



# C++ 프로그래밍

김 형 기

[hk.kim@jbnu.ac.kr](mailto:hk.kim@jbnu.ac.kr)

# Today

---

- 디버깅이란
- 비주얼 스튜디오에서의 디버깅
  - 중단점 사용 방법
  - 메모리 확인 방법

# What is debugging?

- De"bug"

- 버그(=오류)를 제거하는 것

- (Remind) 오류의 종류?

- 컴파일 / 링크 오류 → 오류의 원인이 명확하며, 출력 창을 통해 쉽게 원인 파악 가능

- 런타임 오류
  - 논리적 오류

} 괴로운 오류... 문제 발생 원인을 찾기 어려움

IDE에서 제공하는 디버깅 기능과 테스트를 현명하게 사용해야 문제를 빠르게 해결 가능!

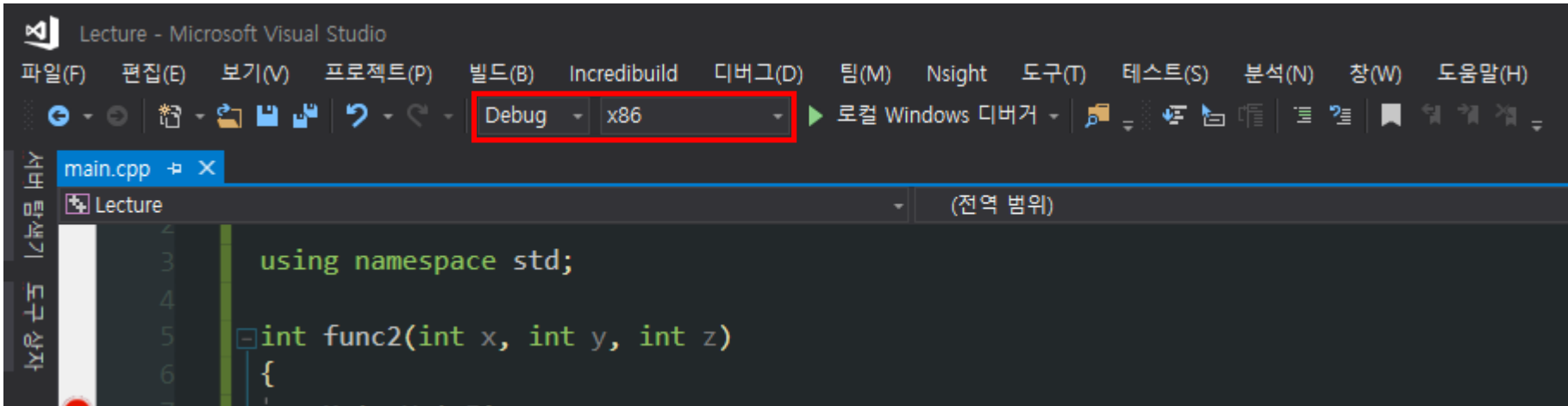
- 중요! 개발 숙련도가 높아지더라도 오류 없는 프로그램을 단번에 개발하는 것은 불가능

- 숙련도가 높아짐에 따라 오류의 "확률"이 줄고, 디버깅 속도가 빨라지는 것

- 즉, 오류를 많이 내고(?) 스스로 디버깅을 반복적으로 수행하고 고민하는 경험을 쌓는 것이 중요

# Preface

- 디버깅을 하기 전 현재 디버깅이 가능한 구성인지 확인
  - Release 모드에서는 값을 제대로 볼 수 없음

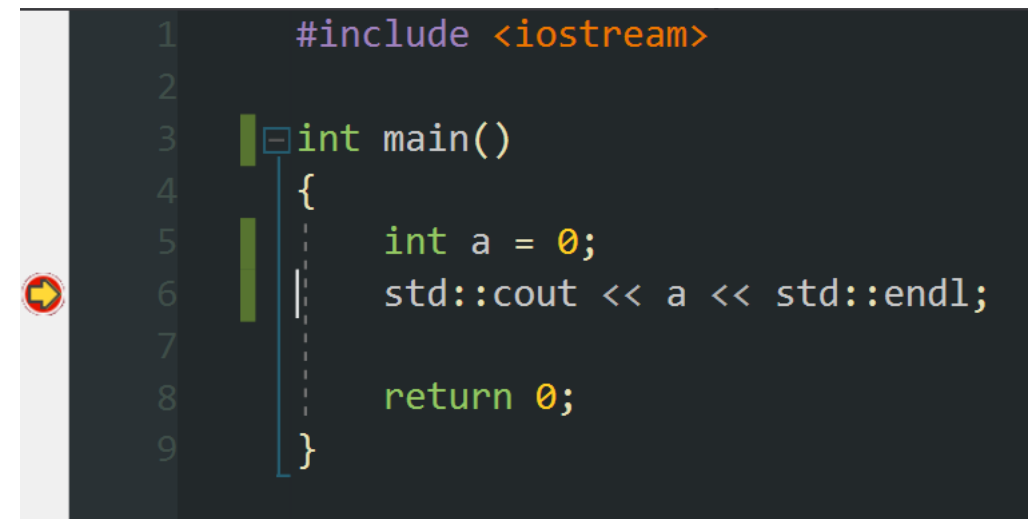
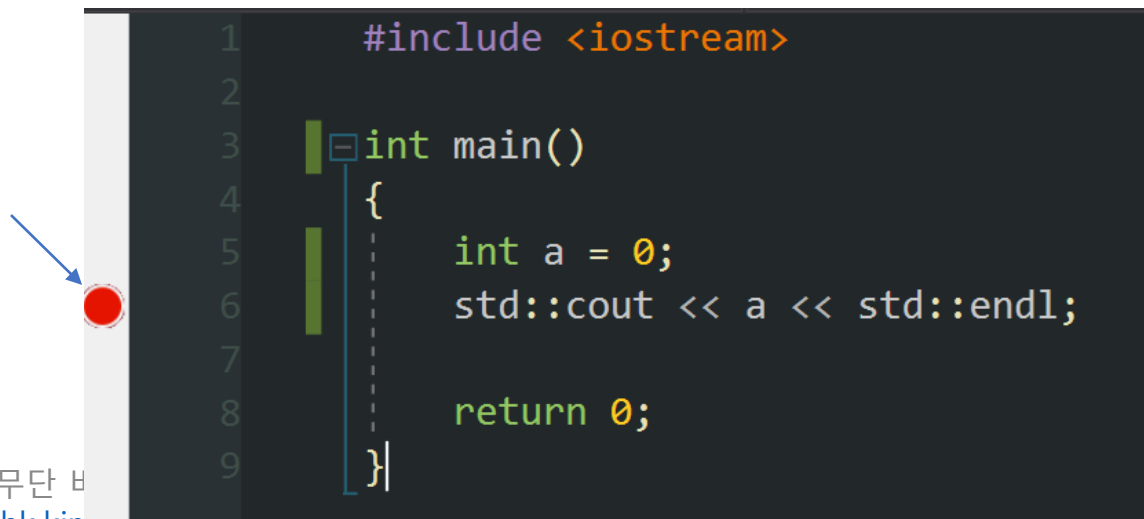


- 특별한 이유가 없으면 x86 플랫폼으로 설정
- 디버그 모드에 들어간 후, 디버그 탭의 창 메뉴를 보면 다양한 디버깅 관련 정보를 띄울 수 있는 창을 선택 가능함

# Debugging Tool 1 – Break point

## ● 중단점

- 임의의 위치에서 프로그램을 정지시킬 수 있는 기능
- 정지하고 싶은 위치의 왼쪽 끝을 클릭하거나 F9을 통해 중단점 생성 가능
- 이후 F5(디버깅)을 통해 프로그램을 빌드 후 실행하면, 해당 위치에서 프로그램이 중단됨
  - Ctrl+F5가 아님에 주의! 솔루션 구성이 디버그 상태여야 함에 주의!
  - 현재 중단되어 있는 위치가 노란색 화살표로 표시됨 (해당 명령문은 아직 실행되지 않은 상태)



# Debugging Tool 1 – Break point

## ● 중단점, 계속

■ 이제, 중단 위치에서 F10과 F11을 통해 디버깅을 진행함

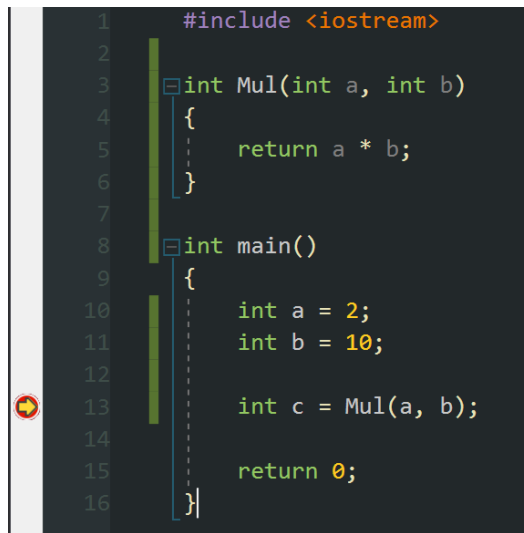
➤ F10 : 해당 명령문 실행하고 다음 명령줄로 진행 (step over)

➤ F11 : 명령문 내에서 호출하는 추가 코드(ex, 함수)로 들어감 (step into)

➤ F5 : 다음 중단점까지 진행

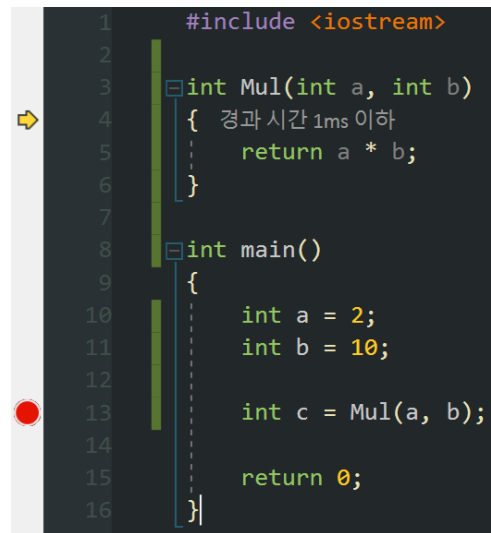
➤ Shift+F11 : 현재 명령문이 호출되는 상위 stack으로 나가기 (step out)

함수 강의 이후에 꼭 다시  
돌아와 확인해보세요!



```
1 #include <iostream>
2
3 int Mul(int a, int b)
4 {
5     return a * b;
6 }
7
8 int main()
9 {
10     int a = 2;
11     int b = 10;
12
13     int c = Mul(a, b);
14
15     return 0;
16 }
```

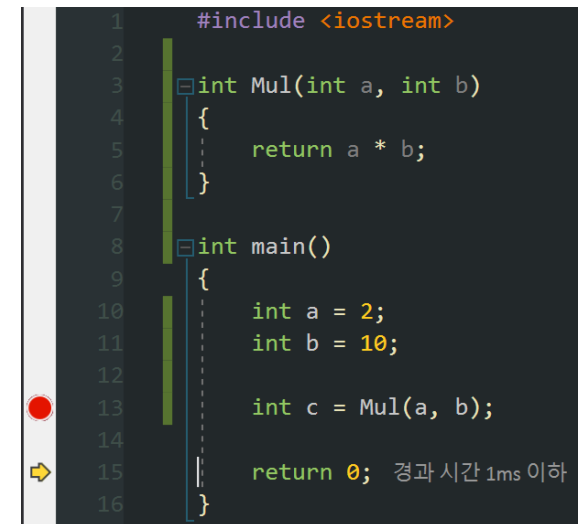
A red circle with a yellow arrow indicates a break point is set at line 13, the call to the `Mul` function.



```
1 #include <iostream>
2
3 int Mul(int a, int b)
4 { 경과 시간 1ms 이하
5     return a * b;
6 }
7
8 int main()
9 {
10     int a = 2;
11     int b = 10;
12
13     int c = Mul(a, b);
14
15     return 0;
16 }
```

The execution has stepped into the `Mul` function at line 4. A yellow arrow points to the start of the function.

F11을 누르면 Mul 계산  
코드 안으로 들어감



```
1 #include <iostream>
2
3 int Mul(int a, int b)
4 {
5     return a * b;
6 }
7
8 int main()
9 {
10     int a = 2;
11     int b = 10;
12
13     int c = Mul(a, b);
14
15     return 0; 경과 시간 1ms 이하
16 }
```

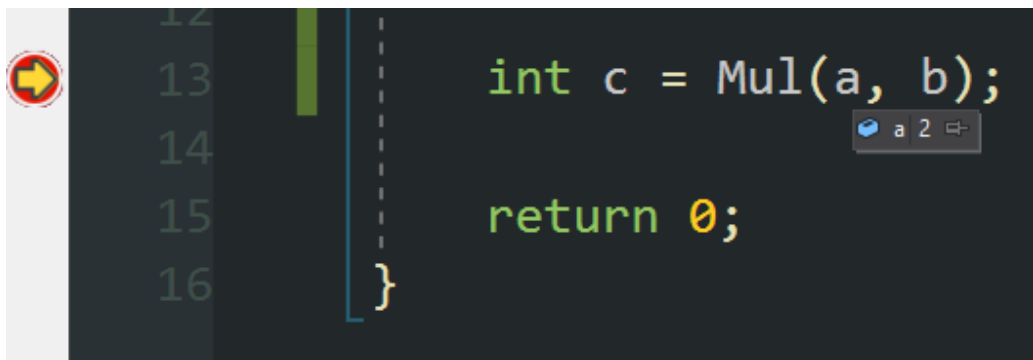
The execution has stepped over the `Mul` function call and is now at line 15 in the `main` function. A yellow arrow points to the next line.

F10을 누르면 해당 코드를  
실행하고 다음 줄로 넘어감

# Debugging Tool 1 – Break point

## ● 중단점, 계속

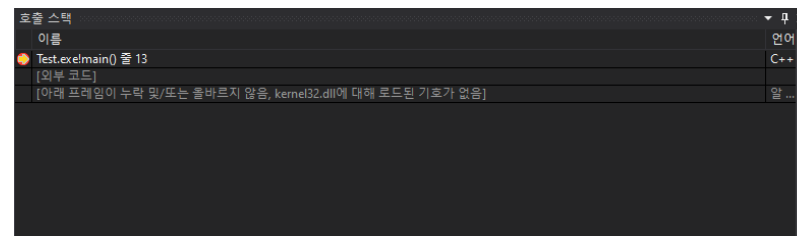
- 그래서? 이 과정에서 해야 할 것은?
- 데이터(메모리) 확인!
  - 변수에 마우스를 올려 값을 확인하거나,
  - 하단 창을 통해 값을 확인



마우스를 값을 보기를 원하는 변수 위에 올려두면, 현재 해당 변수에 저장되어 있는 값이 표시됨

자동			
이름	값	형식	
a	2	int	
b	10	int	
c	-858993460	int	

하단의 창을 통해서도 값을 확인할 수 있음 계속 진행하면서 값이 변하는지, 어떻게 변하는지 확인해 보기  
방금 명령줄에 의해 값이 변한 변수가 빨간색으로 표시됨  
로컬/자동/조사식 중 "로컬"창을 주로 보기를 권장

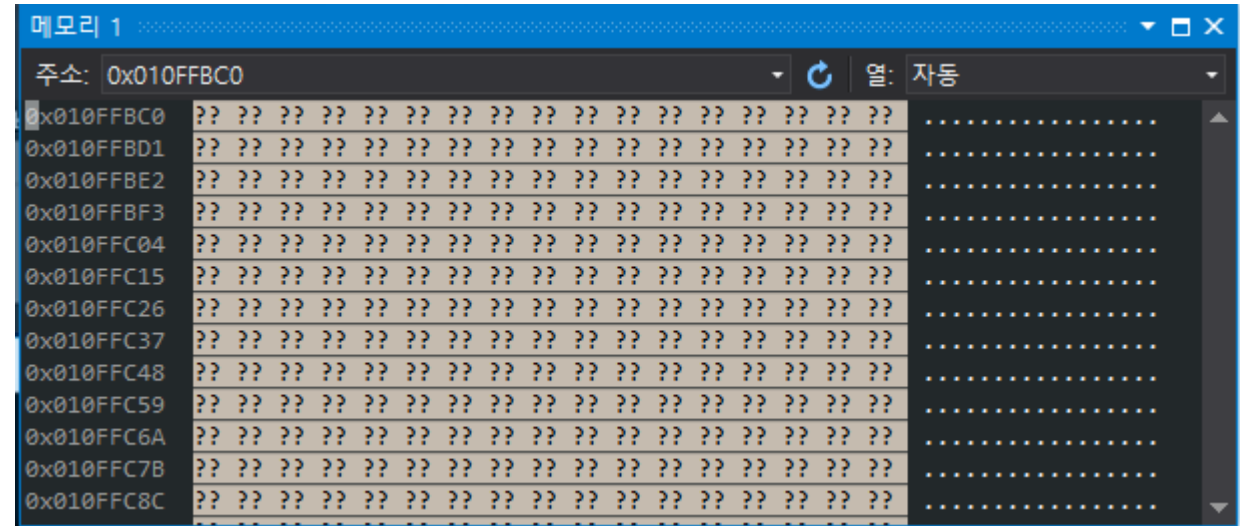
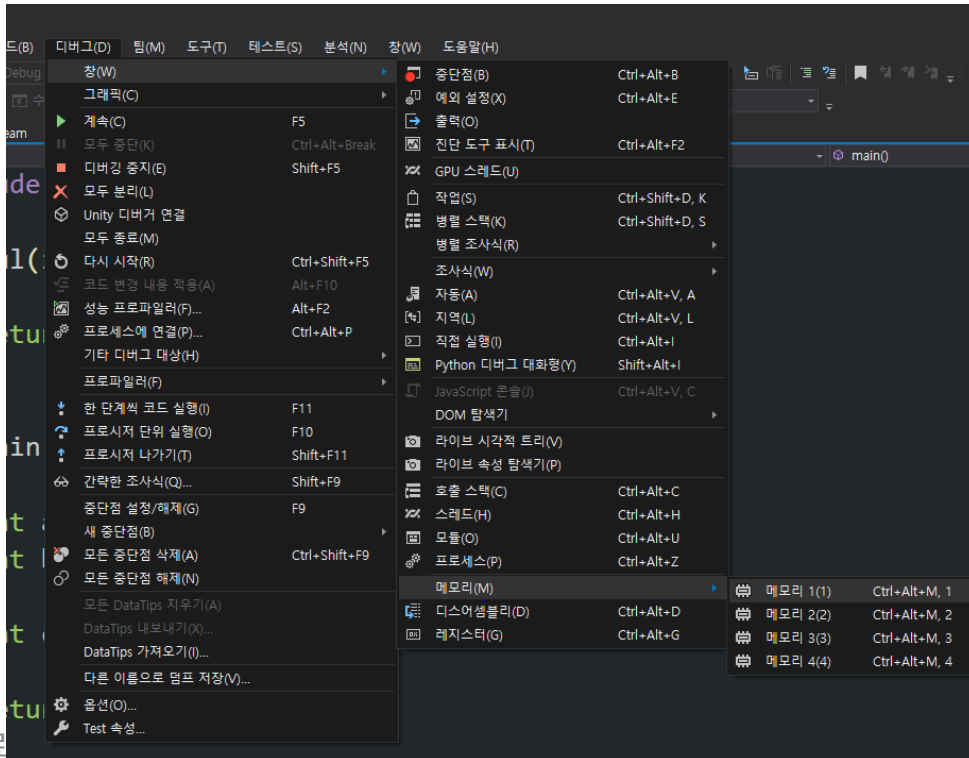


호출 스택 또한 자세히 살펴보는 것이 좋음. 추후 추가 설명

# Debugging Tool 2 – Memory Window

## ● 메모리 윈도우

- 실제 내가 생성한 변수가, 메모리 상에 어떻게 저장되어 있는지 자세히 보려고 할 때 사용
- 디버그→창→메모리→메모리 1 클릭



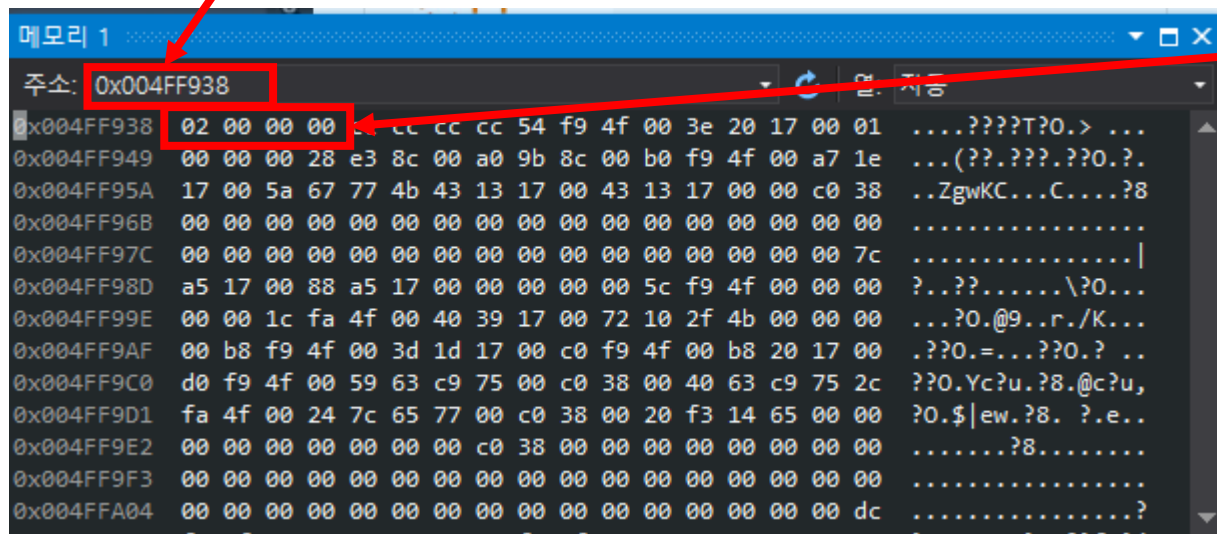


# Debugging Tool 2 – Memory Window

- 메모리 윈도우, 계속

- 변수가 메모리에 어떻게 저장되었는지 보기 위해 주소 부분에 "&변수이름"을 입력
  - &는 변수의 주소를 얻어오기 위한 연산자
- 중단점과 마찬가지로 코드 진행에 따라서 변수값이 계속 업데이트됨

&a를 입력해서 현재 a 변수가 저장된 메모리 주소 (0x004FF938)의 메모리 값을 살펴봄



A는 integer이므로 4 byte 메모리 공간을 차지하고 있고, 그 메모리에 저장된 값을 16진수를 통해 보여줌

# How to debugging?

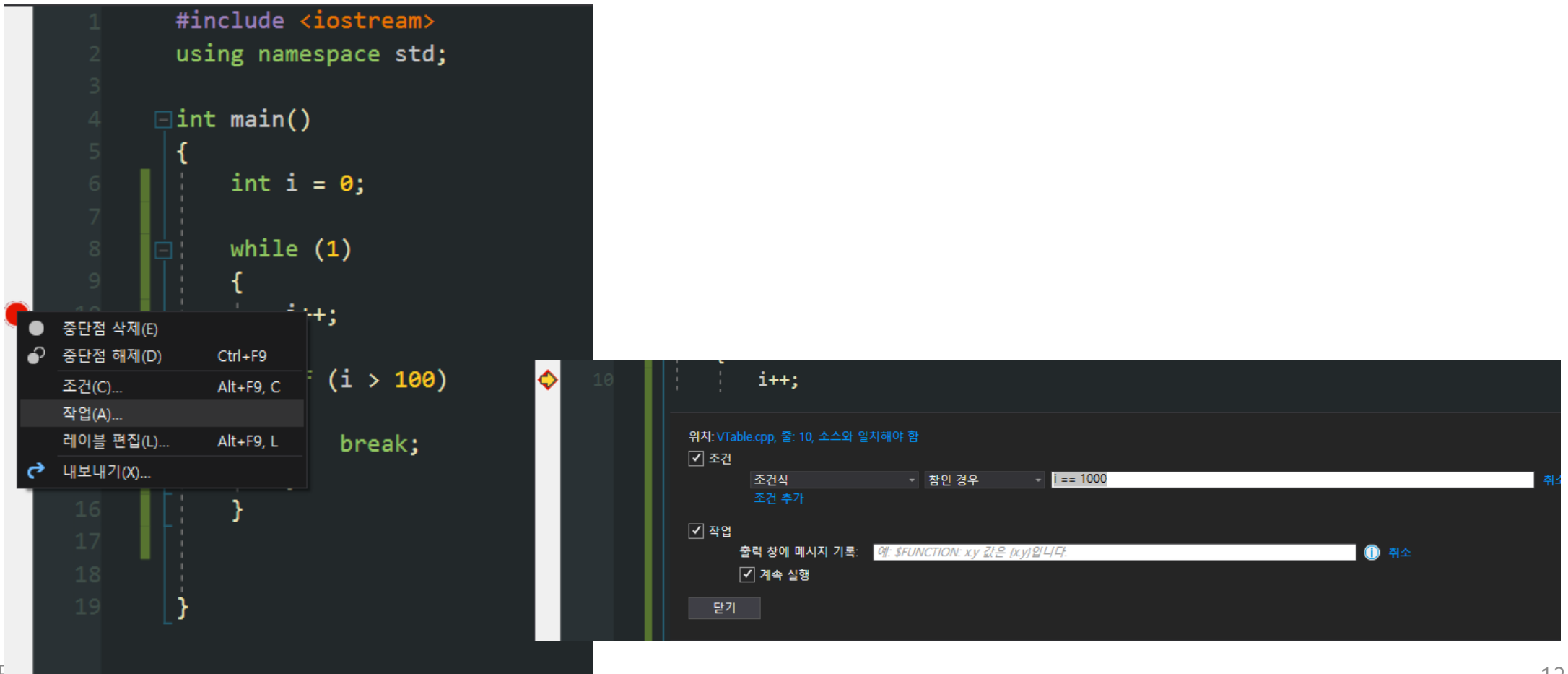
---

- 디버깅 기능을 활용해 살펴 보아야 할 것
- 프로그램의 흐름
  - 내가 생각하는 순서대로 프로그램의 명령문들이 실행되고 있는지 확인
- 프로그램에서 가장 중요한 것은 "데이터"
  - 메모리에 저장된 데이터(변수/객체)가, 내가 의도한 것과 같은 지 확인
- Tip : 우클릭 → "정의로 이동" 으로 해당 문법의 정의 위치를 손쉽게 찾을 수 있음

추가 슬라이드

# Conditional & Action breakpoint

- 중단점 우클릭 → 작업 창을 통해 특정 조건에서 중단점 활성화 가능
  - Loop의 디버딩에 유용하며, 재 컴파일이 필요 없음



# Disassembly & Register

- 디버그 → 창 → 디스어셈블리 / 레지스터

```
0061170C rep stos     dword ptr es:[edi]
0061170E mov         ecx,offset _A14CD92F_main@cpp (061B000h)
00611713 call        @__CheckForDebuggerJustMyCode@4 (0611203h)
        char a = 10;
00611718 mov         byte ptr [a],0Ah
        short b = 10;
0061171C mov         eax,0Ah
00611721 mov         word ptr [b],ax
        int c = 10;
00611725 mov         dword ptr [c],0Ah
    }
0061172C xor         eax,eax
0061172E pop         edi
0061172F pop         esi
00611730 pop         ebx
00611731 add         esp,0E4h
00611737 cmp         ebp,esp
```

명령문의 변환된 어셈블리 코드를 볼 수 있음

```
레지스터
EAX = 0061B000 EBX = 00901000 ECX = 0061B000 EDX = 00000001
ESI = 00611320 EDI = 00AFF7F4 EIP = 00611718 ESP = 00AFF704
EBP = 00AFF7F4 EFL = 00000246
```

현재 시점에 레지스터에 저장된 값들을 볼 수 있음