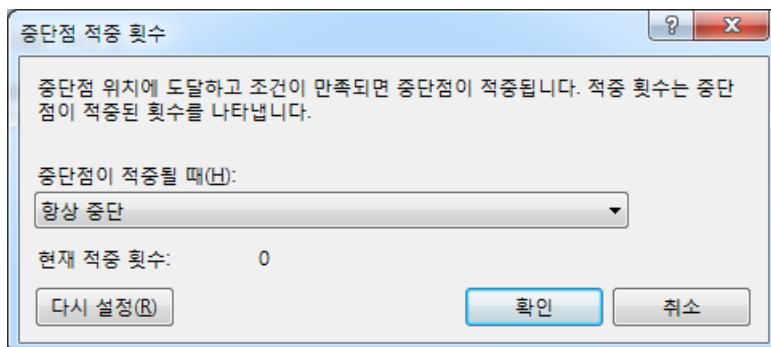
[그림 2-19] 반복문에서 디버깅하기



```
17 |     for( int i = 0; i < 100; ++i )
18 |     {
19 |         Nums[i] = i + nNum1;
20 |         pNums[i] = i + nNum1;
21 |     }
```

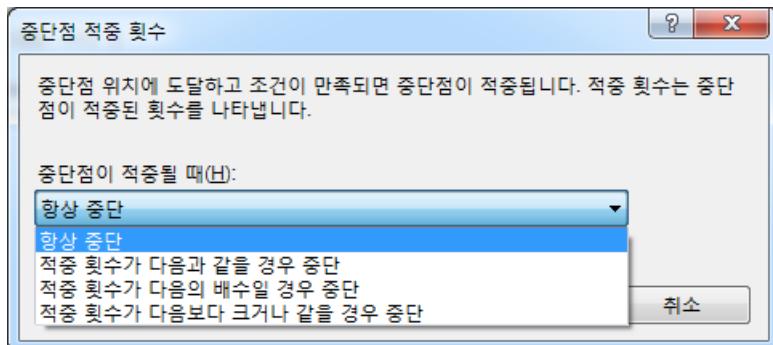
[그림 2-19]과 같이 반복문 안에 중단점을 설정하여 디버깅할 때 매번 중단되지 않고 특정 횟수일 때 중단되기를 원한다면 ‘적중 횟수’를 사용한다. 적중 횟수는 [그림 2-1]처럼 중단점의 팝업 메뉴에서 선택할 수 있다. 적중 횟수 메뉴를 선택하면 [그림 2-20]과 같은 창이 나타난다.

[그림 2-20] 중단점 적중 횟수 창



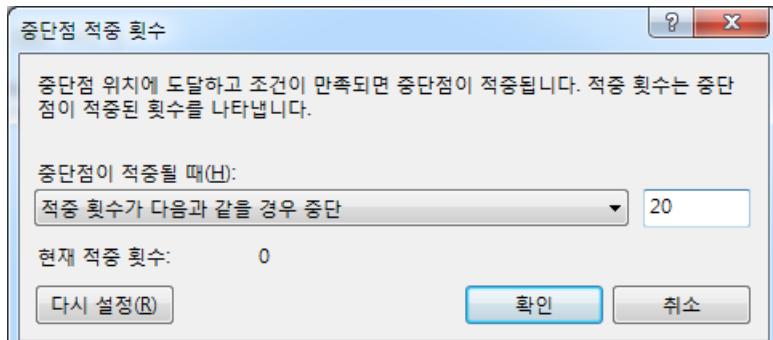
기본 설정은 항상 중단이지만 [그림 2-21]처럼 옵션을 변경할 수 있다.

[그림 2-21] 중단점 적중 횟수 옵션



만약 '적중 횟수가 다음과 같을 경우 중단'을 선택하면 [그림 2-22]와 같이 오른쪽에 텍스트 박스가 표시되고 여기에 횟수를 입력하면 디버깅할 때 설정한 적중 횟수 일 경우에만 중단되게 된다.

[그림 2-22] 중단점 횟수 설정



적중 횟수는 반복문 안에서 디버깅할 때 아주 유용한 기능이다.

2.6 중단점 창

현업에서 만드는 프로그램은 대부분 규모가 크고 복잡하다. 그래서 디버깅할 때 코드의 많은 부분에 중단점을 설정하게 된다. 중단점이 많아지면 중단점을 관리하는 것도 중요해진다. 중단점을 어디에 설정했는지, 중단점에 조건은 있는지 등을 고려해야 한다.

설정된 중단점의 정보는 중단점 창을 통해 확인할 수 있다. 중단점 창이 보이지 않을 때는 메뉴의 [디버그] → [창] → [중단점]을 선택하면 된다.

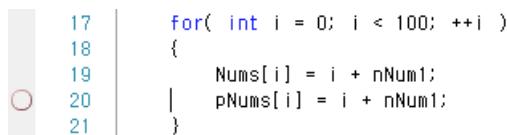
[그림 2-23] 중단점 창



2.7 중단점 비활성화하기

중단점을 삭제하지 않고 일시적으로 중단점을 비활성화하고 싶을 때가 있다. 방법은 해당 중단점 위치에 커서를 이동한 후 단축키 Ctrl+ F9를 누르면 된다. 중단점이 비활성화되면 중단점은 색이 없는 원으로 표시된다.

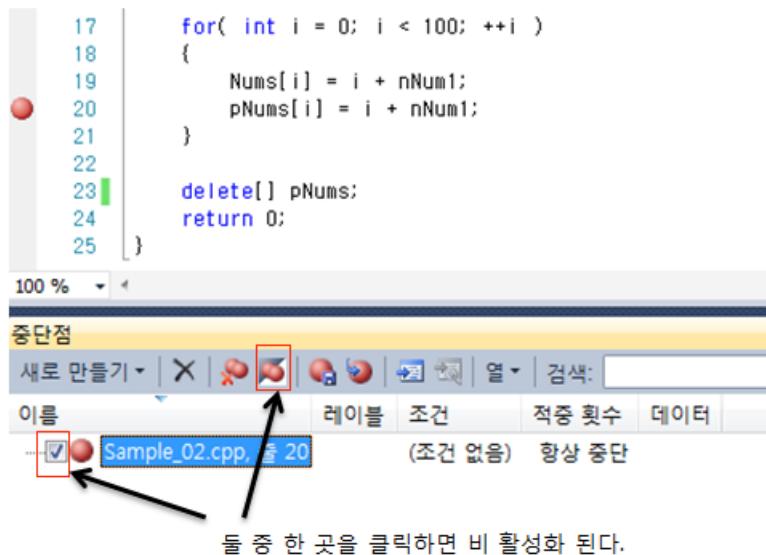
[그림 2-24] 비활성화된 중단점



다시 활성화하고 싶다면 Ctrl + F9를 한 번 더 누르면 된다. 중단점 비활성화는 중

단점 창에서도 할 수 있다.

[그림 2-25] 중단점 창에서 중단점 비 활성화하기

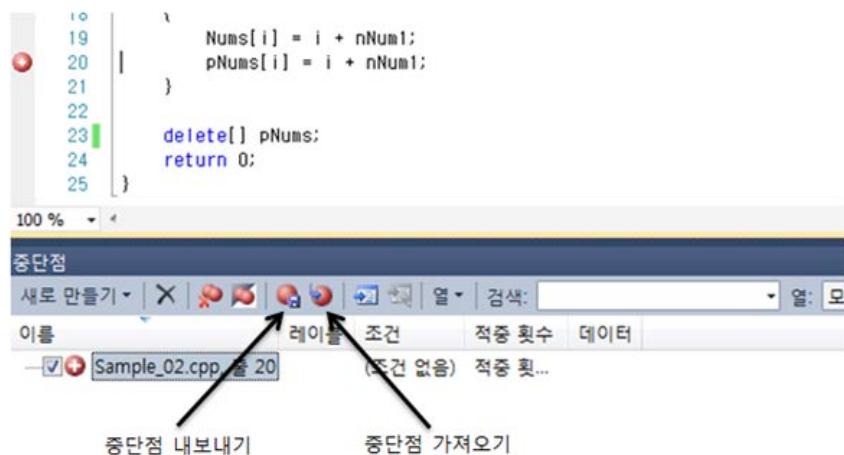


2.8 중단점 내보내기/가져오기

당장 중단점을 사용하지는 않지만 이후 중단점을 사용할 확률이 높을 경우나 같이 작업하는 동료 컴퓨터의 Visual C++에 자신이 디버깅에 사용했던 중단점을 설정해야 할 경우에는 중단점 내보내기/가져오기 기능을 사용한다.

중단점 정보는 xml 파일 포맷으로 내보내거나 가져올 수 있다. 중단점 내보내기/가져오기는 중단점 창에서 쉽게 실행할 수 있다.

[그림 2-26] 중단점 창에서 중단점 내보내기/가져오기



중단점 창에서 중단점 내보내기를 실행할 때는 중단점 창에 표시된 중단점만을 내보낸다(뒤에 설명할 레이블을 사용하면 특정 중단점만 중단점 창에 표시할 수 있다).

다음 그림은 내보낸 중단점 xml 파일 내용의 일부다.

[그림 2-27] 내보낸 중단점의 xml 파일 정보

```
<?xml version="1.0" encoding="utf-8"?>
<BreakpointCollection xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
  <Breakpoints>
    <Breakpoint>
      <Version>15</Version>
      <iEnabled>1</iEnabled>
      <iVisible>1</iVisible>
      <iEmulated>0</iEmulated>
      <iCondition>0</iCondition>
      <ConditionType>WhenTrue</ConditionType>
      <LocationType>SourceLocation</LocationType>
      <TextPosition>
        <Version>4</Version>
        <FileName>..\Sample_02\Sample_02.cpp</FileName>
        <startLine>19</startLine>
        <StartColumn>0</StartColumn>
        <EndLine>19</EndLine>
        <EndColumn>23</EndColumn>
        <MarkerId>41</MarkerId>
        <iLineBased>1</iLineBased>
        <iIsDocumentPathNotFound>0</iIsDocumentPathNotFound>
        <ShouldUpdateTextSpan>1</ShouldUpdateTextSpan>
        <Checksum>
          <Version>1</Version>
          <Algorithm>00000000-0000-0000-0000-000000000000</Algorithm>
          <ByteCount>0</ByteCount>
          <Bytes />
        </Checksum>
      </TextPosition>
      <NamedLocationText>main()</NamedLocationText>
      <NamedLocationLine>10</NamedLocationLine>
      <NamedLocationColumn>0</NamedLocationColumn>
      <HitCountType>IsEqualTo</HitCountType>
      <HitCountTarget>20</HitCountTarget>
      <Language>3a12d0b7-c26c-11d0-b442-00a0244a1dd2</Language>
    </Breakpoint>
  </Breakpoints>
</BreakpointCollection>
```

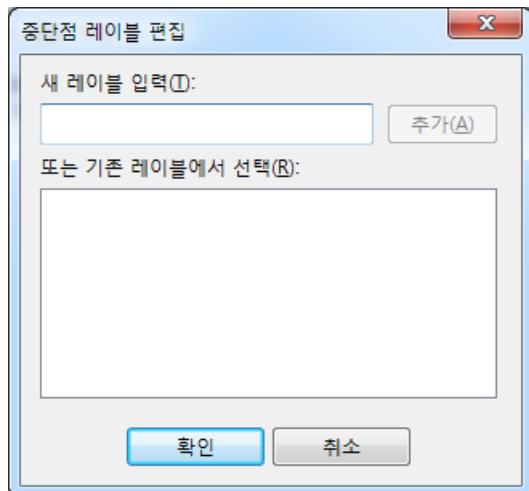
2.9 레이블

중단점에는 ‘레이블(이름)’을 지정할 수 있다. 레이블을 지정하면 어떤 목적을 가진 중단점인지 알 수 있고 비슷한 목적의 중단점을 그룹별로 관리할 수도 있다.

2.9.1 레이블 설정

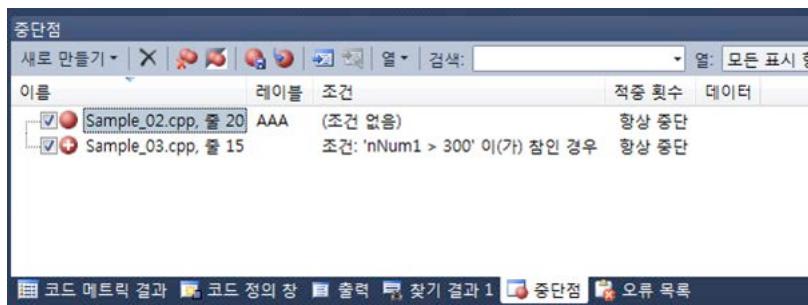
중단점을 마우스 우클릭하면 나타나는 팝업 메뉴에서 ‘레이블 편집’을 선택하면 ‘중단점 레이블 편집’ 창이 나타난다. 여기서 새로운 레이블 이름을 입력한다.

[그림 2-28] 중단점 레이블 편집 창



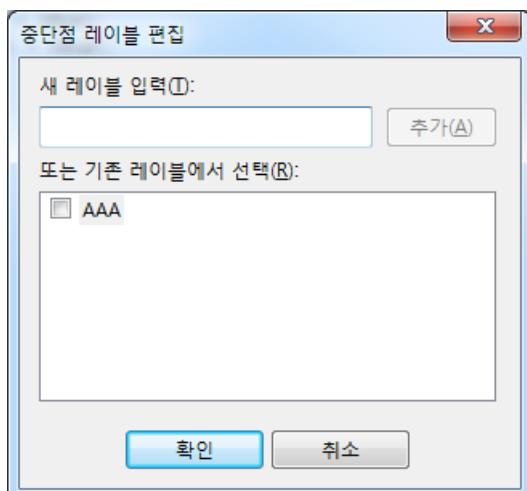
레이블을 입력하고 ‘확인’ 버튼을 누르면 중단점 창에 레이블이 표시된다.

[그림 2-29] 레이블이 표시된 중단점 창



만들어 놓은 레이블은 재사용할 수 있다. 다른 중단점에 같은 레이블을 사용하려고 때 이전에 만들어 놓은 레이블을 선택하면 된다.

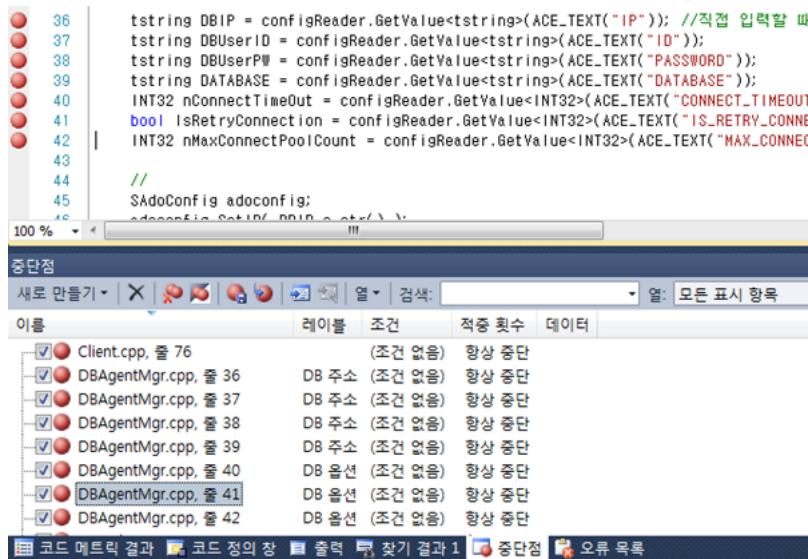
[그림 2-30] 앞에서 만들어 놓은 ‘AAA’ 레이블을 선택



2.9.2 그룹별 관리

서로 비슷한 목적을 가진 레이블이 있을 경우에는 같은 레이블로 지정하여 그룹별로 관리할 수 있다.

[그림 2-31] 같은 목적의 중단점들은 같은 레이블을 설정



[그림 2-31]을 보면 36 ~ 39번째 줄까지의 중단점은 DB 주소 디버깅을 위한 것이고, 40 ~ 42번째 줄까지의 중단점은 DB 옵션 디버깅을 위한 것임을 쉽게 알 수 있다. 레이블을 이용해 중단점을 그룹화하면 중단점 창에서 특정 중단점만 표시하여 중단점 정보를 쉽게 파악할 수 있다. 특정 레이블의 중단점만 표시하고 싶을 때는 검색 창을 이용해 원하는 레이블을 입력한다.

[그림 2-32] DB 옵션 레이블만 표시

The screenshot shows a debugger interface with the following details:

- Code View:** Shows C++ code for a configuration reader. Lines 36-45 are visible, including the declaration of an `SAdoConfig adoconfig;` object.
- Search Results:** A search bar at the top right contains the text "DB 옵션". Below it is a table titled "중단점" (Breakpoints) with three entries:

이름	레이블	조건	적중 횟수	데이터
DAgentMgr.cpp, 줄 40	DB 옵션 (조건 없음)	항상 중단		
DAgentMgr.cpp, 줄 41	DB 옵션 (조건 없음)	항상 중단		
DAgentMgr.cpp, 줄 42	DB 옵션 (조건 없음)	항상 중단		

- Toolbar:** Includes icons for code metrics, code search, output, find results, breakpoints, and other debugger functions.

2.10 직접 실행

디버깅 중 직접 코딩하지 않고 어떤 결과가 나오는지 테스트해야 할 때가 있다. 이럴 때는 ‘직접 실행’ 기능을 사용하면 된다. 직접 실행 창이 보이지 않으면 메뉴의 [디버그] → [창] → [직접 실행]을 선택하면 된다.

[그림 2-33] 디버깅 중 직접 실행 창

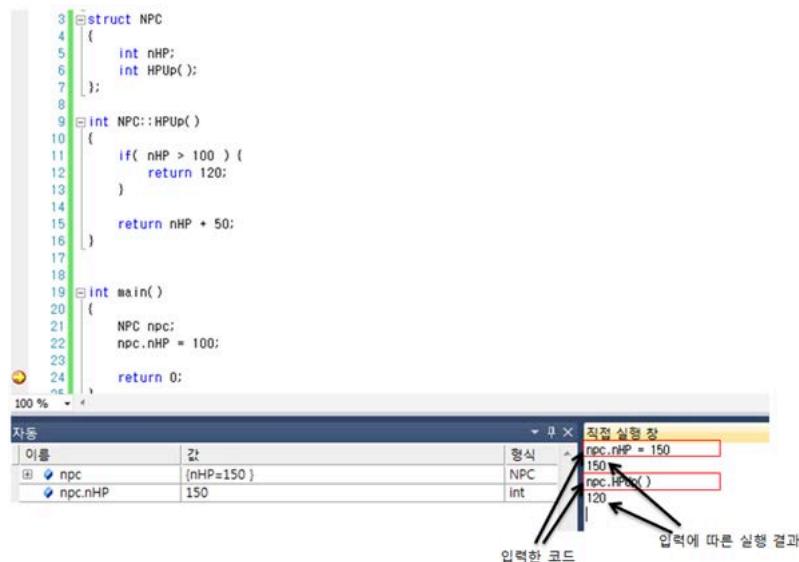
```
3 struct NPC
4 {
5     int nHP;
6     int HPUp();
7 }
8
9 int NPC::HPUp()
10 {
11     if( nHP > 100 ) {
12         return 120;
13     }
14
15     return nHP + 50;
16 }
17
18
19 int main()
20 {
21     NPC npc;
22     npc.nHP = 100;
23
24     return 0;
}
100 %
```

여기에서 실행하고 싶은 코드를 입력한다

자동	이름	값	형식
npc	{nHP=100}	NPC	
npc.nHP	100	int	

디버깅 중 임의의 코드에서 어떤 결과가 나오는지 알고 싶을 때는 [그림 2-34]처럼 ‘직접 실행 창’에 코드를 입력한다.

[그림 2-34] 직접 실행 창에 코드 입력



디버깅 중 직접 실행 창에 npc 객체의 nHP 멤버에 실제 코드와는 다른 값을 설정한 후 npc 객체의 HPUp() 멤버를 실행하여 어떤 결과가 나오는지 확인한 것이다.

참고로 [그림 2-34]를 보면 직접 실행 창에서 현재 사용 중인 변수값을 변경하면 바로 적용된다는 것을 알 수 있다(원래 코드에서는 npc의 nHP 멤버에 100을 설정했는데 직접 실행 창에서 150을 설정하고 [그림 2-34] 우측 하단의 자동 창을 보면 npc의 nHP가 150으로 변경되어 있다. 직접 실행하기 전인 [그림 2-33]을 보면 nHP는 원래 코드 대로 100이다).

직접 실행할 때 한 가지 주의할 점은 [그림 2-33]에 있는 NPC 구조체의 멤버 중 int HPUp() 멤버가 인라인으로 되어 있으면 직접 실행으로 HPUp() 멤버를 호출할 수 없다는 점이다.

```
struct NPC
{
    int nHP;

    int HPUp()
    {
        if( nHP > 100 ) {
            return 120;
        }

        return nHP + 50;
    }
};
```

앞의 코드는 NPC 선언 안에 HPUp() 멤버를 정의해서 암묵적으로 인라인화된다.

2.11 DataTips

회사 업무를 하면서 기억해야 할 사항이 있는데 그 내용이 길지 않다면 보통 메모지를(포스트잇 같은) 모니터나 책상에 붙여 놓는다. 왜냐하면 메모지는 우리가 눈에 잘 띠는 곳에 놓아두면 잊어버리지 않고 볼 수 있기 때문이다. 디버깅할 때도 중요한 사항은 메모를 하고 그것을 눈에 잘 띠는 곳에 놓아두면 좋을 것이다. 디버깅 할 때 관련된 정보를 볼 수 있는 눈에 잘 띠는 곳을 찾는다면 그것은 Visual C++ 내부일 것이다. 그리고 메모를 하기에 가장 적합한 기능이 DataTips다.

디버깅할 때 DataTips에 필요한 내용을 메모하면 디버깅할 때 참고할 수 있다. 이 내용은 디버깅이 끝나도 사라지지 않는다. 말보다는 직접 보는 것이 쉽게 이해가 갈 테니 [그림 2-35]를 보자.

[그림 2-35] DataTips 설정

```
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 위치시키면 위처럼  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 메모창이 보인다

DataTips가 설정되었다는 표시

DataTips를 설정한 후 디버깅과 관련된 메모를 입력한다. DataTips는 에디터 내에서 마음대로 이동할 수 있다.

[그림 2-36] DataTips에 메모 추가 및 이동

The screenshot shows two code snippets from the Visual Studio IDE. The top snippet is located at line 32 of the HBPlayerMgr::Release() function. It contains a for loop that iterates from 0 to nCount, calling SAFE_DELETE on each element of m_Players. A DataTip is shown for the variable nCount, which is annotated with '플레이어의 최대 수' (Maximum number of players). The bottom snippet is also at line 32 of the same function. It includes a call to m_Players.clear() and sets m_nMaxPlayerCount to 0. Another DataTip is shown for nCount, also annotated with '플레이어의 최대 수'. Both DataTips have a small icon with a plus sign and a minus sign, indicating they can be collapsed or expanded.

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

```
32 void HBPlayerMgr::Release()
33 {
34     size_t nCount = m_Players.size();
35
36     for( size_t i = 0; i < nCount; ++i )
37     {
38         SAFE_DELETE( m_Players[i] );
39     }
40
41     m_Players.clear();
42
43     m_nMaxPlayerCount = 0;
44 }
```

디버깅이 끝나면 DataTips는 숨긴다.

[그림 2-37] 디버깅이 끝난 후

```
32 void HBPlayerMgr::Release()
33 {
34     size_t nCount = m_Players.size();
35
36     for( size_t i = 0; i < nCount; ++i )
37     {
38         SAFE_DELETE( m_Players[ i ] );
39     }
40
41     m_Players.clear();
42
43     m_nMaxPlayerCount = 0;
44 }
```

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

[그림 2-38] 디버깅이 끝난 후 DataTips 내용 확인

```
32 void HBPlayerMgr::Release()
33 {
34     size_t nCount = m_Players.size();
35
36     for( size_t i = 0; i < nCount; ++i )
37     {
38         SAFE_DELETE( m_Players[ i ] );
39     }
40
41     m_Players.clear();
42
43     m_nMaxPlayerCount = 0;
44 }
```

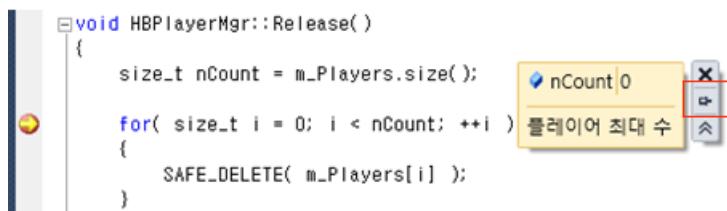
마우스 커서를 DataTips 마크 위에 가져가면 다음과 같은 내용이 표시된다.

1. 이 마크에 마우스 커서를 가져가면
2. 메모 내용과 마지막 디버그 세션의
값이 출력된다.

마지막 디버그 세션의 값
nCount: 0
플레이어의 최대 수

참고로 [그림 2-38]을 보면 디버깅 상태가 아닌 경우에는 DataTips 마크가 표시되는데 이 마크가 표시되지 않는다면 다음과 같은 상황이니 반드시 확인해야 한다.

[그림 2-39] 디버깅이 끝난 후 DataTips 마크가 표시 확인



2.11.1 DataTips 가져오기/내보내기

중단점처럼 DataTips도 xml 파일 형식으로 가져오거나 내보낼 수 있다. 방법은 메뉴의 [디버그]를 선택한 후 [DataTips 가져오기]나 [DataTips 내보내기]를 선택하면 된다.

[그림 2-40] 메뉴의 DataTips 내보내기/가져오기



또 [그림 2-40]을 보면 알 수 있듯이 모든 DataTips는 지울 수도 있고 현재 선택된 파일의 DataTips 전부만을 지울 수도 있다.

2.12 실행 중인 프로그램 디버깅

디버깅 실행이 아닌 일반적인 방법으로 프로그램을 실행했는데 실행 후 프로그램이 이상한 행동을 해서 디버깅을 해야 정확한 문제를 알 수 있을 경우에는 어떻게 해야 할까? 프로그램을 종료하고 Visual C++에서 디버깅을 실행한 후 디버깅해야 할까? 골치 아프게 만드는 버그는 대부분 프로그램을 매번 실행할 때마다 발생하는 버그가 아니라 불특정하게 갑자기 발생하는 버그다. 골치 아픈 버그를 잡으려고 프로그램을 종료한 뒤 Visual C++로 디버깅을 실행하면 이때는 또 버그가 나타나지 않아 버그를 잡는데 많은 시간이 걸리기도 한다.

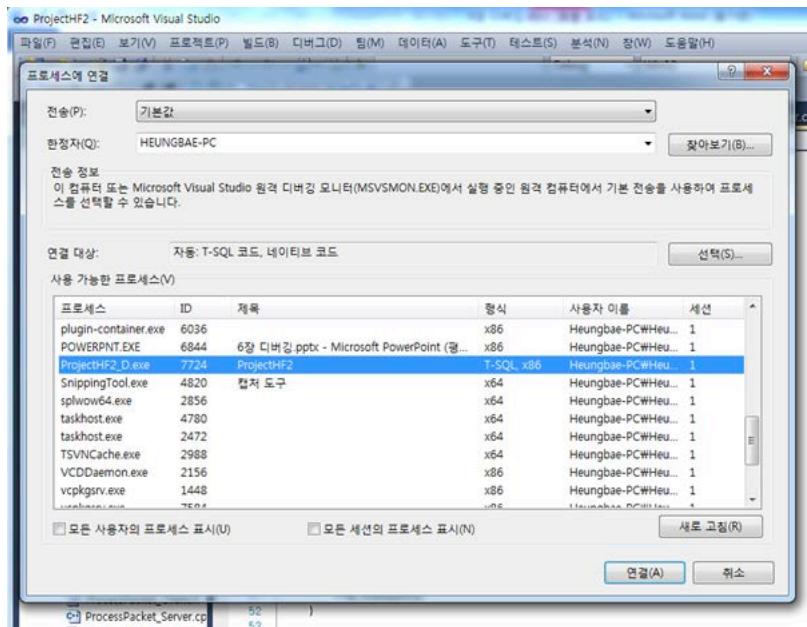
이런 경우를 대비해서 Visual C++은 실행 중인 프로그램을 디버깅할 수 있는 기능을 지원한다. 바로 디버그 기능 중 ‘프로세스에 연결’ 기능을 사용하면 된다. 이 기능을 사용하려면 메뉴의 [디버그] → [프로세스에 연결]을 선택하면 된다.

[그림 2-41] 프로세스에 연결 기능



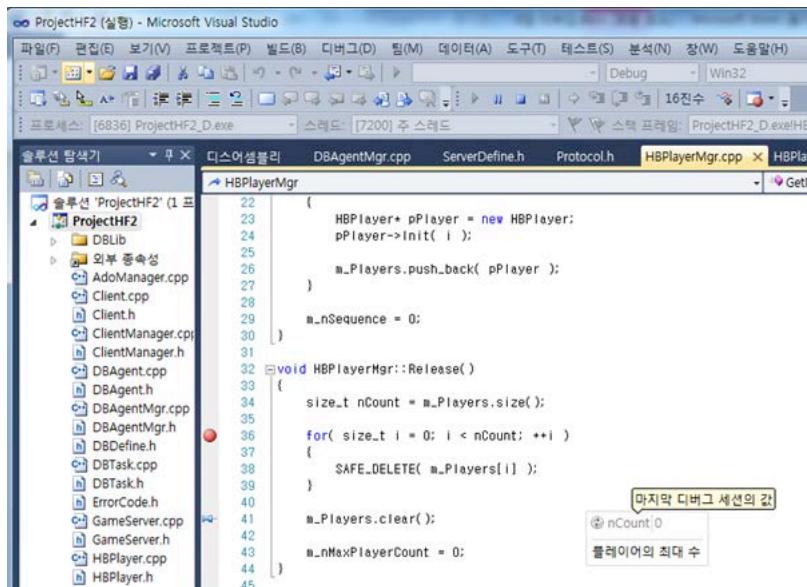
프로세스에 연결 기능을 이용해 디버깅을 하려면 먼저 디버깅할 프로그램의 프로젝트를 연 후 메뉴에서 [프로세스에 연결]을 선택한다.

[그림 2-42] 프로세스에 연결 실행



[그림 2-42]는 ProjectHF2_D.exe라는 프로그램을 실행한 후 이 프로그램의 Visual C++ 프로젝트를 열고, ‘프로세스에 연결’ 메뉴를 실행한 화면이다. 여기서 하단의 연결 버튼을 누르면 프로젝트와 실행 중인 프로그램이 서로 연결되면서 다음과 같이 디버깅이 실행된다.

[그림 2-43] 프로세스에 연결을 사용하여 디버깅하기



여기서부터는 일반적인 디버깅처럼 진행하면 된다. 이 기능은 프로그램의 코드만 가지고 있으면 활용할 수 있는 대단히 유용한 기능이니 반드시 기억하기를 바란다.

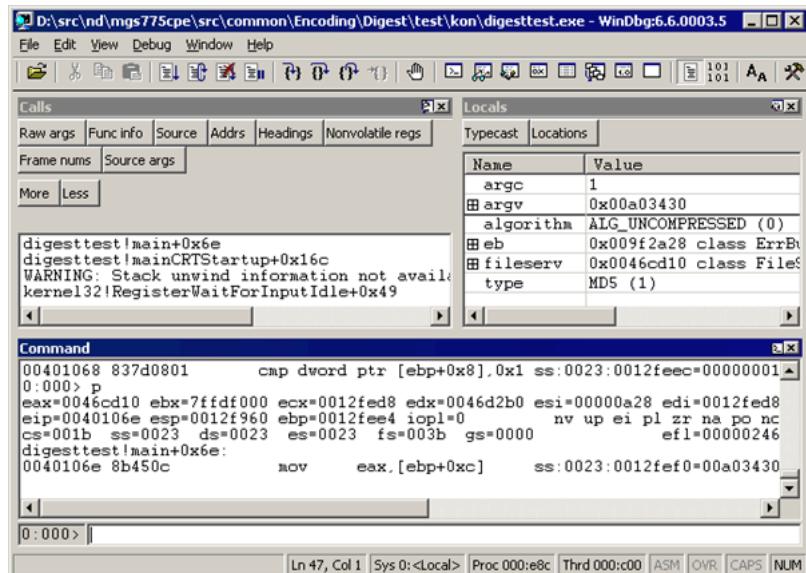
2.13 덤프 파일 디버깅

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

이를 대비해 프로그램에 어떠한 설정을 하면 갑자기 죽어버릴 때 ‘덤프 파일’이라는 것을 남길 수가 있다. 이 덤프 파일만 있으면 프로그램이 죽었을 당시의 상황을 파악할 수 있다. 일종의 블랙박스라고 생각하면 이해가 쉬울 것이다. 덤프 파일의 확장자는 .dmp다.

덤프 파일을 입수하면 이제 이 파일로 디버깅을 하면 된다.

[그림 2-44] WinDbg



예전에는 덤프 파일을 디버깅하려면 ‘WinDbg’라는 다소 불편한 디버그 툴을 이용해야 했으나, Visual C++ 9 이후부터는 Visual C++ 자체에서도 덤프 파일을 디버깅할 수 있게 됐다.

```
#include <cstdio>
#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;
    }

```

앞 코드의 출처는 <http://zhangyafeikimi.javaeye.com/blog/384351>이다.

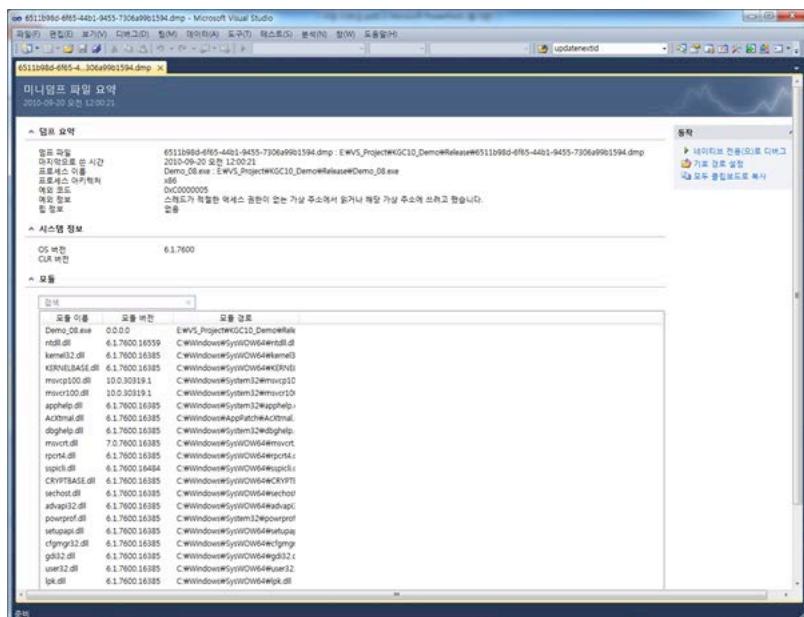
앞 코드는 google-breakpad라는 라이브러리를 사용하여 프로그램이 비정상적으로 종료됐을 때 덤프 파일을 남기게 한다. 코드를 살펴보면 CrashFunction() 함수에서 무조건 프로그램이 죽게 되어 있다. 이것을 빌드해서 실행하면 덤프 파일이 생성된다.

[그림 2-45] 덤프 파일 생성

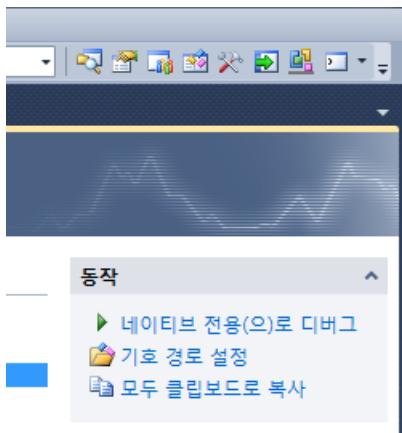
굽기	새 풀더	
이름	수정한 날짜	유형
6511b98d-6f65-44b1-9455-7306a99b1594.dmp	2010-09-20 오전 12:00	Crash Dump
Demo_08.exe	2010-09-19 오후 11:53	응용 프로그램
Demo_08.pdb	2010-09-19 오후 11:53	Program Database

덤프 파일을 더블 클릭하면 Visual C++이 자동 실행된다.

[그림 2-46] 덤프 파일을 통해 Visual C++이 실행된 모습

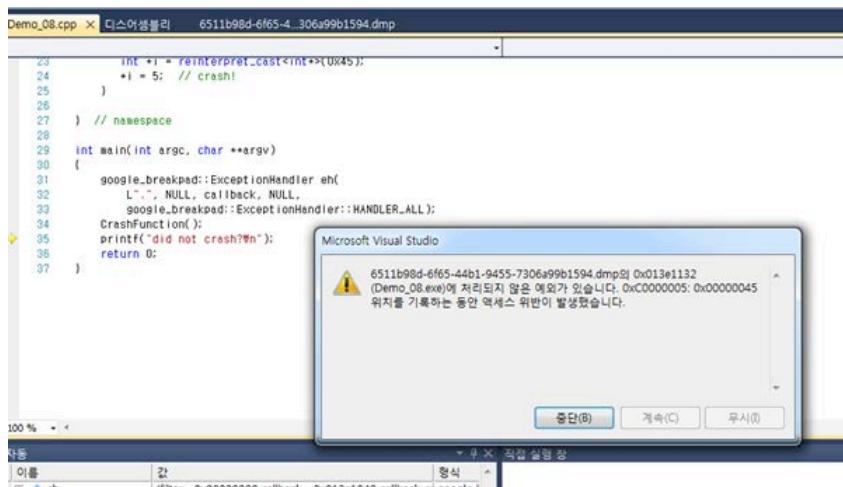


[그림 2-47] 우측 상단이 확대된 모습



[그림 2-46]의 우측 상단에 있는 ‘네이티브 전용(으)로 디버그’를 클릭한다([그림 2-47]). 그러면 [그림 2-48]과 같이 디버깅 모드가 실행되면서 프로그램이 죽은 위치를 알려준다.

[그림 2-48] 덤프 파일 디버깅



덤프 파일을 디버깅하려면 해당 프로그램의 소스 코드와 심볼 파일(.pdb 파일)이 필요하다. Visual C++에서 덤프 파일 디버깅을 지원하기 전까지는 WindDbg라는 프로그램을 따로 설치해야 하고 사용 방법도 어려웠지만(WinDbg에서 디버깅할 때는 커맨드 라인 명령어로 해야 한다) 이제는 덤프 파일 디버깅도 무척 쉬워졌다.

이것으로 Visual C++에 있는 디버깅 기능의 대부분을 배웠다고 할 수 있다. 지금 까지 설명한 것을 잘 기억한다면 어려운 디버깅도 좀 더 쉽고 빠르게 해낼 수 있을 것이다.