IDA 5.x Manual

- Manual 01 -

ASM Love
어셈블리어 개발자그룹
[어셈러브] asmlove.co.kr

Last Update 2007. 02

이강석 / certlab@gmail.com

어셈블리어 개발자 그룹 :: 어셈러브

http://www.asmlove.co.kr

IDA Pro 는 Disassembler 프로그램입니다.
기계어로 되어있는 실행파일을 어셈블리언어 형태로 변환시켜주는 프로그램이죠.
Disassembler 종류로는 IDA Pro 말고도 W32dasm, .NET Reflector 등이 있습니다.
IDA는 Linux 버전과 Windows 버전이 있으며 이 문서에서는 Windows IDA Pro 5.0을 기준으로 설명할까 합니다.

IDA의 기능중 특히 FLIRT(Fast Library Identification and Recognition Technology)는 기계어의 코드로부터 컴파일러 특유의 Library 함수를 산출해 낼수 있는 강력한 기능과 PIT(Parameter Identification and Tracking)는 파라미터 사용을 철회할 수 기능이 있습니다.

또한, IDA는 많은 CPU를 지원합니다. 참고 : http://www.datarescue.com/idabase/idaproc.htm

```
AMD K6-2 3D-Now! extensions
ARM Architecture version 3, 4 and 5 including Thumb Mode and DSP instructions. Updated in 4.9, ARM/WinCE debugger.
ATMEL AVR (comes with source code)
DEC PDP-11(comes with source code)
Fujitsu FR (comes with source code)
GameBoy
H8/300 , H8/300L , H8/300H, H8S/2000 , H8S/2600(comes with source code)
H8/500(comes with source code)
Hitachi HD 6301, HD 6303, Hitachi HD 64180
INTEL 8080
INTEL 8085
INTEL 80196 (comes with source code)
INTEL 8051 (comes with source code)
INTEL 860XR (comes with source code)
INTEL 960 (comes with source code)
INTEL 80x87 and 80x87
INTEL Pentium family
Java Virtual Machine (comes with source code)
KR1878 (comes with source code)
Microsoft .NET
Mitsubishi MELPS740(comes with source code)
MN102 (comes with source code)
MOS Technologies 6502 (comes with source code)
Motorola MC680xx. , Motorola CPU32 (68330), Motorola MC6301, MC6303,
MC6800, MC6801, MC6803, MC6805, MC6808, MC6809, MC6811, M68H12C
Motorola ColdFire
NSC CR16 (comes with source code)
PIC 12XX, PIC 14XX, PIC 18XX, PIC 16XXX (comes with source code)
Rockwell C39 (comes with source code)
SAM8 (comes with source code)
SGS Thomson ST-7, and ST-20 (comes with source code)
TLCS900 (comes with source code)
XA (comes with source code)
xScale
Z80, Zilog Z8, Zilog Z180, Zilog Z380 (comes with source code)
```

지금 이 문서에서 중요한 부분인데 언급만 한점과 설명이 부족한 부분들은 계속되는 업데이트를 통해 보충을 할것입니다.

Download

IDA Pro 5.0 Evaluation version [17.6MB]
http://www.datarescue.com/idabase/idadowndemo.htm

IDA Pro 4.3 Free Version [11.4MB]
http://www.datarescue.be/idafreeware/freeida43.exe

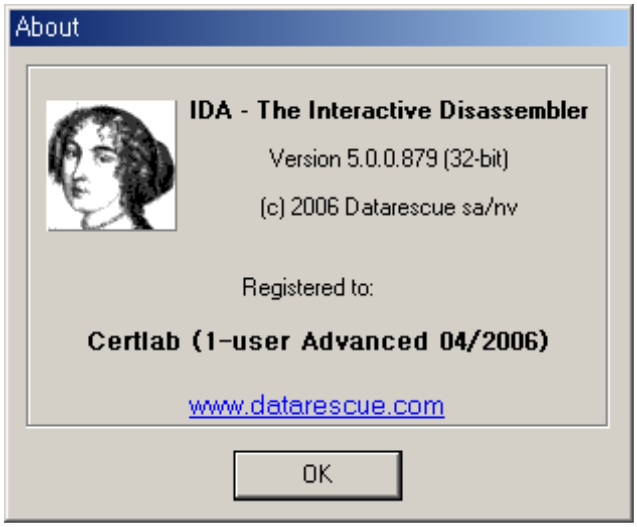
기타 IDA에 자세히 알고 싶다면 Datarescue 사이트를 참고하시기 바랍니다. http://www.datarescue.com

목 차

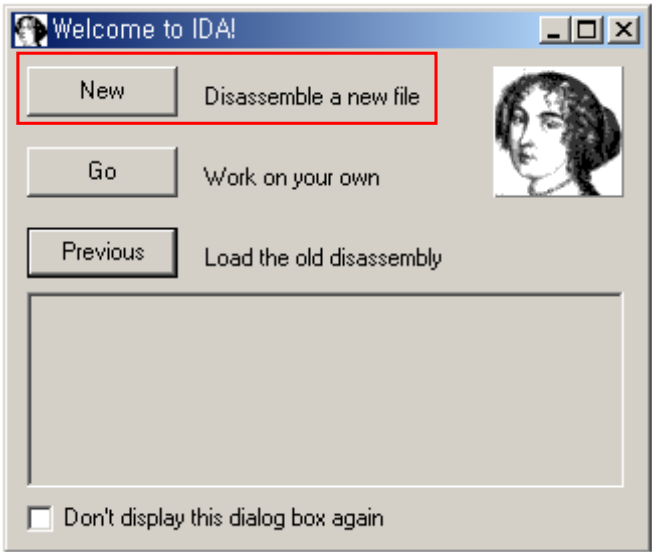
시작 하기에 앞서 디어셈블러를 통해 분석을 하고 디버깅을 하는 제일 기본적인 이유는 다른 이유도 많이 있겠지만 프로그램의 진행흐름과 메모리에서의 실행흐름과 나아가 "프로그램" 을 이해하는 것입니다. 이 문서를 기본 Guideline으로 좋은 목적을 갖고 시스템에 한걸음 깊숙이 접근하셨으면 좋겠습니다.

IDA Start

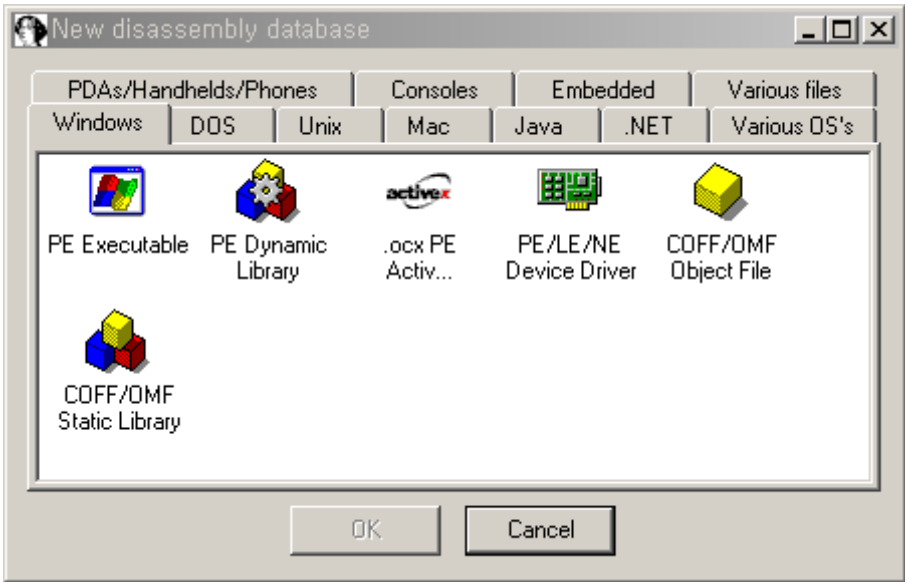
IDA를 실행하면 다음과 같이 IDA 버전과 사용자명이 나오는 About 창이 나오는데 OK 를 누릅니다. OK를 누르지 않아도 몇초후 다음 화면이 나옵니다.



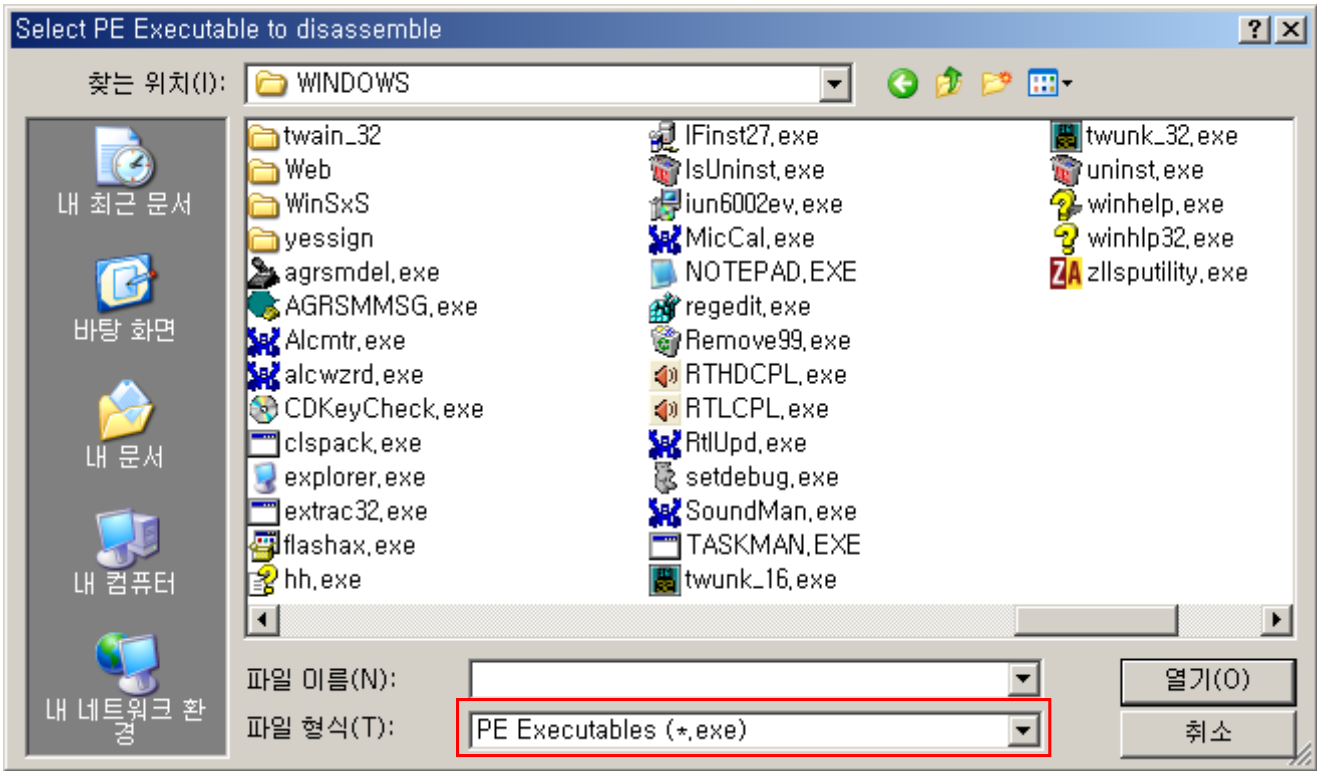
IDA에서 파일을 불러올때 여러 가지 방법이 있는데 처음부터 하나씩 알아보도록 하겠습니다.
이 화면에서 New를 누릅니다.



New를 누르면 다음화면을 볼수 있는데 파일의 Type을 직접 선택한후 파일을 불러올수가 있습니다.

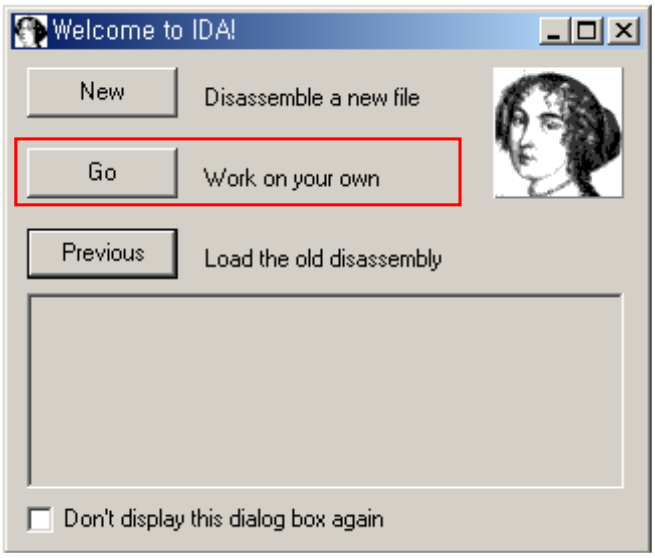


WIndows 의 기본 실행파일 구조인 PE Executable 을 선택하고 나서 OK를 눌러봅니다.
그러면 파일을 선택할수 있는 창이 뜨게 되고, 선택한 PE File을 디어셈블 할수 있게 됩니다.
기본적으로 Windows 에서의 PE File에는 EXE, DLL, OCX 등이 있습니다.

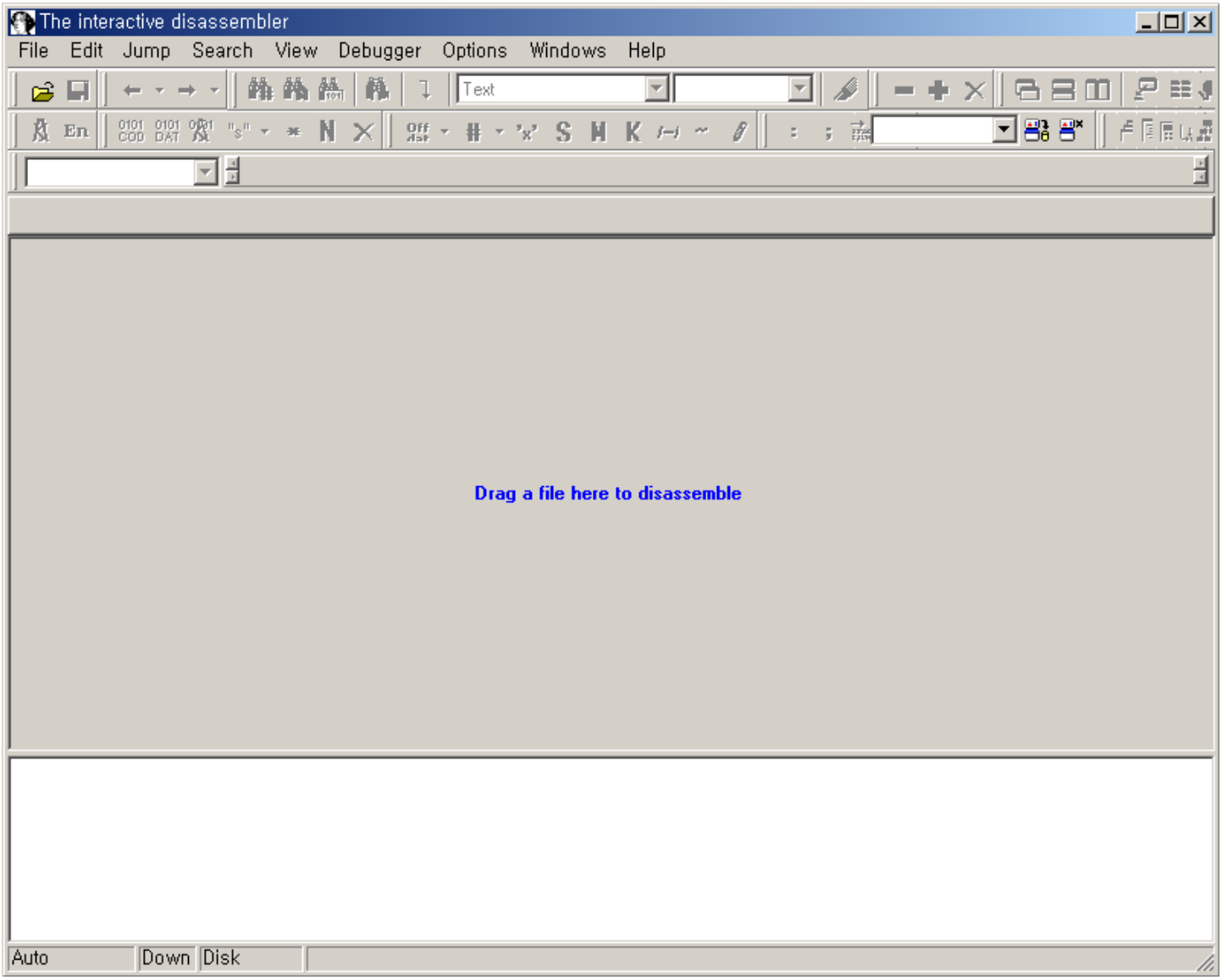


이렇게 선택을 하고 파일을 열었다면 IDA가 분석을 시작하게 됩니다.

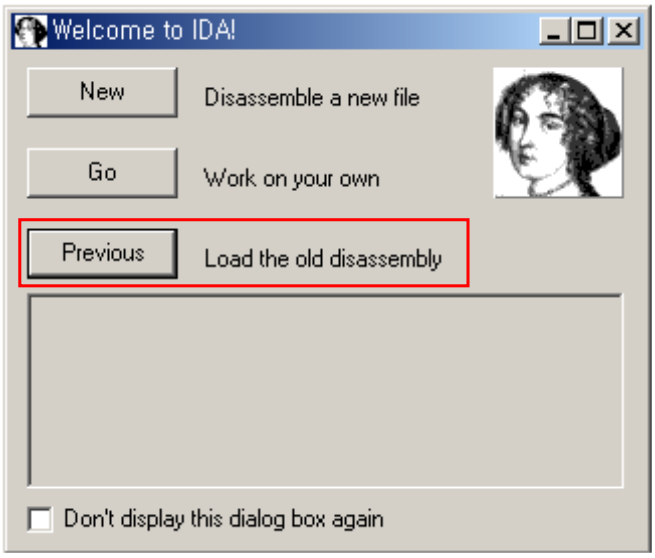
이제 다시 처음으로 돌아와서 Go를 눌러보겠습니다.



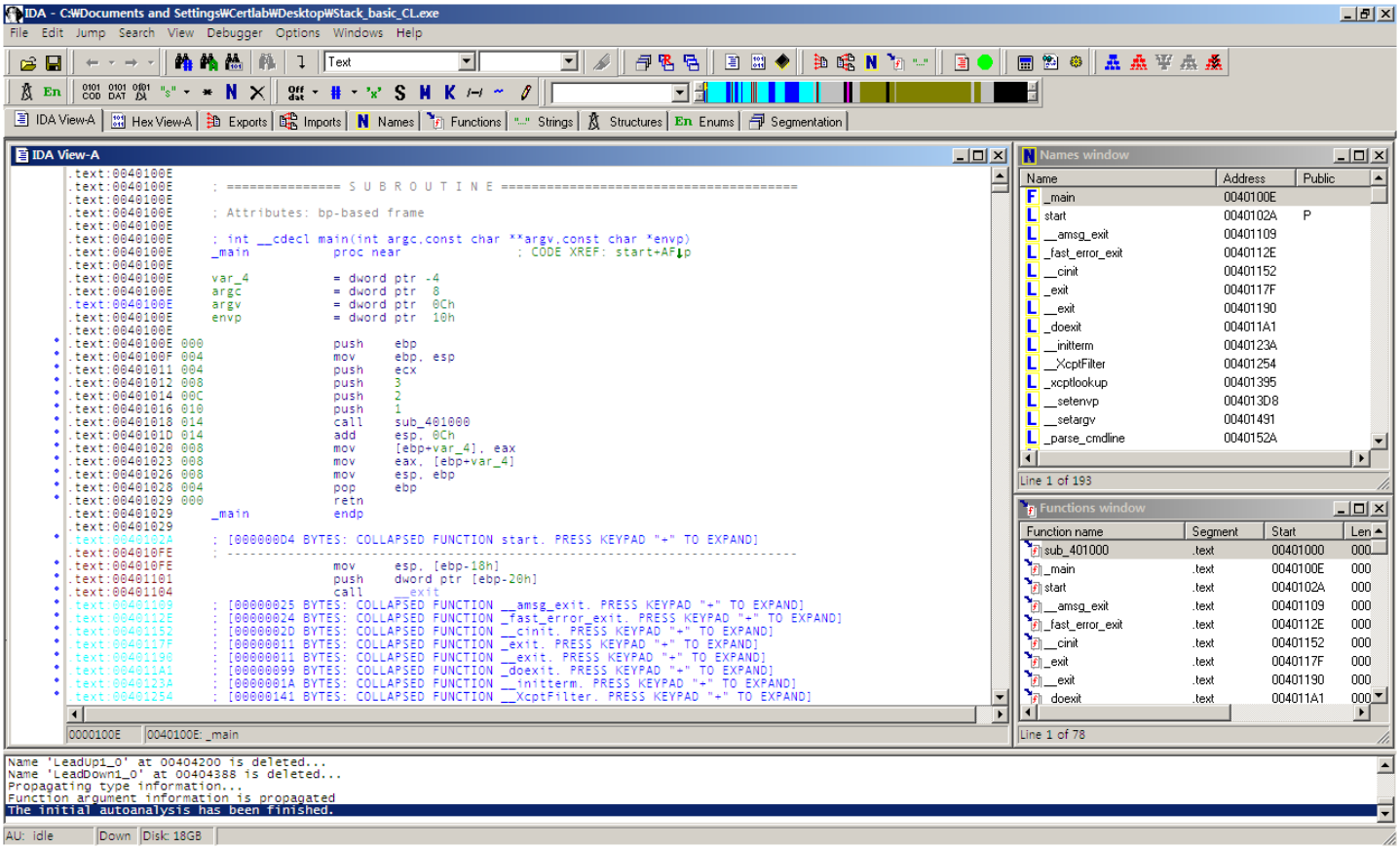
그러면 바로 디어셈블 할수 있는 작업창이 바로 나오는데 디어셈블 할 파일을 Drag 해서 작업창에 올려놓아도 되고 File->Open 을 이용해서 파일을 불러와도 됩니다.



다시 처음으로 돌아와서 Previous를 눌러보겠습니다.



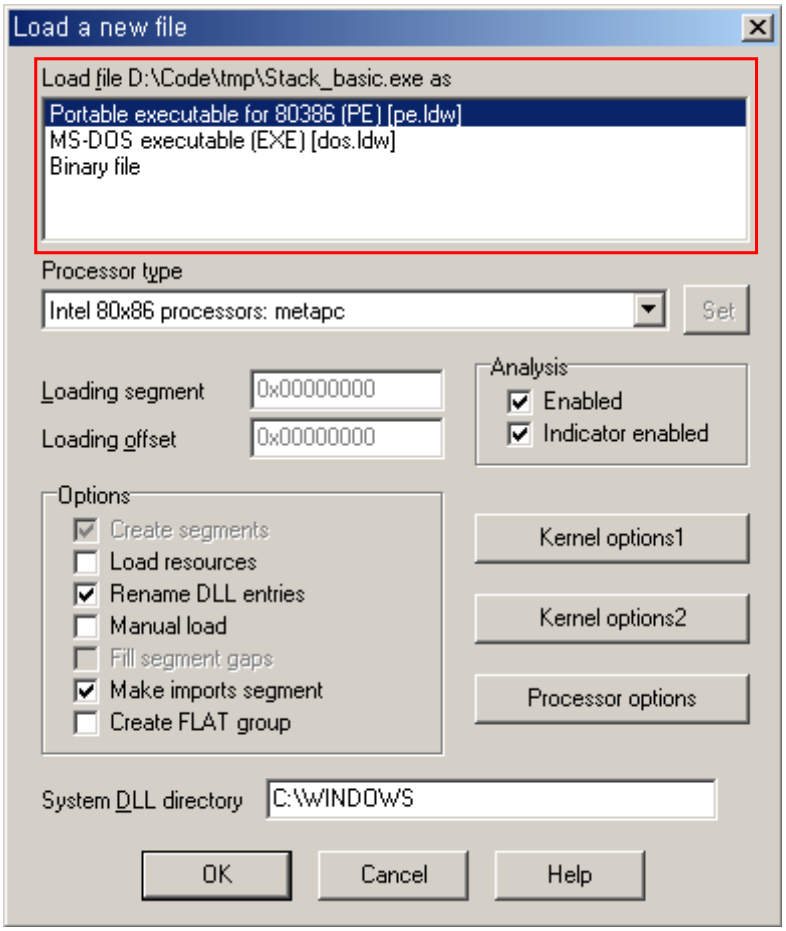
Previous는 작업의 편리성과 신속성을 위해 최근에 작업했던 파일을 자동으로 불러와서 디어셈블 하는것을 볼수 있습니다.



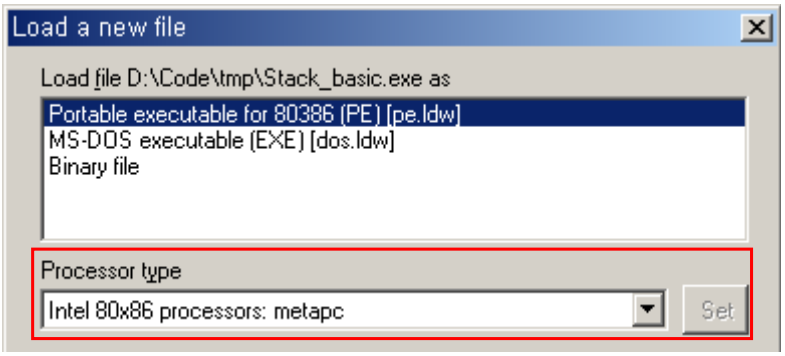
IDA에서 파일을 불러오는 방법을 간단히 알았습니다.

이제 파일을 불러올때의 화면을 자세히 보도록 하겠습니다.
다음은 처음 New를 선택했을때 파일을 불러오는 화면입니다.

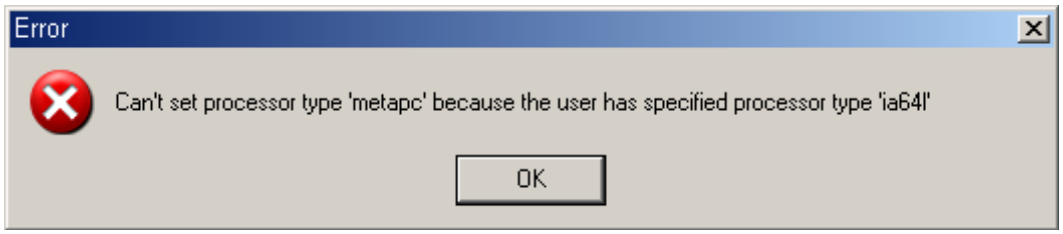
Load file 에서는 디어셈블 할 파일의 Type을 선택하는데 기본적으로 선택되어진 PE Format을 선택합니다.
PE Format에 대해서는 이 문서에서 다루지 않으니 다른문서에서 찾아보시길 바랍니다.
한마디로 PE File은 Windows에서의 실행파일이라고 생각하시면 됩니다.
(Windows에서 보통 PE Format은 exe, dll, ocx 등이 있습니다.)



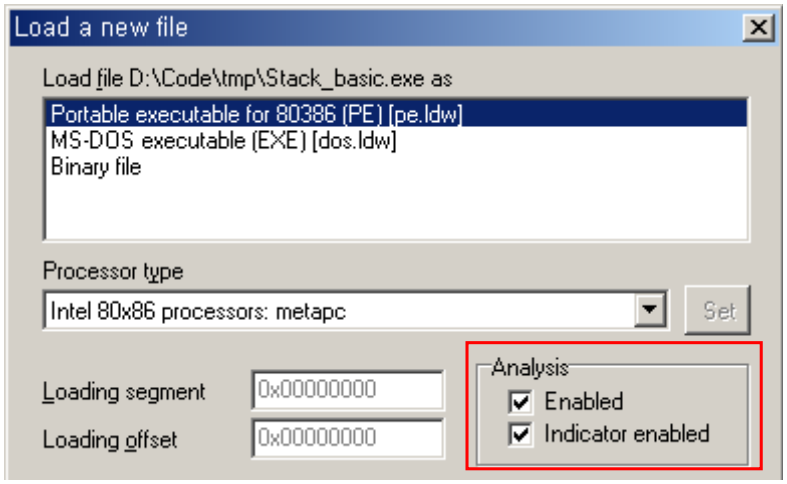
Processor Type에서는 쓰고 있는 컴퓨터의 Processor을 결정합니다.



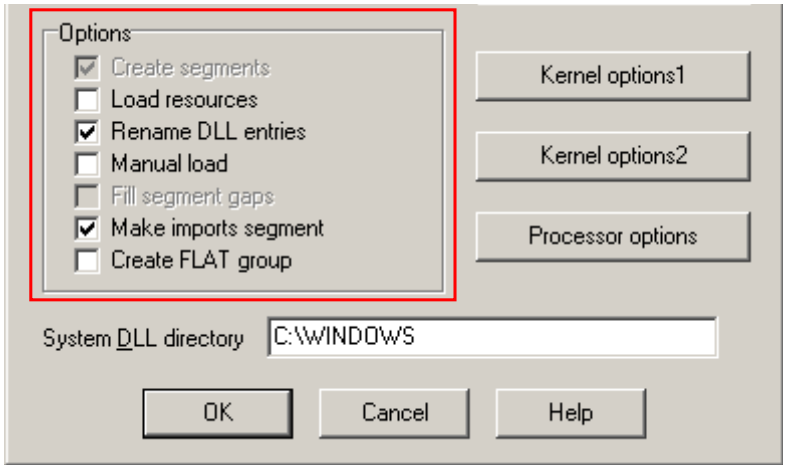
만약 자신의 Processor와 다른 Processor를 선택하면 Error메시지가 나타나면서 바로 종료됩니다.



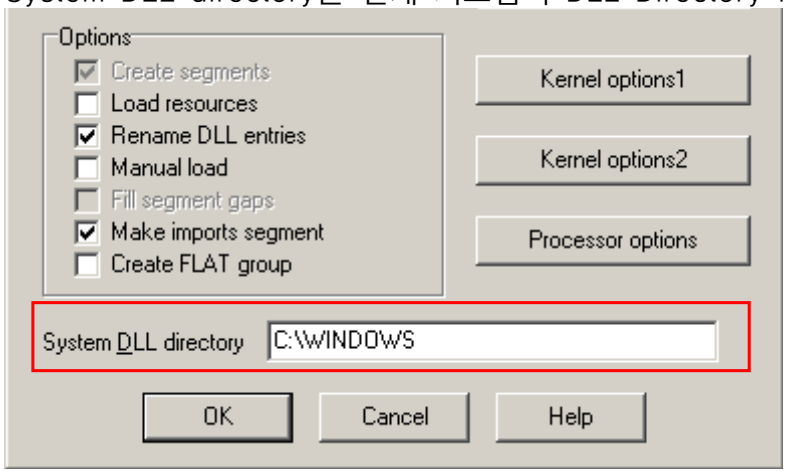
Analysis에서 Enabled, Indicator enabled 옵션을 켜야 합니다.



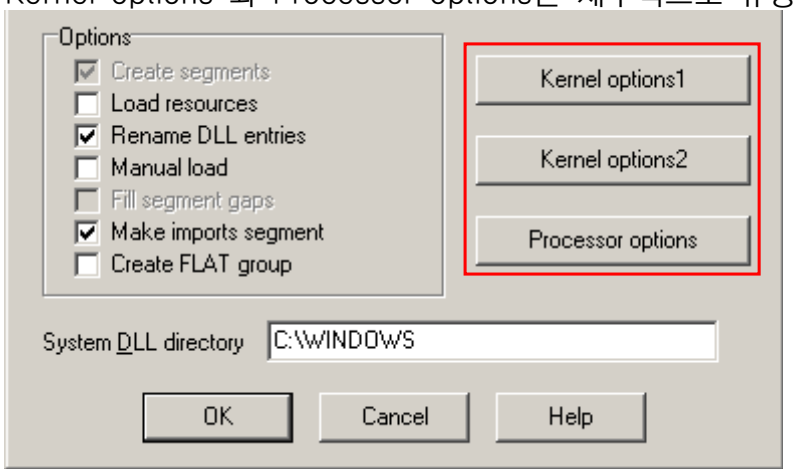
Options에서는 아래 두 개의 옵션에 체크 합니다.



System DLL directory는 현재 시스템의 DLL Directory Path를 넣는곳입니다.

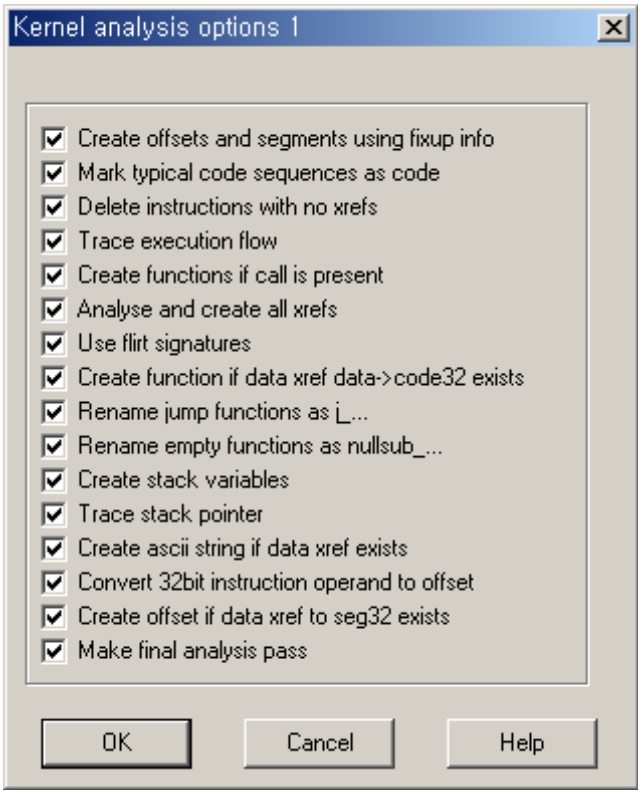


Kernel options 와 Processor options는 세부적으로 튜닝할 때 설정하는데 다음에 다루도록 하겠습니다.

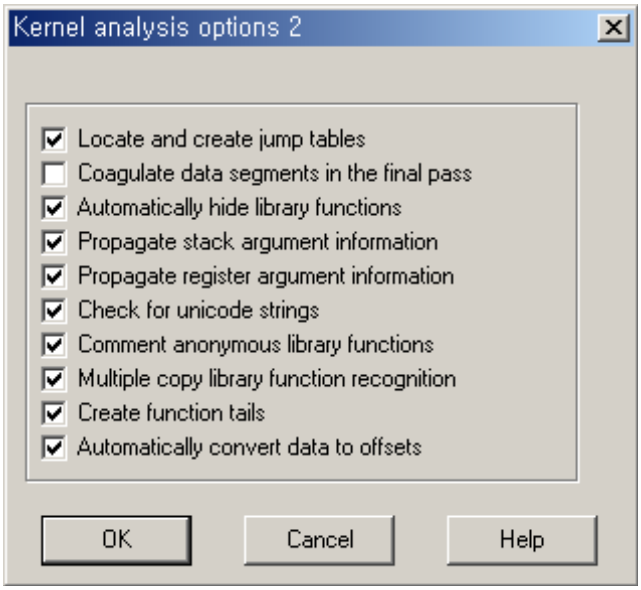


바로 위의 화면에서 빨간색으로 표시된 각 버튼을 누르면 아래와 같은 화면을 볼수 있습니다.

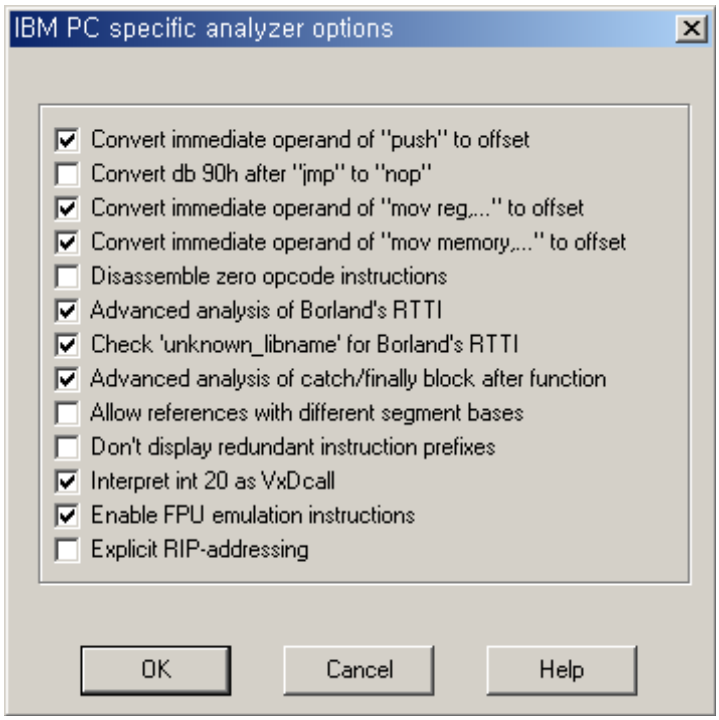
Kernel options1



Kernel options2



Processor options



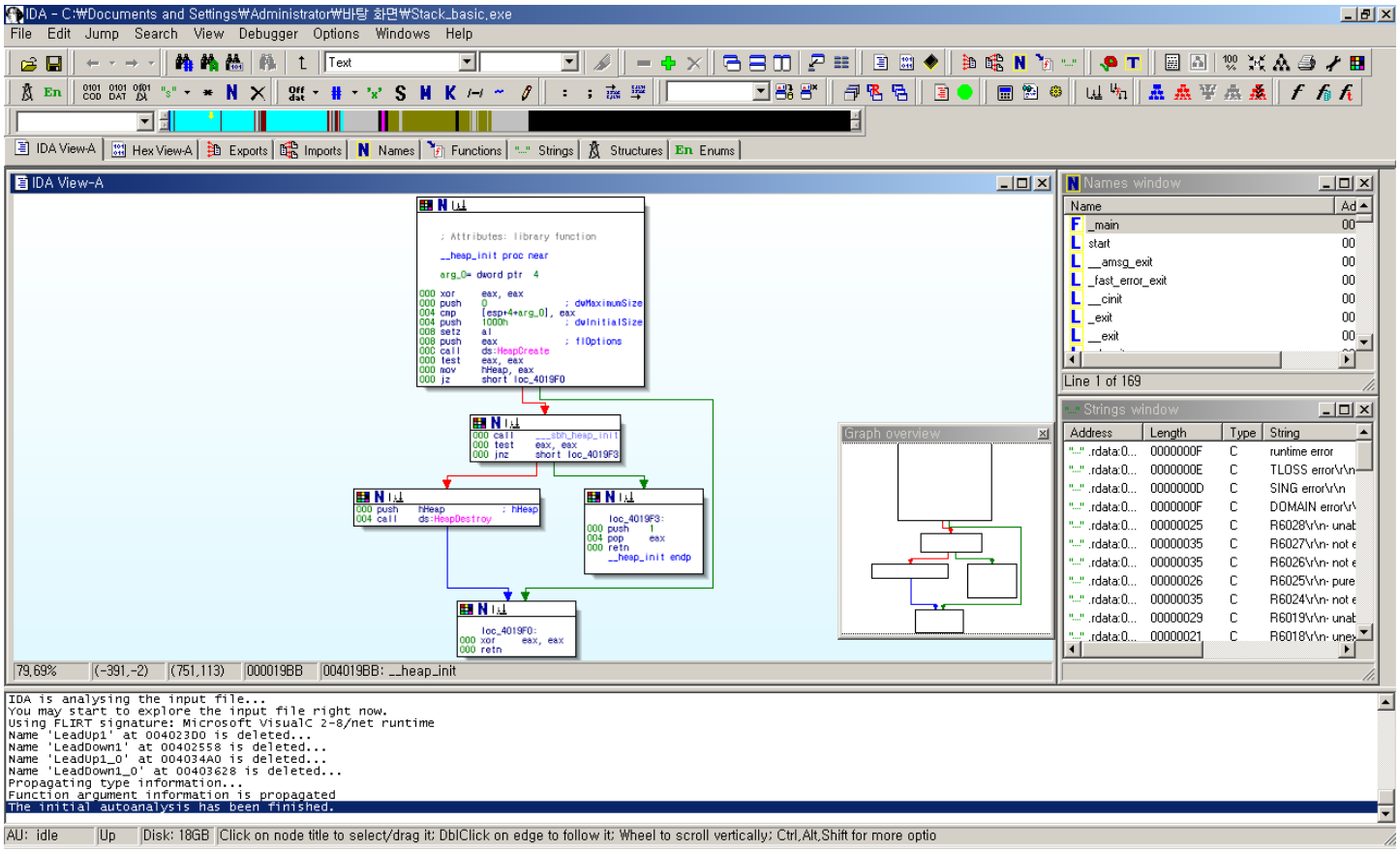
이제 파일을 불러옵니다.

IDA에서 파일을 Open하게 되면 분석과정을 통해 id0, id1, nam, til 파일이 생성됩니다. 종료하게 되면 id0, id1, nam, til 파일이 지워지고 idb 파일이 새로 생성이 됩니다.

idb 파일은 실행파일의 Database 파일이고, 분석과정에서 주석을 달면 idb 파일에 저장이 되는데 실행했던 파일 없이 나중에 이 idb 파일만 열면 다시 프로그램을 불러서 디어셈블 할필요 없이 디어셈블된 코드들을 볼수가 있습니다.

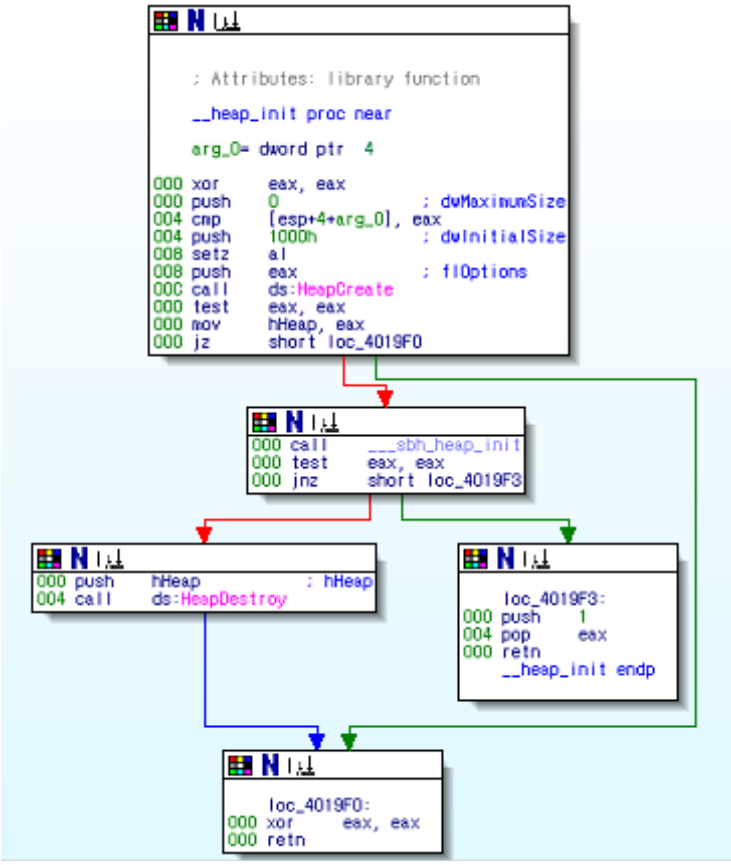
이말은 한번 프로그램을 디어셈블해서 idb이 파일이 생기면 다시 파일을 불러서 디어셈블 안해도 된다는 말이죠.

예제로 첨부한 stack_basic.exe 파일을 디어셈블 한 모습을 볼수 있습니다.



5.x부터 Graph overview 기능이 있어서 처음 디어셈블 하고 나면 바로 각 함수들의 관계와 분기점들을 Graph 화면으로 보여주는데 4.x에 비해 정말 분석하기가 편해졌다고 할수 있습니다.

Spacebar 로 디어셈블 코드들이 나오는 Text view 화면과 Layout Graph 화면으로 전환할수 있습니다.



CL Compile

이제 무엇을 해야 할까요?
(분석하기전에 잠깐 화제를 돌려 컴파일 하는법에 대해 얘기 해볼까 합니다.)

다음은 예제로 쓰인 Stack_basic 프로그램 소스입니다.
Stackbasic.c 를 컴파일해서 실행파일이 생기면 IDA로 디어셈블해서 디버깅을 할 것입니다.

Stack_basic.c

```
1   func(int a, int b, int c)
2   {
3       return (a+b+c);
4   }
5
6   int main(void)
7   {
8       int c = func(1,2,3);
9       return c;
10  }
11
```

그런데 보통 컴파일을 할때 Visual Studio 6.0 으로 컴파일을 합니다.

다음은 Visual Studio 6.0에서 컴파일한 Stackbasic_VC.exe 파일정보입니다.

크기:	168KB (172,080 바이트)
디스크 할당 크기:	172KB (176,128 바이트)

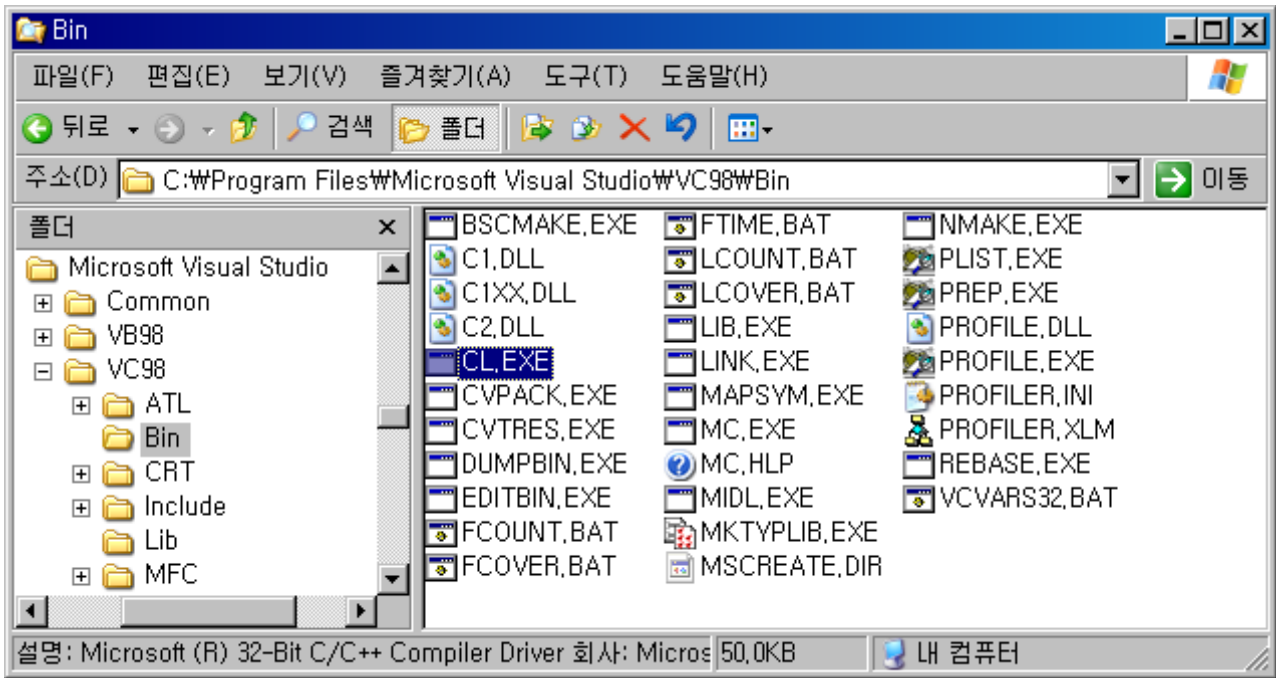
문제는 Visual Studio 로 컴파일을 하게 되면 디버깅에 관련된 코드들도 함께 디어셈블 되기 때문에 불필요한 코드들이 많이 보이게 됩니다. (물론 컴파일 할때 옵션을 설정해서 할 수는 있습니다.)

```
.text:00401060     ; =============== S U B R O U T I N E =======================================
.text:00401060
.text:00401060     ; Attributes: bp-based frame
.text:00401060
.text:00401060     main            proc near               ; CODE XREF: _main↑j
.text:00401060
.text:00401060     var_44          = dword ptr -44h
.text:00401060     var_4           = dword ptr -4
.text:00401060
.text:00401060 000                 push    ebp
.text:00401061 004                 mov     ebp, esp
.text:00401063 004                 sub     esp, 44h        ; Integer Subtraction
.text:00401066 048                 push    ebx
.text:00401067 04C                 push    esi
.text:00401068 050                 push    edi
.text:00401069 054                 lea     edi, [ebp+var_44] ; Load Effective Address
.text:0040106C 054                 mov     ecx, 11h
.text:00401071 054                 mov     eax, 0CCCCCCCCh
.text:00401076 054                 rep stosd               ; Store String
.text:00401078 054                 push    3
.text:0040107A 058                 push    2
.text:0040107C 05C                 push    1
.text:0040107E 060                 call    j_func          ; Call Procedure
.text:0040107E
.text:00401083 060                 add     esp, 0Ch        ; Add
.text:00401086 054                 mov     [ebp+var_4], eax
.text:00401089 054                 mov     eax, [ebp+var_4]
.text:0040108C 054                 pop     edi
.text:0040108D 050                 pop     esi
.text:0040108E 04C                 pop     ebx
.text:0040108F 048                 add     esp, 44h        ; Add
.text:00401092 004                 cmp     ebp, esp        ; Compare Two Operands
.text:00401094 004                 call    __chkesp        ; Call Procedure
.text:00401094
.text:00401099 004                 mov     esp, ebp
.text:0040109B 004                 pop     ebp
.text:0040109C 000                 retn                    ; Return Near from Procedure
.text:0040109C
.text:0040109C     main            endp
.text:0040109C
.text:0040109C     ; ---------------------------------------------------------------------------
```

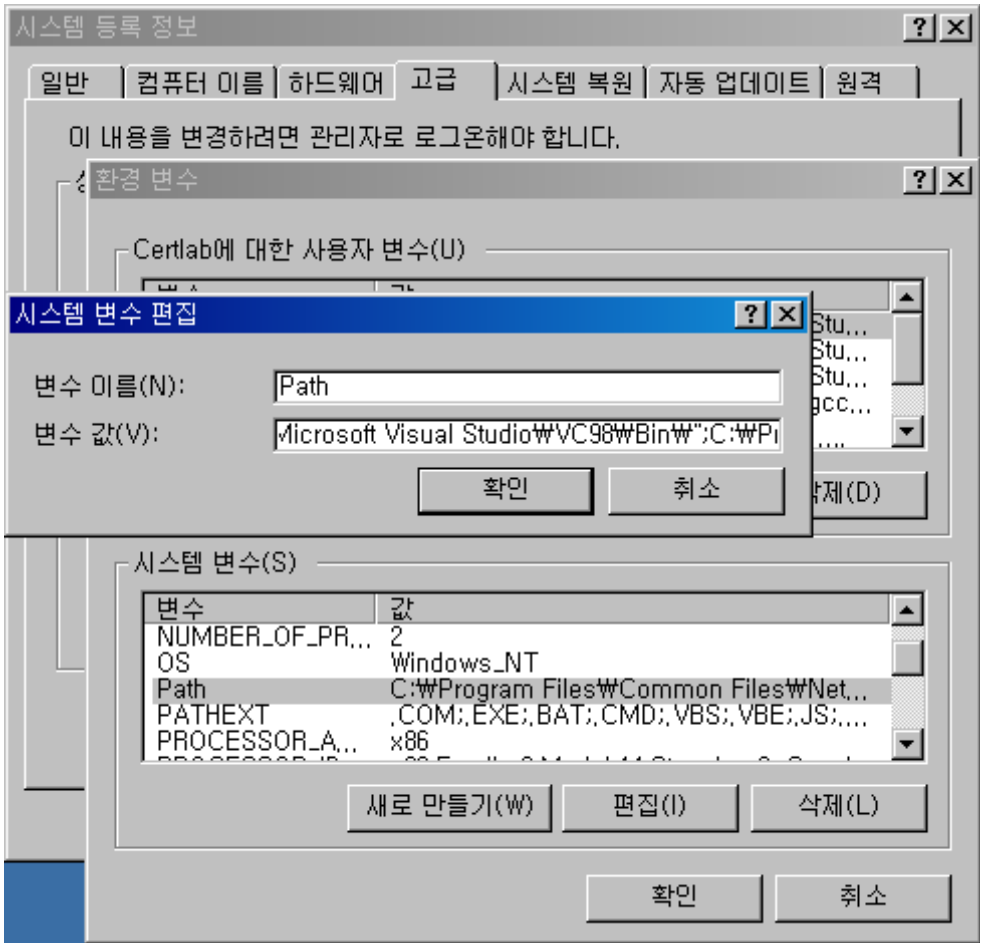
이 문서를 보시는 분들중에 IDA에 대해 잘 모르시고, 어셈블리어도 처음 접하시는분들도 계실것입니다. 처음부터 이렇게 필요없는 코드들이 섞인 디어셈블된 코드들을 분석하기는 쉽지가 않습니다.

쉽게 분석할수 있는 좋은 방법이 없을까요?

Visual Studio 안에 포함되어 있는 CL 을 이용해서 컴파일을 해보도록 하겠습니다.



컴파일을 하기전에 Path에 CL이 있는 경로를 넣어줘야 됩니다.



이제 CL 로 컴파일을 해보겠습니다.
CL의 세부옵션은 CL /help 라고 입력하면 많은 옵션들을 볼수 있구요.

Stackbasic.c 소스를 다음과 같이 컴파일을 합니다. 그럼 obj 파일과 exe 파일이 생깁니다.

```
[root@D:₩Code₩C]#cl Stack_basic.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8804 for 80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.

Stack_basic.c
Microsoft (R) Incremental Linker Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/out:Stack_basic.exe
Stack_basic.obj
```

여기서 비주얼 스튜디오로 컴파일한 파일과 비교해보겠습니다.
일단 파일 사이즈부터 틀린것을 한눈에 알수 있는데.. 디어셈블하면 뭐가 틀릴까요?

Visual Studio Compile	CL Compile
Stackbasic_VC.exe	Stack_basic_CL.exe
크기: 168KB (172,080 바이트) 디스크 할당 크기: 172KB (176,128 바이트)	크기: 36.0KB (36,864 바이트) 디스크 할당 크기: 36.0KB (36,864 바이트)

이제 IDA에서 방금 CL로 컴파일한 Stack_basic.exe 파일을 열어보겠습니다.
한눈에 봐도 불필요한 코드들은 없고, 한마디로 분석하기가 쉽게 되었습니다.

처음 어셈블리어를 공부할 때나 디버깅할때 등등 이런식으로 컴파일해서 분석 하면 편하지만.
항상 모든 소스들을 이렇게 CL로 컴파일을 해서 보는것은 좋은생각은 아닙니다.

```
.text:0040100E     ; =============== S U B R O U T I N E =======================================
.text:0040100E
.text:0040100E     ; Attributes: bp-based frame
.text:0040100E
.text:0040100E     ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:0040100E     _main           proc near               ; CODE XREF: start+AF↓p
.text:0040100E
.text:0040100E     var_4           = dword ptr -4
.text:0040100E     argc            = dword ptr  8
.text:0040100E     argv            = dword ptr  0Ch
.text:0040100E     envp            = dword ptr  10h
.text:0040100E
.text:0040100E 000                 push    ebp
.text:0040100F 004                 mov     ebp, esp
.text:00401011 004                 push    ecx
.text:00401012 008                 push    3
.text:00401014 00C                 push    2
.text:00401016 010                 push    1
.text:00401018 014                 call    sub_401000
.text:0040101D 014                 add     esp, 0Ch
.text:00401020 008                 mov     [ebp+var_4], eax
.text:00401023 008                 mov     eax, [ebp+var_4]
.text:00401026 008                 mov     esp, ebp
.text:00401028 004                 pop     ebp
.text:00401029 000                 retn
.text:00401029     _main           endp
.text:00401029
```

실제 Virus & Worm & 취약점 등을 분석 할때 CL로 컴파일 한것과 같은, 아니면 이렇게 보기 편하게 디어셈블 되는것이 아니기 때문이죠. Packing 되어있으면 Unpacking 하고, 하고 나서도 분석하기가 그렇게 쉽지는 않습니다. 실행파일 흐름 중간 중간에 바이러스의 Function 들이 있을수도 있구요.

어떤일에 있어서나 시작이 정말 중요합니다.
다음 장에서 부터 다룰 예제 프로그램인 Stackbasic 프로그램은 정말 간단한 프로그램입니다.
예제를 보고, 자신이 직접 만든 코드 들을 디어셈블&디버깅을 하면서 차근차근 공부하는 것을 추천합니다.

공부는 언제까지나 자기 스스로가 하는것입니다.

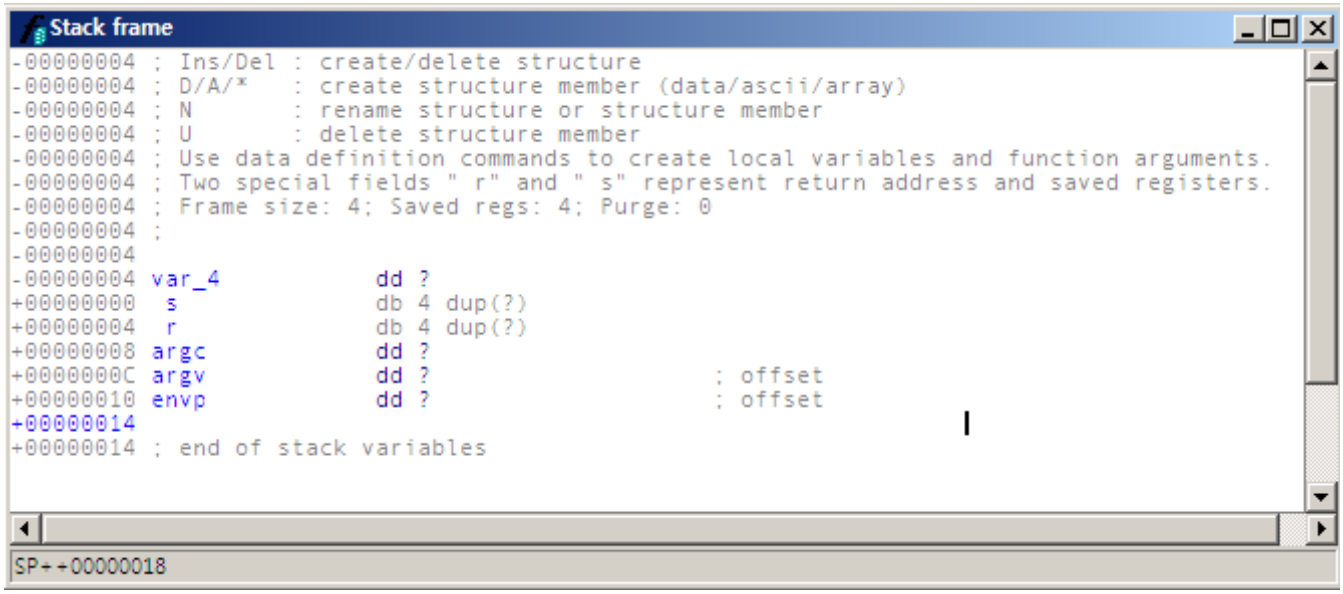
IDA 기본구성

다음 화면을 보니 IDA로 디어셈블된 코드들이 보입니다.
그런데 왜 하필 수많은 코드들중에 이 부분부터 보여지는것일까요?
이 부분이 프로그램의 시작점으로써 프로그램이 시작되면 이 부분부터 시작이 되는것입니다.

```
.text:0040100E     ; =============== S U B R O U T I N E =======================================
.text:0040100E
.text:0040100E     ; Attributes: bp-based frame
.text:0040100E
.text:0040100E     ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:0040100E     _main           proc near               ; CODE XREF: start+AF↓p
.text:0040100E
.text:0040100E     var_4           = dword ptr -4     // 선언한 지역변수
.text:0040100E     argc            = dword ptr  8     // argc, argv, envp 매개변수
.text:0040100E     argv            = dword ptr  0Ch
.text:0040100E     envp            = dword ptr  10h
.text:0040100E
.text:0040100E 000                 push    ebp
.text:0040100F 004                 mov     ebp, esp
.text:00401011 004                 push    ecx
.text:00401012 008                 push    3
.text:00401014 00C                 push    2
.text:00401016 010                 push    1
.text:00401018 014                 call    sub_401000
.text:0040101D 014                 add     esp, 0Ch
.text:00401020 008                 mov     [ebp+var_4], eax
.text:00401023 008                 mov     eax, [ebp+var_4]
.text:00401026 008                 mov     esp, ebp
.text:00401028 004                 pop     ebp
.text:00401029 000                 retn
.text:00401029     _main           endp
.text:00401029
```

여기서 각 변수들의 위치를 보려면 빨간박스안에 있는 var_c4, argc, argv, envp 부분을 더블클릭을 합니다.

그럼 다음화면과 같이 stackframe 이 나타나게 됩니다.



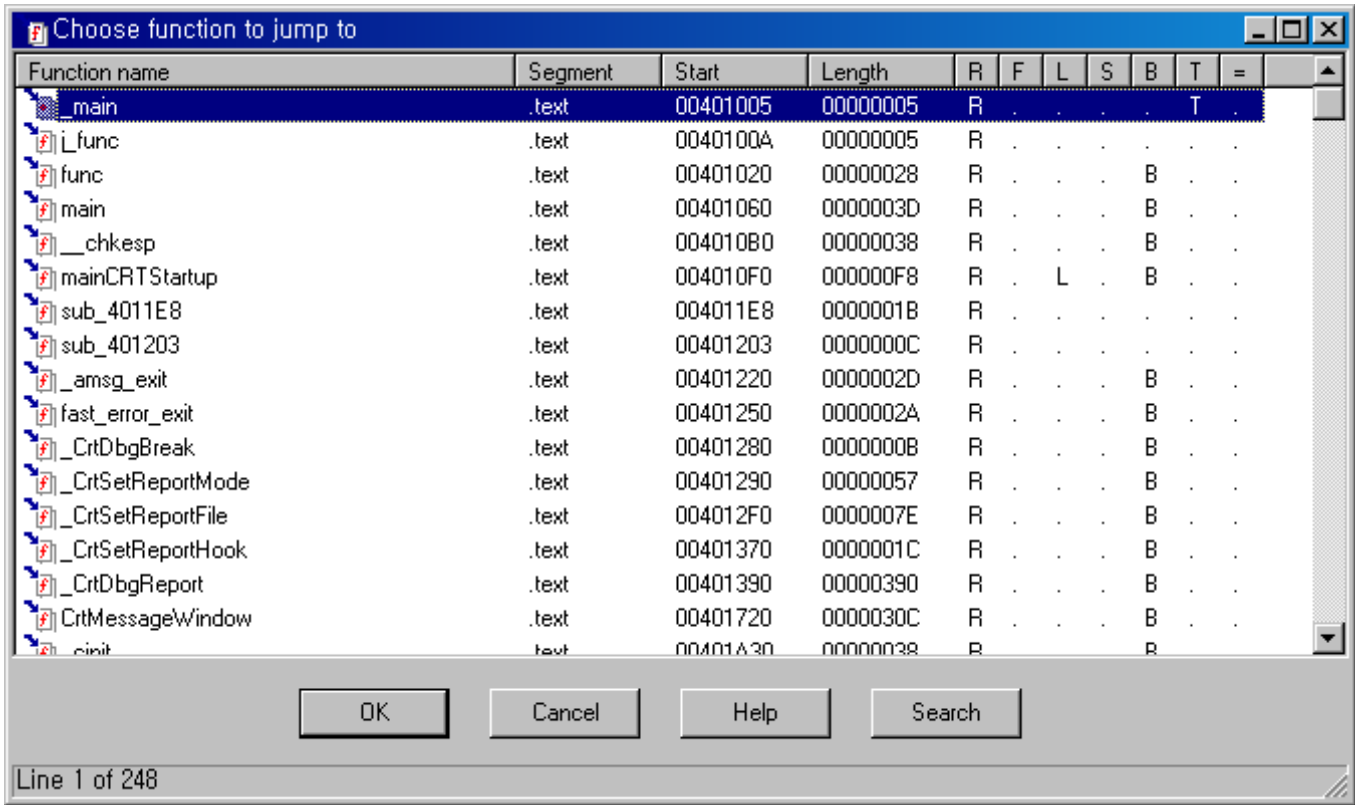
참고로 IDA에서 스크롤를 맨 위로 올려보면 다음과 같은 정보가 나옵니다.

```
Input MD5   : D97847B5E0651E6BAEB7FBC5CE179AFC

File Name   : C:\Documents and Settings\Certlab\Desktop\Stack_basic_CL.exe
Format      : Portable executable for 80386 (PE)
Imagebase   : 400000
Section 1. (virtual address 00001000)
Virtual size                  : 0000352E (  13614.)
Section size in file          : 00004000 (  16384.)
Offset to raw data for section: 00001000
Flags 60000020: Text Executable Readable
Alignment     : default
OS type          :  MS Windows
Application type:  Executable 32bit

                .686p
                .mmx
                .model flat
```

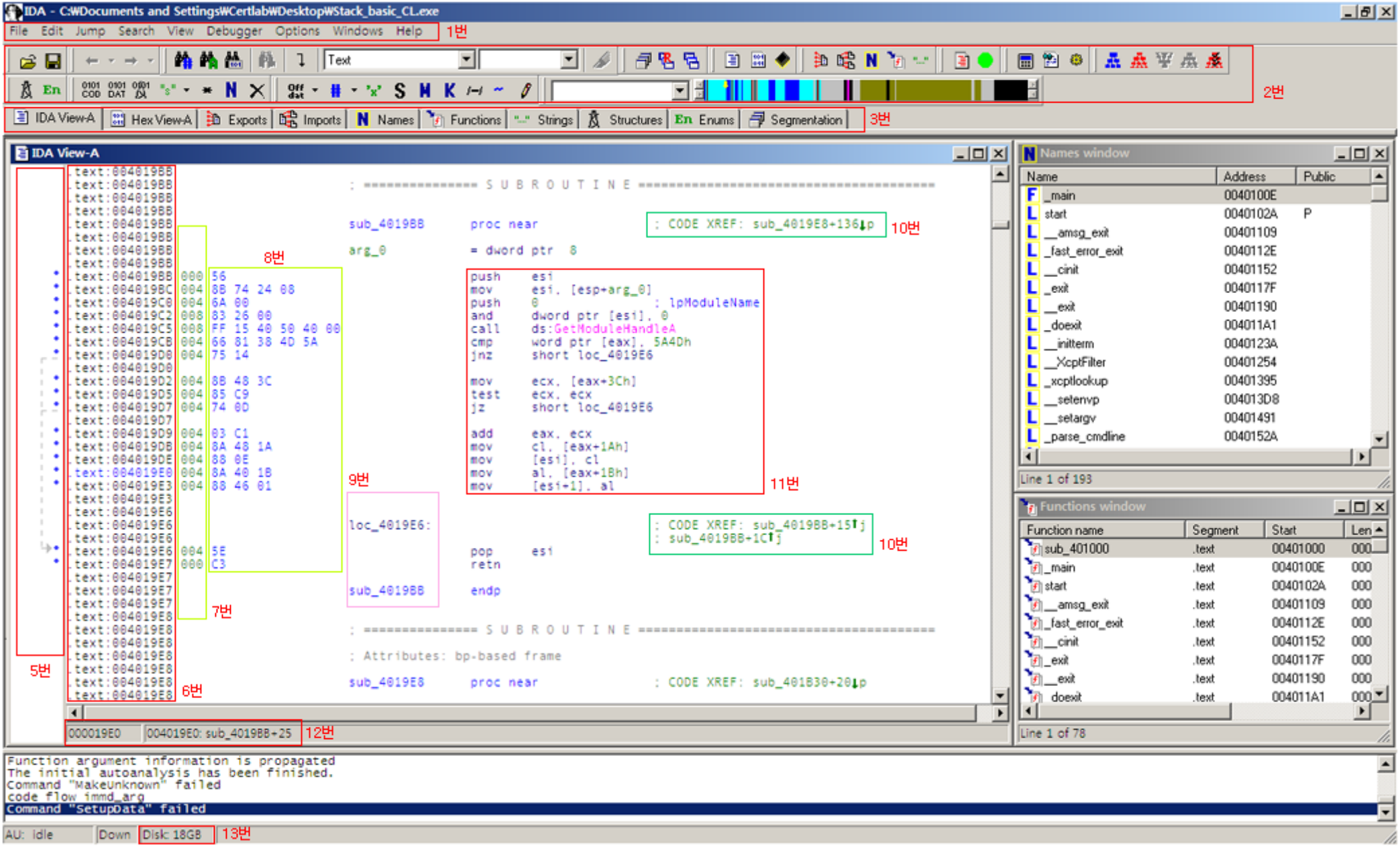
여기서 다음의 정보들을 얻을수 있습니다.

Open한 파일의 MD5정보 / Open한 파일의 절대경로 해당 File의 Format 정보 / Imagebase 정보 Section의 Size / OS정보 등등

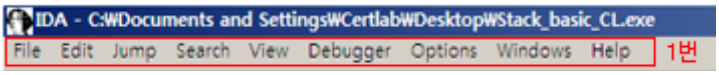
디어셈블 Code view 상태에서 Ctrl+p를 누르면 다음과 같은 화면이 나옵니다.
프로그램의 실행의 어디에 있든 Ctrp+p를 눌러서 _main 을 누르면 프로그램의 시작인 main 함수로 가게 되고, 마찬가지로 다른 해당함수로 가고 싶으면 이런식으로 이동하면 됩니다.



이제 전체적인 IDA 부분 부분을 보도록 하겠습니다. 다음 화면은 IDA의 전체화면입니다.



1번



IDA 의 메뉴바입니다. IDA의 모든 기능들이 이 메뉴 안에 다 있습니다.

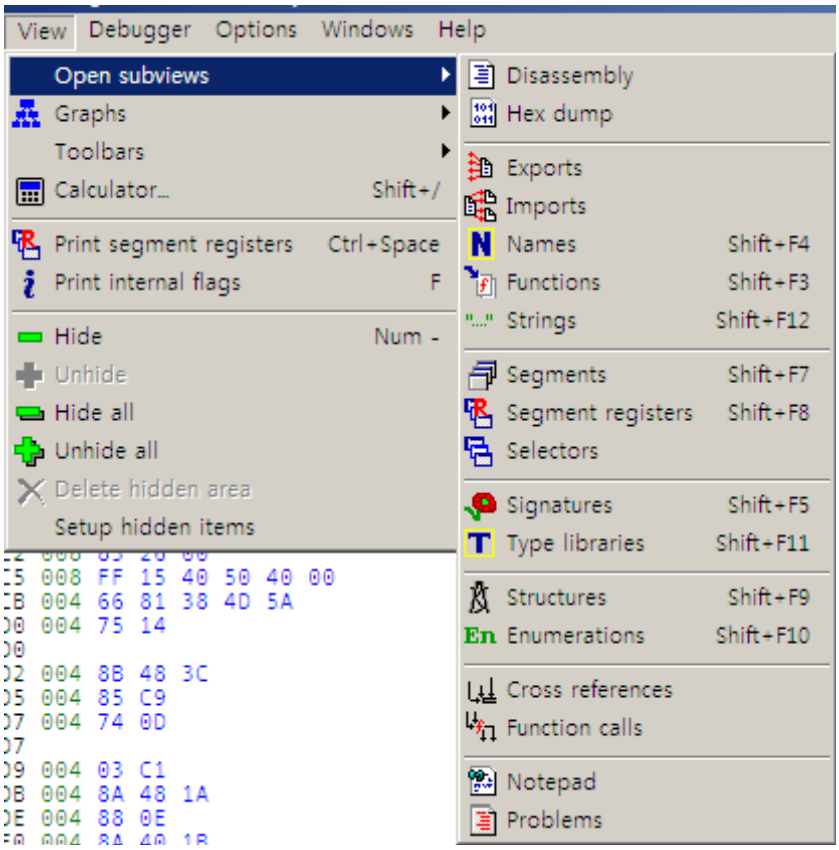
2번

IDA의 옵션들의 icon 모음인 툴바이고, 사용자의 취향에 맞게 추가/삭제를 할 수 있습니다.
보시는 화면은 주로 쓰는 옵션만 추가한 모습입니다.
icon을 추가하는 방법은 툴바의 빈공간에 마우스 우클릭을 통해 추가를 할수 있으며,
삭제할때는 현재 보이시는 아이콘 그룹의 왼쪽에 세로바가 있습니다.
그곳을 마우스 클릭->Drag해서 close 하시면 됩니다.

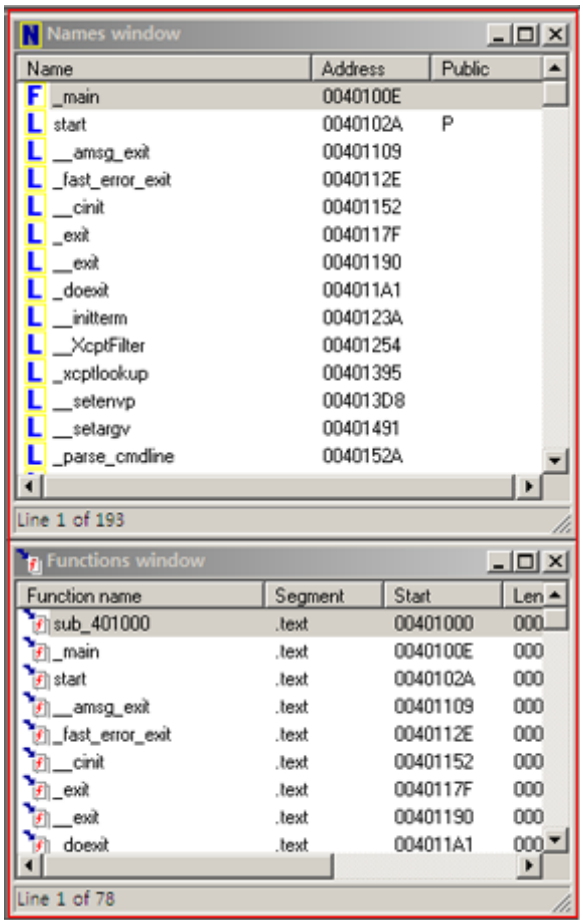
3번

IDA의 Subview입니다. 기본 디어셈블된 화면은 IDA View-A 이고,
추가로 디어셈블된 창을 또 하나 열고 싶으면 메뉴바에 View-> Open Subviews -> Disassembly
를 눌르면 IDA View-B 이런식으로 또하나의 창이 새로 열리게 됩니다.
마찬가지로 Hex View-A 도 창을 또 열수가 있습니다.

이런식으로 보고 싶은 Subview 를 아래 화면에서 선택하면 추가가 됩니다.

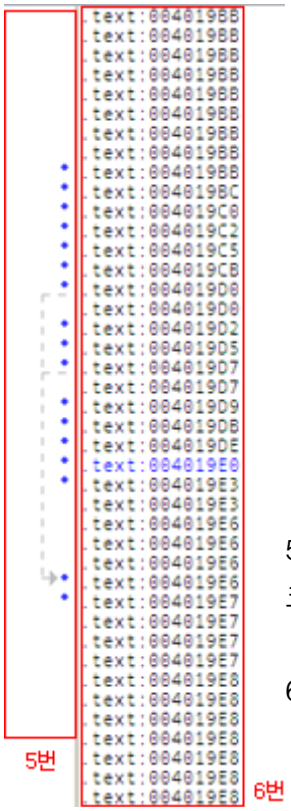


4번



분석할 때 가독성을 위해 자주 사용하는 Subview를 오른쪽에 배치합니다.
꼭 이렇게 배치를 안해도 됩니다.

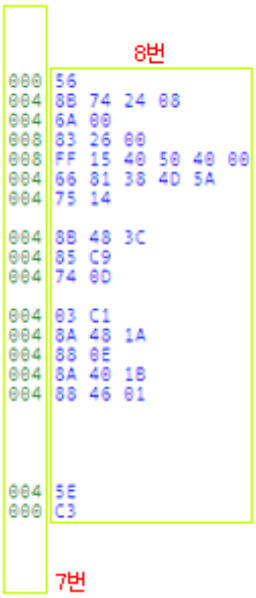
5, 6번



5번은 함수의 분기점이나 코드에서 Jump가 있으면 그곳으로 방향으로 표시해줍니다.
코드가 어디로 분기되는지 어디로 Jump되는지 알수가 있습니다.

6번은 Vitual Address를 나타냅니다.

7번, 8번

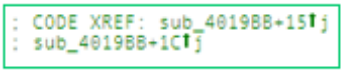


7번은 Stack Pointer입니다.
8번은 해당 어셈블리어 명령의 OPCode를 나타냅니다.

9번

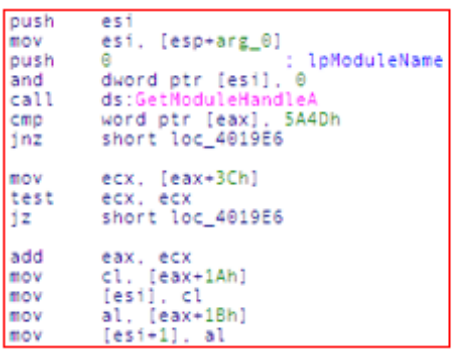
Code Location 부분입니다. 코드의 흐름중에 `mov ecx, [eax+3Ch]` `test ecx, ecx` `jz short loc_4019E6` 와 같이 분기점이 발생했을때 그 해당되는 location 으로 분기가 됩니다.

10번



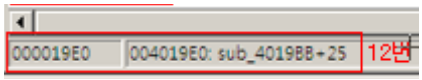
Code Reference 부분입니다. 더블클릭을 하면 현재 location 부분에서 분기점이 발생된 부분으로 이동하게 됩니다. 즉, 호출을 한곳으로 이동이 되고, ESC 를 누르면 다시 전 화면으로 이동이 됩니다.

11번



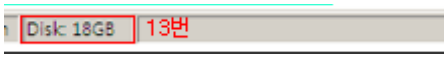
Open한 프로그램의 디어셈블된 코드입니다.

12번



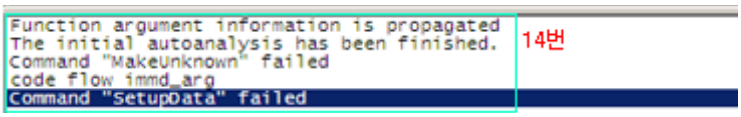
현재 마우스나 키보드로 선택된 위치의 디어셈블된 코드의 위치정보를 나타냅니다.
왼쪽은 Vitual Address 를 나타내고, 오른쪽은 Virtual Address와 현재 location의 시작된 곳에서의 offset를 나타냅니다.

13번



현재 하드디스크의 남은 용량을 반올림해서 나타냅니다. 18.4GB -〉 18GB

14번



IDA의 마지막 명령이나 정보들을 Log형태로 보여주는 곳입니다.

15번

IDA의 현재 작동상태를 나타냅니다.

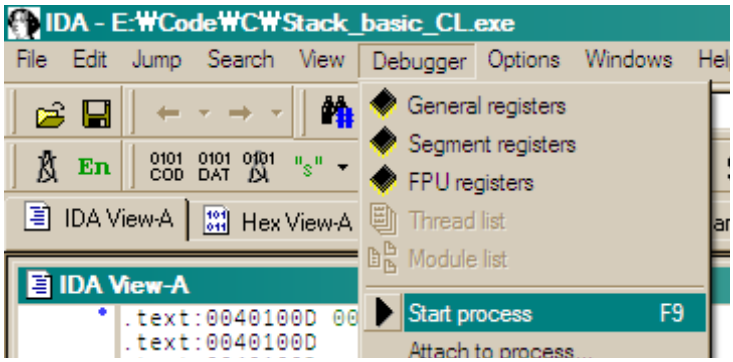
Start Debugging

이제부터 Debugging을 하면서 Stack의 변화에 대해 알아보고자 합니다.
그대로 따라하셔도 되고, 그냥 눈으로 봐도 한눈에 알수 있도록 스냅샷을 많이 찍었습니다.

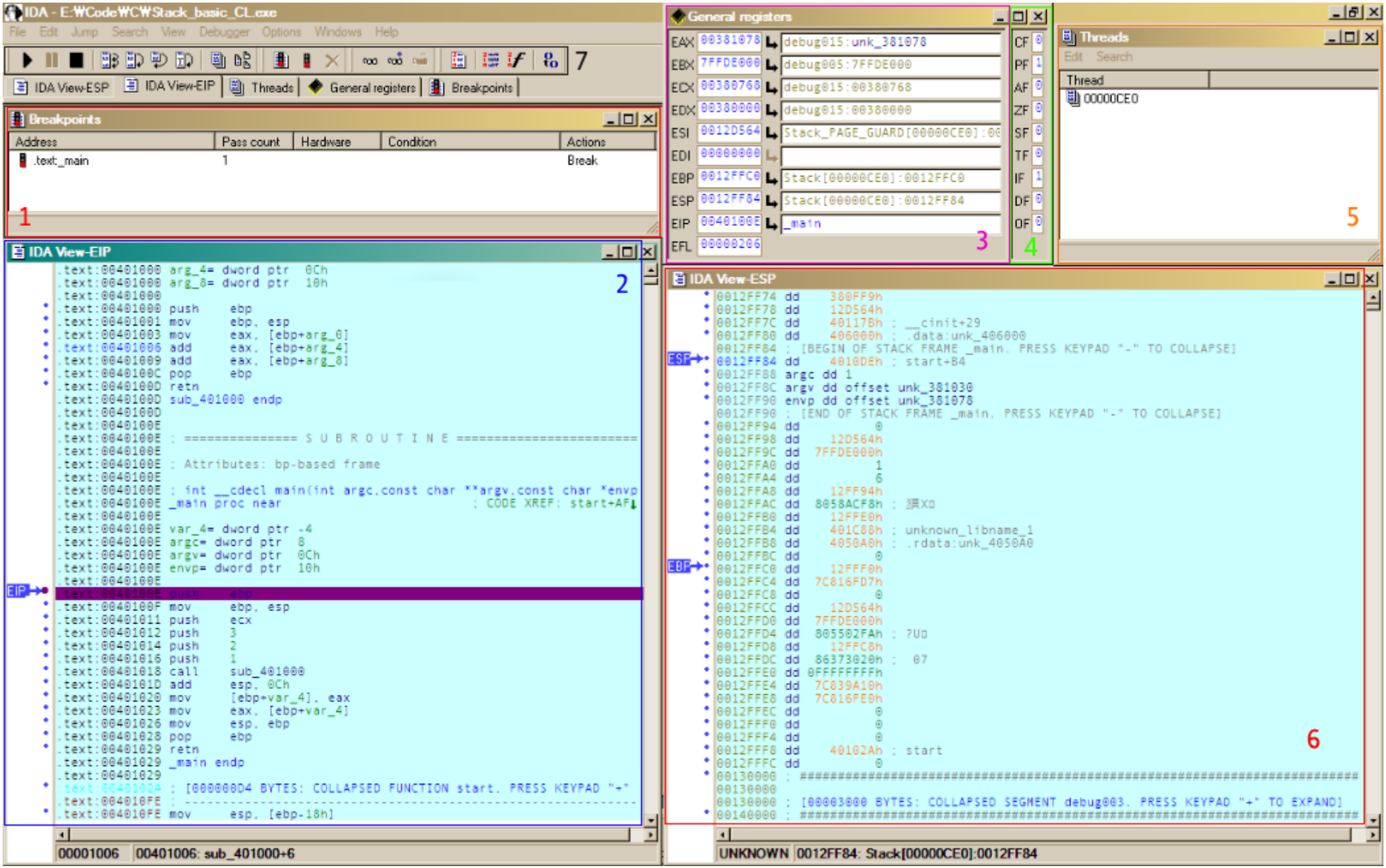
CL 로 컴파일한 stackbasic_cl.exe 파일을 Debugging을 해보도록 합시다.
아래 화면 "push ebp" 지점에 F2를 누르면 다음과 같이 빨간색 라인이 생기면서 Breakpoint가 걸리게 됩니다.

```
.text:0040100E                  ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:0040100E                  _main           proc near               ; CODE XREF: start+AF↓p
.text:0040100E
.text:0040100E                  var_4           = dword ptr -4
.text:0040100E                  argc            = dword ptr  8
.text:0040100E                  argv            = dword ptr  0Ch
.text:0040100E                  envp            = dword ptr  10h
.text:0040100E
.text:0040100E 000 55                           push    ebp
.text:0040100F 004 8B EC                        mov     ebp, esp
.text:00401011 004 51                           push    ecx
.text:00401012 008 6A 03                        push    3
.text:00401014 00C 6A 02                        push    2
.text:00401016 010 6A 01                        push    1
.text:00401018 014 E8 E3 FF FF+                 call    sub_401000
.text:0040101D 014 83 C4 0C                     add     esp, 0Ch
.text:00401020 008 89 45 FC                     mov     [ebp+var_4], eax
.text:00401023 008 8B 45 FC                     mov     eax, [ebp+var_4]
.text:00401026 008 8B E5                        mov     esp, ebp
.text:00401028 004 5D                           pop     ebp
.text:00401029 000 C3                           retn
```

이제 Debugger –〉 Start process를 누르거나 F9를 누르게 되면 이제 Debugging 화면으로 바뀌게 됩니다.

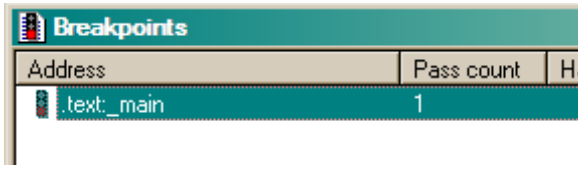


다음 화면은 IDA Debugging 화면이고,
아까 Breakpoint를 건 지점(push ebp)에 멈춰져 있는것을 볼수 있습니다.


IDA - E:₩Code₩C₩Stack_basic_CL.exe
File Edit Jump Search View Debugger Options Windows Help
7
IDA View-ESP
IDA View-EIP
Threads
General registers
Breakpoints
Breakpoints
Address
Pass count
Hardware
Condition
Actions
.text:_main
1
Break
1
General registers
EAX 00381078 debug015:unk_381078
EBX 7FFDE000 debug005:7FFDE000
ECX 00380768 debug015:00380768
EDX 00380000 debug015:00380000
ESI 0012D564 Stack_PAGE_GUARD[00000CE0]:00
EDI 00000000
EBP 0012FFC0 Stack[00000CE0]:0012FFC0
ESP 0012FF84 Stack[00000CE0]:0012FF84
EIP 0040100E _main
EFL 00000206
3
CF 0
PF 1
AF 0
ZF 0
SF 0
TF 0
IF 1
DF 0
OF 0
4
Threads
Edit Search
Thread
00000CE0
5
IDA View-EIP
2
.text:00401000 arg_4= dword ptr 0Ch
.text:00401000 arg_8= dword ptr 10h
.text:00401000 push ebp
.text:00401001 mov ebp, esp
.text:00401003 mov eax, [ebp+arg_0]
.text:00401006 add eax, [ebp+arg_4]
.text:00401009 add eax, [ebp+arg_8]
.text:0040100C pop ebp
.text:0040100D retn
.text:0040100D sub_401000 endp
.text:0040100E ; ========== S U B R O U T I N E ==========
.text:0040100E ; Attributes: bp-based frame
.text:0040100E ; int __cdecl main(int argc,const char **argv,const char *envp
.text:0040100E _main proc near ; CODE XREF: start+AF↓
.text:0040100E var_4= dword ptr -4
.text:0040100E argc= dword ptr 8
.text:0040100E argv= dword ptr 0Ch
.text:0040100E envp= dword ptr 10h
EIP
.text:0040100E push ebp
.text:0040100F mov ebp, esp
.text:00401011 push ecx
.text:00401012 push 3
.text:00401014 push 2
.text:00401016 push 1
.text:00401018 call sub_401000
.text:0040101D add esp, 0Ch
.text:00401020 mov [ebp+var_4], eax
.text:00401023 mov eax, [ebp+var_4]
.text:00401026 mov esp, ebp
.text:00401028 pop ebp
.text:00401029 retn
.text:00401029 _main endp
.text:00401029
.text:0040102A ; [000000D4 BYTES: COLLAPSED FUNCTION start. PRESS KEYPAD "+"
.text:004010FE ; ---------- (----)
.text:004010FE mov esp, [ebp-18h]
00001006 00401006: sub_401000+6
IDA View-ESP
0012FF74 dd 380FF9h
0012FF78 dd 12D564h
0012FF7C dd 40117Bh ; __cinit+29
0012FF80 dd 406000h ; .data:unk_406000
0012FF84 ; [BEGIN OF STACK FRAME _main, PRESS KEYPAD "-" TO COLLAPSE]
ESP 0012FF84 dd 4010DEh ; start+B4
0012FF88 argc dd 1
0012FF8C argv dd offset unk_381030
0012FF90 envp dd offset unk_381078
0012FF90 ; [END OF STACK FRAME _main, PRESS KEYPAD "-" TO COLLAPSE]
0012FF94 dd 0
0012FF98 dd 12D564h
0012FF9C dd 7FFDE000h
0012FFA0 dd 1
0012FFA4 dd 6
0012FFA8 dd 12FF94h
0012FFAC dd 8058ACF8h ; 頂X□
0012FFB0 dd 12FFE0h
0012FFB4 dd 401C88h ; unknown_libname_1
0012FFB8 dd 4050A0h ; .rdata:unk_4050A0
0012FFBC dd 0
EBP 0012FFC0 dd 12FFF0h
0012FFC4 dd 7C816FD7h
0012FFC8 dd 0
0012FFCC dd 12D564h
0012FFD0 dd 7FFDE000h
0012FFD4 dd 805502FAh ; ?U□
0012FFD8 dd 12FFC8h
0012FFDC dd 86373020h ; 07
0012FFE0 dd 0FFFFFFFFh
0012FFE4 dd 7C839A10h
0012FFE8 dd 7C816FE0h
0012FFEC dd 0
0012FFF0 dd 0
0012FFF4 dd 0
0012FFF8 dd 40102Ah ; start
0012FFFC dd 0
00130000 ; ##########
00130000
00130000 ; [00003000 BYTES: COLLAPSED SEGMENT debug003. PRESS KEYPAD "+" TO EXPAND]
00140000 ; ##########
6
UNKNOWN 0012FF84: Stack[00000CE0]:0012FF84

1번 : Breakpoints

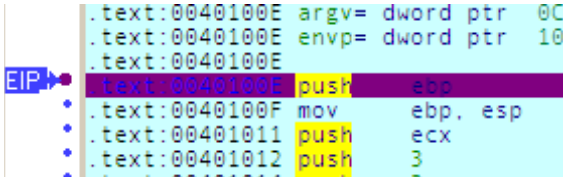
// 현재 설정된 Breakpoints 목록을 보여줍니다.



2번 : IDA View-EIP

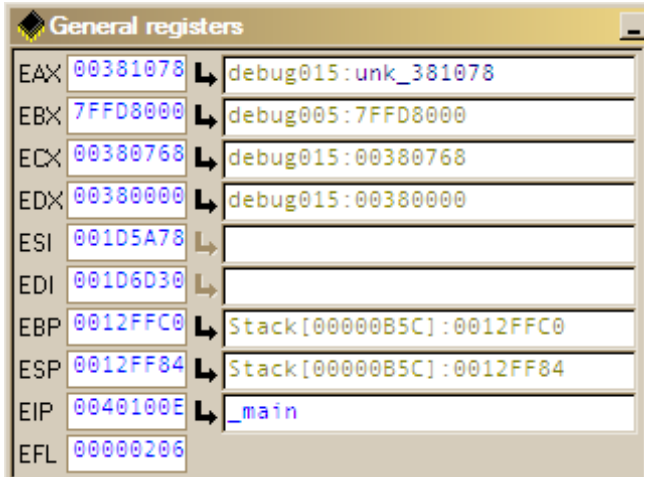
// 프로그램의 Code 부분. EIP는 다음실행할 주소를 가리킵니다.

// 여기서는 EIP가 가리키는 부분을 유심히 봅시다.



3번 : Registers

// Registers 상태를 보여줍니다. 자세한 내용은 36Page를 참고해주세요.

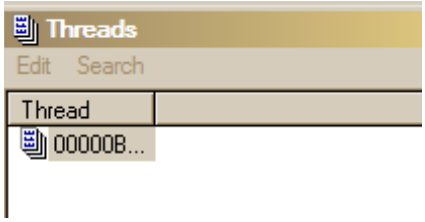


4번 : Flags

// Flags 상태를 보여줍니다. 자세한 내용은 36Page를 참고해주세요.

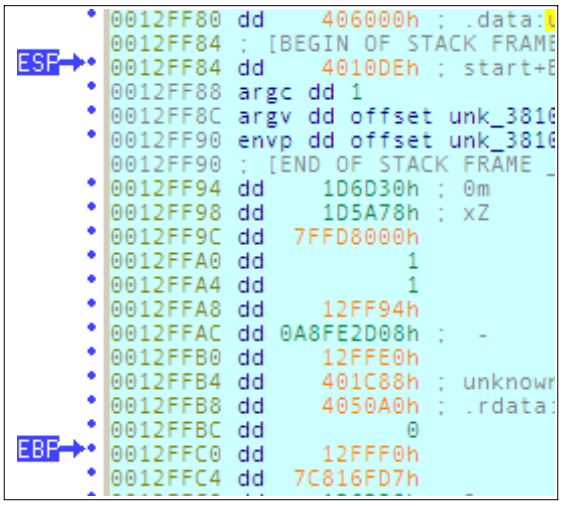
5번 : Threads

// Threads 목록



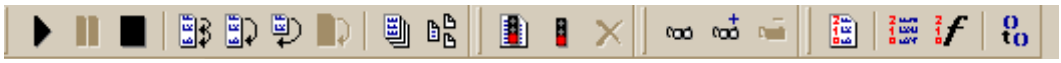
6번 : IDA View-ESP

// 현재 Stack의 흐름을 보여주는 곳입니다. 여기서 ESP와 EBP가 가리키는 곳의 주소와
// 가리키는 값을 유심히 보시기 바랍니다.



7번 : Menubar

// Debugging에 필요한 각종 유용한 기능들을 버튼으로 모아놓았습니다.



다음은 Debugging에 필요한 단축키 설명입니다.

F9 : Debugging 시작
F7 : Step into // 함수 안으로 들어가면서 실행
F8 : Step Over // 함수도 한줄로 인식하여 한줄씩 실행
Ctrl+F2 : Debugging 종료
Shift+/ : 계산기

자 이제 Debugging을 시작 합시다.

다음 코드가 스택에서 일어나는 일들을 자세히 보도록 하겠습니다.

```
.text:0040100E push    ebp
.text:0040100F mov     ebp, esp
.text:00401011 push    ecx
.text:00401012 push    3
.text:00401014 push    2
.text:00401016 push    1
.text:00401018 call    sub_401000
.text:0040101D add     esp, 0Ch
.text:00401020 mov     [ebp+var_4], eax
.text:00401023 mov     eax, [ebp+var_4]
.text:00401026 mov     esp, ebp
.text:00401028 pop     ebp
.text:00401029 retn
```

```
#include <stdio.h>

func(int a, int b, int c)
{
        return (a+b+c);
}

int main(void)
{
        int c = func(1,2,3);
        return c;
}
```

다음 화면은 처음 push ebp 부분을 실행하기전의 Stack의 상태입니다.
ESP, EBP 각각의 위치를 눈여겨 봅시다.

```
ESP→ 0012FF84 dd    4010DEh ; start+B4
     0012FF88 argc dd 1
     0012FF8C argv dd offset unk_381030
     0012FF90 envp dd offset unk_381078
     0012FF90 ; [END OF STACK FRAME _main. PRESS KEYPA
     0012FF94 dd          3
     0012FF98 dd    1F1F40h ; @
     0012FF9C dd  7FFDE000h
     0012FFA0 dd          1
     0012FFA4 dd          1
     0012FFA8 dd    12FF94h
     0012FFAC dd 0A866ED08h ;  ?
     0012FFB0 dd    12FFE0h
     0012FFB4 dd    401C88h ; unknown_libname_1
     0012FFB8 dd    4050A0h ; .rdata:unk_4050A0
     0012FFBC dd          0
EBP→ 0012FFC0 dd    12FFF0h
     0012FFC4 dd  7C816FD7h
     0012FFC8 dd          3
```

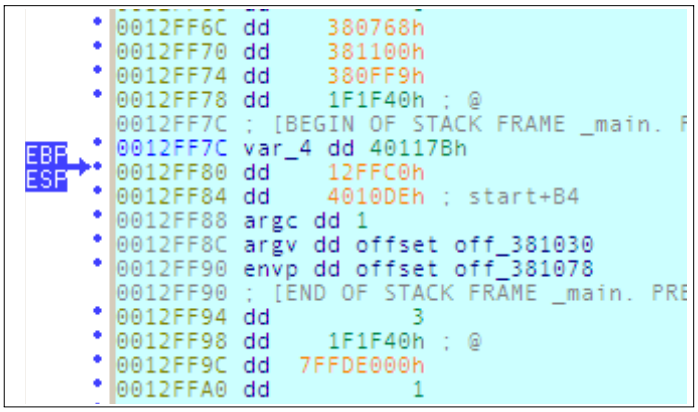
.text:0040100E push   ebp
// push ebp를 하면 다음 화면과 같이 EBP가 있는자리가 SFP로 변하고,
// 그 바로 밑에가 RET로 바뀌게 되고, ESP는 다음주소를 가르키는것을 볼수 있습니다.

```
     0012FF80 ; [BEGIN OF STACK FRAME _main. PRESS KEYP
ESP→ 0012FF80 dd    12FFC0h ; Stack[0000062C]:saved_fp
     0012FF84 dd    4010DEh ; start+B4
     0012FF88 argc dd 1
     0012FF8C argv dd offset off_381030
     0012FF90 envp dd offset off_381078
     0012FF90 ; [END OF STACK FRAME _main. PRESS KEYPAD
     0012FF94 dd          3
     0012FF98 dd    1F1F40h ; @
     0012FF9C dd  7FFDE000h
     0012FFA0 dd          1
     0012FFA4 dd          1
     0012FFA8 dd    12FF94h
     0012FFAC dd 0A866ED08h ;  ?
     0012FFB0 dd    12FFE0h
     0012FFB4 dd    401C88h ; unknown_libname_1
     0012FFB8 dd    4050A0h ; .rdata:unk_4050A0
     0012FFBC dd          0
EBP→ 0012FFC0 saved_fp dd    12FFF0h ; Stack[0000062C]:
     0012FFC4 retaddr dd  7C816FD7h
     0012FFC8 dd          3
     0012FFCC dd    1F1F40h ; @
     0012FFD0 dd  7FFDE000h
```

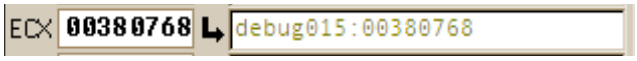
.text:0040100F mov ebp, esp

// mov ebp, esp 를 실행하면 EBP가 ESP가 있는 곳을 가리키게 되고,

// EBP와 ESP가 같은 지점을 가리키는것을 볼수 있습니다.



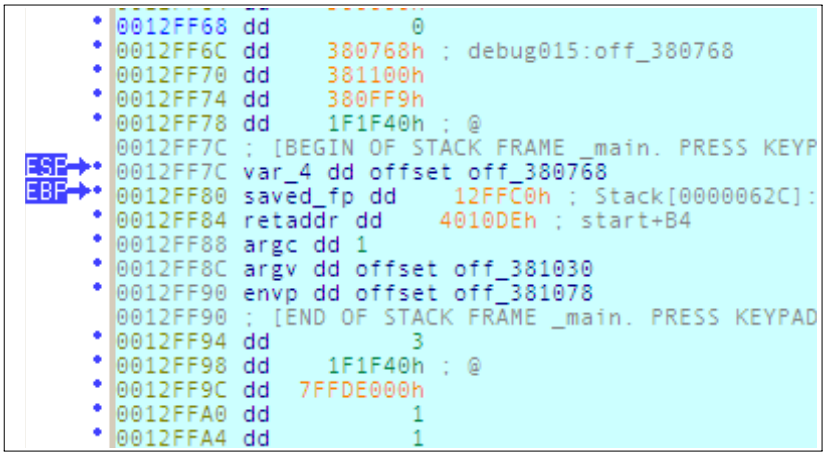
.text:00401011 push ecx



// EBP가 가리키는 곳이 SFP로 되었고, EBP 바로 밑에가 RET, argc, argv, envp 가 있는것을

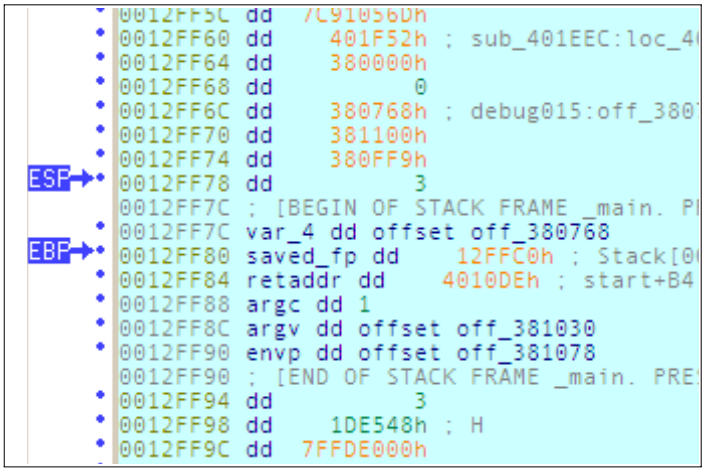
// 볼수 있습니다.

// 또한 ESP가 한칸 내려가면서 현재 ECX 값 00380768 이 Stack에 들어가는것을 볼수 있습니다.



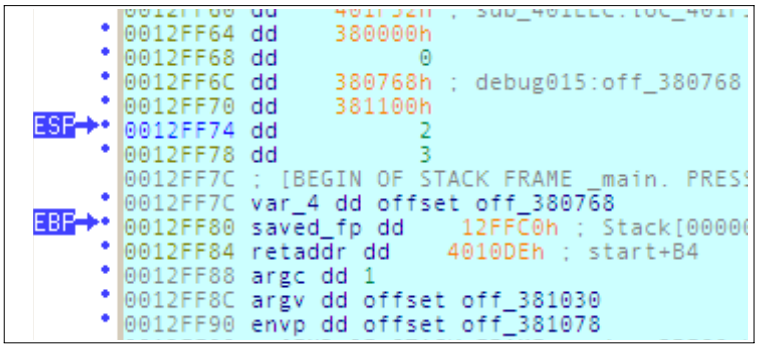
.text:00401012 push 3

// ESP가 한칸 밑으로 내려가면서 'push 3' 3값이 들어간것을 볼수 있습니다.



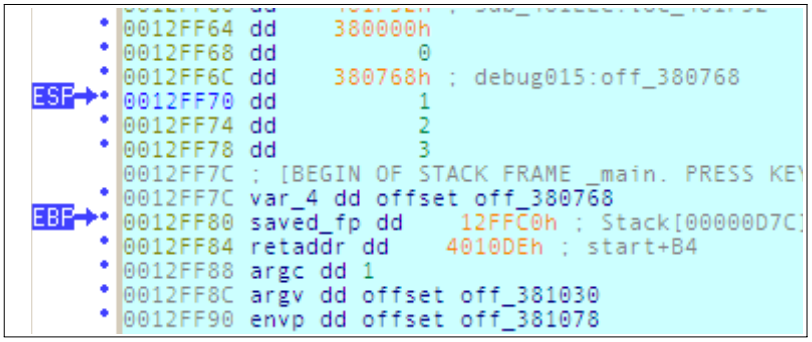
.text:00401014 push 2

// ESP가 한칸 내려가면서 'push 2' 2값이 들어간것을 볼수 있습니다.



.text:00401016 push 1

// ESP가 한칸 내려가면서 'push 1' 1값이 들어간것을 볼수 있습니다.



.text:00401018 call sub_401000

-> Step into

```
0012FF64 dd    380000h
0012FF68 dd          0
0012FF6C ; [BEGIN OF STACK FRAME sub_401000.
ESP→
0012FF6C dd    40101Dh ; _main+F
0012FF70 arg_0 dd 1
0012FF74 arg_4 dd 2
0012FF78 arg_8 dd 3
0012FF78 ; [END OF STACK FRAME sub_401000. P
0012FF7C dd    380768h ; debug015:off_380768
EBP→
0012FF80 saved_fp dd    12FFC0h ; Stack[0000
0012FF84 retaddr dd    4010DEh ; start+B4
0012FF88 argc dd 1
0012FF8C argv dd offset off_381010
0012FF90 envp dd offset off_381078
0012FF94 dd    1CC9D8h ; 芙
```

.text:00401000 push ebp

```
0012FF5C dd  7C91056Dh
0012FF60 dd    401F52h ; sub_401EEC:loc_401F52
0012FF64 dd    380000h
0012FF68 ; [BEGIN OF STACK FRAME sub_401000. PRES
ESP→
0012FF68 dd    12FF80h ; Stack[00000E5C]:saved_fp
0012FF6C dd    40101Dh ; _main+F
0012FF70 arg_0 dd 1
0012FF74 arg_4 dd 2
0012FF78 arg_8 dd 3
0012FF78 ; [END OF STACK FRAME sub_401000. PRESS
0012FF7C ; [BEGIN OF STACK FRAME _main. PRESS KEY
0012FF7C var_4 dd offset off_380768
EBP→
0012FF80 saved_fp dd    12FFC0h ; Stack[00000E5C]
0012FF84 retaddr dd    4010DEh ; start+B4
0012FF88 argc dd 1
0012FF8C argv dd offset off_381010
```

.text:00401001 mov ebp, esp

EAX 00381078 → debug015:00381078

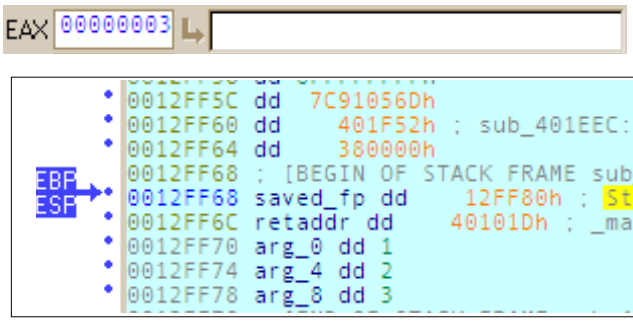
```
0012FF58 dd 0FFFFFFFFh
0012FF5C dd  7C91056Dh
0012FF60 dd    401F52h ; sub_401EEC:l
0012FF64 dd    380000h
0012FF68 ; [BEGIN OF STACK FRAME sub_
EBP
ESP→
0012FF68 dd    12FF80h
0012FF6C dd    40101Dh ; _main+F
0012FF70 arg_0 dd 1
0012FF74 arg_4 dd 2
0012FF78 arg_8 dd 3
0012FF78 ; [END OF STACK FRAME sub_40
```

.text:00401003 mov eax, [ebp+arg_0]

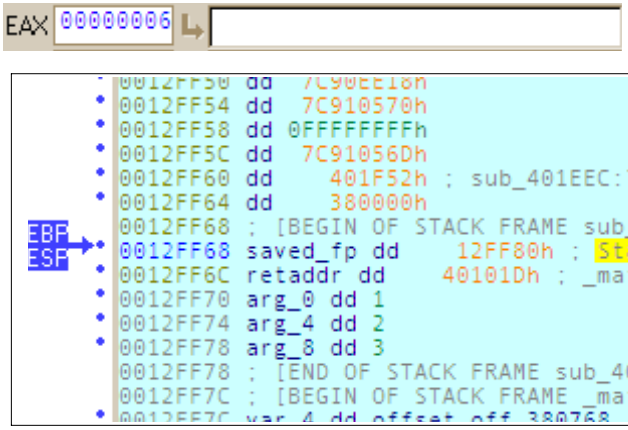
EAX 00000001

```
0012FF58 dd 0FFFFFFFFh
0012FF5C dd  7C91056Dh
0012FF60 dd    401F52h ; sub_401EEC:loc_401F5
0012FF64 dd    380000h
0012FF68 ; [BEGIN OF STACK FRAME sub_401000.
EBP
ESP→
0012FF68 saved_fp dd    12FF80h ; Stack[00000
0012FF6C retaddr dd    40101Dh ; _main+F
0012FF70 arg_0 dd 1
0012FF74 arg_4 dd 2
0012FF78 arg_8 dd 3
0012FF78 ; [END OF STACK FRAME sub_401000. PR
```

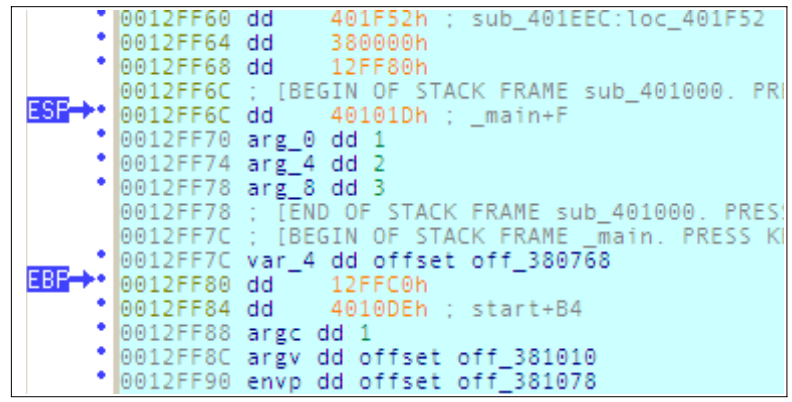
.text:00401006 add eax, [ebp+arg_4]



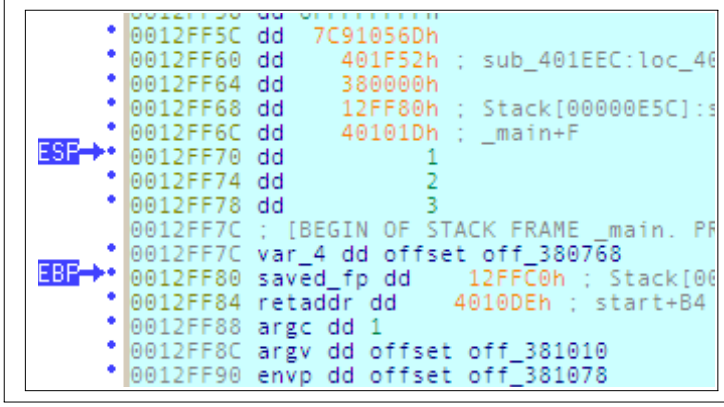
.text:00401009 add eax, [ebp+arg_8]



.text:0040100C pop ebp



.text:0040100D retn



.text:0040101D add esp, 0Ch

```
0012FF68 dd    12FF80h ; Stack[00000960
0012FF6C dd    40101Dh ; _main+F
0012FF70 dd          1
0012FF74 dd          2
0012FF78 dd          3
0012FF7C ; [BEGIN OF STACK FRAME _main.
ESP→
0012FF7C var_4 dd offset off_380768
EBP→
0012FF80 saved_fp dd    12FFC0h ; Stack
0012FF84 retaddr dd    4010DEh ; start+
0012FF88 argc dd 1
0012FF8C argv dd offset off_381010
0012FF90 envp dd offset off_381078
0012FF90 ; [END OF STACK FRAME _main. F
```

.text:00401020 mov [ebp+var_4], eax

```
0012FF70 dd          1
0012FF74 dd          2
0012FF78 dd          3
0012FF7C ; [BEGIN OF STACK FRAME _main. PRESS KEYPAD
ESP→
0012FF7C var_4 dd 6
EBP→
0012FF80 saved_fp dd    12FFC0h ; Stack[0000096C]:sa
0012FF84 retaddr dd    4010DEh ; start+B4
0012FF88 argc dd 1
0012FF8C argv dd offset off_381010
0012FF90 envp dd offset off_381078
0012FF90 ; [END OF STACK FRAME _main. PRESS KEYPAD "
0012FF94 dd    1BCE28h ; (
```

.text:00401023 mov eax, [ebp+var_4]

```
EAX 00000006
```

```
0012FF68 dd    12FF80h ; Stack[00000
0012FF6C dd    40101Dh ; _main+F
0012FF70 dd          1
0012FF74 dd          2
0012FF78 dd          3
0012FF7C ; [BEGIN OF STACK FRAME _ma
ESP→
0012FF7C var_4 dd 6
EBP→
0012FF80 saved_fp dd    12FFC0h ; St
0012FF84 retaddr dd    4010DEh ; sta
0012FF88 argc dd 1
0012FF8C argv dd offset off_381010
0012FF90 envp dd offset off_381078
0012FF90 ; [END OF STACK FRAME _main
0012FF94 dd    1BCE28h ; (
```

.text:00401026 mov esp, ebp

```
0012FF6C dd    40101Dh ; _main+F
0012FF70 dd          1
0012FF74 dd          2
0012FF78 dd          3
0012FF7C ; [BEGIN OF STACK FRAME _main
EBP→
0012FF7C var_4 dd 6
ESP→
0012FF80 saved_fp dd    12FFC0h ; Stac
0012FF84 retaddr dd    4010DEh ; start
0012FF88 argc dd 1
0012FF8C argv dd offset off_381010
0012FF90 envp dd offset off_381078
```

.text:00401028 pop ebp

```
0012FF78 dd          3
0012FF7C dd          6
0012FF80 dd    12FFC0h
0012FF84 ; [BEGIN OF STACK FRAME _main.
ESP→0012FF84 dd    4010DEh ; start+B4
0012FF88 argc dd 1
0012FF8C argv dd offset off_381010
0012FF90 envp dd offset off_381078
0012FF90 ; [END OF STACK FRAME _main. P
0012FF94 dd    1BCE28h ; (
0012FF98 dd    1DBC58h ; X
0012FF9C dd  7FFDD000h
0012FFA0 dd          1
0012FFA4 dd          1
0012FFA8 dd    12FF94h
0012FFAC dd 0A90F3D08h ;  =
0012FFB0 dd    12FFE0h
0012FFB4 dd    401C88h ; unknown_libnam
0012FFB8 dd    4050A0h ; .rdata:unk_405
0012FFBC dd          0
EBP→0012FFC0 dd    12FFF0h
0012FFC4 dd  7C816FD7h
0012FFC8 dd    1BCE28h ; (
0012FFCC dd    1DBC58h ; X
```

.text:00401029 retn

```
0012FF78 dd          3
0012FF7C dd          6
0012FF80 dd    12FFC0h ; Stack[0000096C]:s
0012FF84 dd    4010DEh ; start+B4
ESP→0012FF88 dd          1
0012FF8C dd    381010h
0012FF90 dd    381078h
0012FF94 dd    1BCE28h ; (
0012FF98 dd    1DBC58h ; X
0012FF9C dd  7FFDD000h
0012FFA0 dd          1
0012FFA4 dd          1
0012FFA8 dd    12FF94h
0012FFAC dd 0A90F3D08h ;  =
0012FFB0 dd    12FFE0h
0012FFB4 dd    401C88h ; unknown_libname_1
0012FFB8 dd    4050A0h ; .rdata:unk_4050A0
0012FFBC dd          0
EBP→0012FFC0 saved_fp dd    12FFF0h ; Stack[00
0012FFC4 retaddr dd  7C816FD7h
0012FFC8 dd    1BCE28h ; (
0012FFCC dd    1DBC58h ; X
0012FFD0 dd  7FFDD000h
```

간단한 프로그램을 통해 Stack의 흐름을 보았습니다.
이렇게 간단한 프로그램에서부터 복잡한 프로그램까지 이런식으로 디버깅을 연습하거나 공부하시면 코드의 흐름을 잘 이해할 수가 있을것입니다.

Debugging을 종료하려면 Ctrl+F2 를 누르면 됩니다.

Registers

EAX	입출력과 거의 모든 산술연산에 사용됩니다. 곱셋과 나눗셈, 변환 명령어등은 반드시 Eax 레지스터를 필요하게 되고, 함수의 반환값은 EAX에 저장이 됩니다.
EBX	일반적인 계산 용도로 쓰입니다.
ECX	루프의 반복 횟수나 좌우방향의 시프트 비트 수를 기억하고, 그외의 계산에도 사용됩니다.
EDX	일반적인 계산 용도로 쓰입니다.

$ 중요한 4가지만 설명을 하였고, 더욱 자세한 설명은 다른 문서를 참고하시기 바랍니다.

FLAGS

OF [Over Flow]	산술연산후 상위 비트의 오버플로를 나타냄
DF [Direction]	스트링 데이터를 이동하거나 비교할때 왼쪽 또는 오른쪽으로의 방향을 결정한다.
SF [Sign]	산술결과의 부호를 나타낸다.[0=양수,1=음수]
ZF [zero]	산술연산 또는 비교동작의 결과를 나타낸다. [0=결과가 0이 아님,1=결과가 0임]
CF [Carry]	산술연산후 상위 비트로부터의 캐리 그리고 시프트 또는 회전동작의 결과 마지막 비트 내용을 저장한다.
TF [trap]	프로세서가 단일 스텝 모드(single-step mode)를 동작할수 있도록 해준다.

$ 중요한 6가지만 설명을 하였고, 더욱 자세한 설명은 다른 문서를 참고하시기 바랍니다.

이것으로 IDA 5.x Manual 1부를 마칠까 합니다.
1부에서는 정말 기초적인 메뉴의 구성과 간단한 프로그램의 디버깅모습을 담았고,
2부에서는 좀 더 실용적인 부분과 분석하는 부분을 담도록 하겠습니다.

긴글 읽어주셔서 감사드리고, 오타나 잘못된 내용이 있으면 지적해주시면 감사드리겠습니다.
-〉 certlab@gmail.com

 # Reference

The Art of Assembly Language / Randall Hyde
http://www.asmlove.co.kr
http://www.datarescue.be
http://www.securityproof.org