Lab Report on:

Reverse Engineering

SWE 4802: Software Maintenance Lab

Lutfun Nahar Lota

Assistant Professor

Department of Computer Science and Engineering

Islamic University of Technology

Submitted By:

Abrar Mahabub (200042103)

Mashrur Ahsan (200042115)

Nafisa Maliyat (200042133)

Shanta Maria (200042172)

Table of Contents

IDA Pro

IDA (Interactive Disassembler) is a comprehensive tool used for reverse engineering, which means understanding how a program works without having access to the source code. IDA offers a range of useful features such as disassembling binary files, displaying code flow visually through graphs and flowcharts, and running a debugger to see what each instruction is doing. It also allows use of Python scripts to automate the analysis process. In addition, it also works with many CPU types and platforms, making it suitable for analyzing different types of programs. The progress of any analysis is saved as .idb file so users can continue their work anytime.

Features of IDA Pro

Some prominent features of this tool are as follows:

- **Disassembly**: Converts executables into symbolic assembly code for many file formats and architectures
- **Decompiler (Pseudocode)**: Generates human-readable C-like pseudocode to ease understanding of binary logic (cloud-based in limited versions, local in Pro)
- **Cross-references & Type Reconstruction**: Automatically identifies cross-links, functions, stack frames, variables, and reconstructed data types
- **Interactive Editing**: Analysts can override auto-analysis results—rename, re-flag, retype—for precision reverse-engineering
- **Programmability & Automation**: Support scripting via IDC and IDAPython, plus plugin support to extend capabilities.
- **Plugin Architecture**: Open SDK enables writing GUI enhancements and analysis tools; large ecosystem of community plugins
- **FLIRT Signature Recognition**: Automatically identifies standard library functions to simplify disassembly
- **idalib (Headless Mode)**: Use IDA engine as a library for C++ or Python automation, running offline or in batch environments (Pro only)
- **Private Lumina Add-on**: Enterprise feature for sharing function metadata and analysis across teams (Pro only)
- **Teams Add-on**: Enables team collaboration, file version control, and syncing (Pro only)

C Code

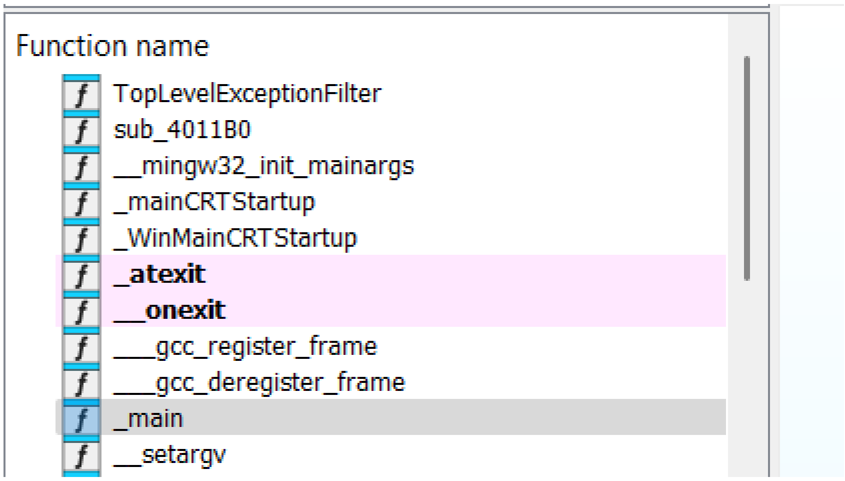
This program is designed to determine whether a given year is a leap year. It accepts a numerical input representing a year and applies a sequence of conditional logic. Specifically, the program checks if the year is divisible by 4 but not by 100, except when it is also divisible by 400. If these conditions are satisfied, the year is classified as a leap year, which comprises 366 days. Otherwise, the year is identified as a common year with 365 days.

```c
#include <stdio.h>
int main() {
   int year;
   printf("Enter a year: ");
   scanf("%d", &year);

   if (year % 400 == 0) {
      printf("%d is a leap year.", year);
   }

   else if (year % 100 == 0) {
      printf("%d is not a leap year.", year);
   }

   else if (year % 4 == 0) {
      printf("%d is a leap year.", year);
   }
   else {
      printf("%d is not a leap year.", year);
   }

   return 0;
}
```

Analysis of our C code

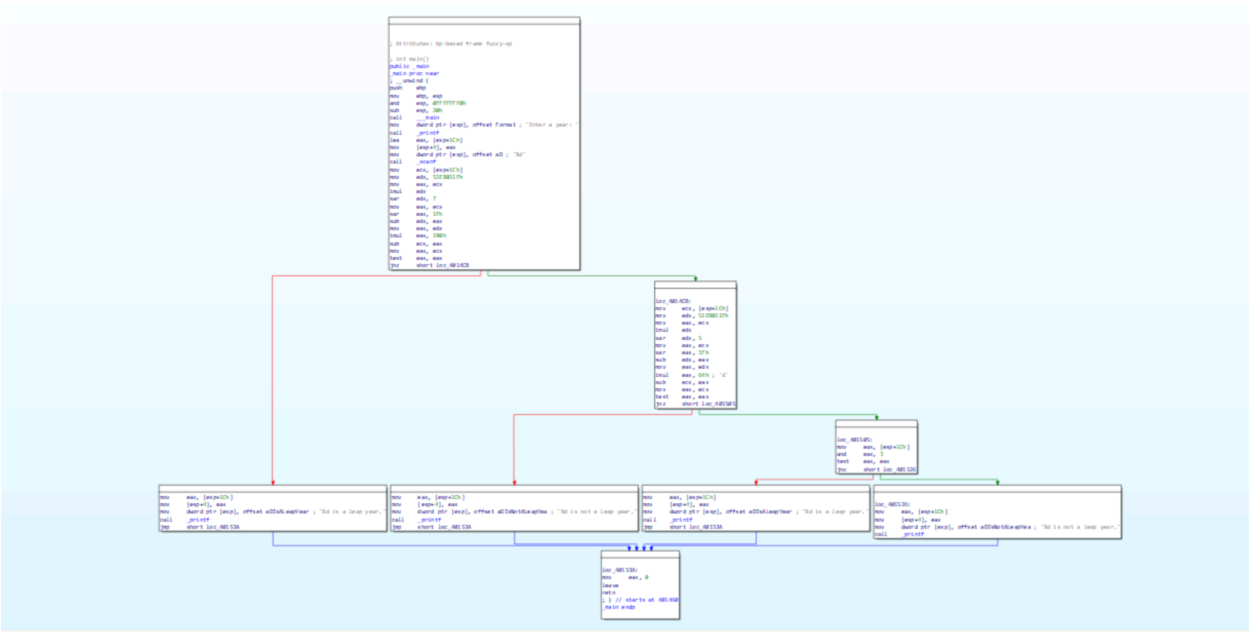
Main function Identification

- In the left panel, we have the functions window
- After scrolling through the list of detected functions, we can find a function named "_main".



```
Function name
TopLevelExceptionFilter
sub_4011B0
__mingw32_init_mainargs
_mainCRTStartup
_WinMainCRTStartup
_atexit
__onexit
___gcc_register_frame
___gcc_deregister_frame
_main
__setargv
```

- We can view IDA view-A by double-clicking the "_main" function.



```
; Attributes: bp-based frame fuzzy-sp

; int main()
public _main
_main proc near
; __unwind {
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 20h
call    ___main
mov     dword ptr [esp], offset Format ; "Enter a year: "
call    _printf
lea     eax, [esp+1Ch]
mov     [esp+4], eax
mov     dword ptr [esp], offset aD ; "%d"
call    _scanf
mov     ecx, [esp+1Ch]
mov     edx, 51EB851Fh
mov     eax, ecx
imul    edx
sar     edx, 7
mov     eax, ecx
sar     eax, 1Fh
sub     edx, eax
mov     eax, edx
imul    eax, 190h
sub     ecx, eax
mov     eax, ecx
test    eax, eax
jnz     short loc_4014CB
```

```
loc_4014CB:
mov     ecx, [esp+1Ch]
mov     edx, 51EB851Fh
mov     eax, ecx
imul    edx
sar     edx, 5
mov     eax, ecx
sar     eax, 1Fh
sub     edx, eax
mov     eax, edx
imul    eax, 64h ; 'd'
sub     ecx, eax
mov     eax, ecx
test    eax, eax
jnz     short loc_401505
```

```
loc_401505:
mov     eax, [esp+1Ch]
and     eax, 3
test    eax, eax
jnz     short loc_401526
```

```
mov     eax, [esp+1Ch]
mov     [esp+4], eax
mov     dword ptr [esp], offset aDIsALeapYear ; "%d is a leap year."
call    _printf
jmp     short loc_40153A
```

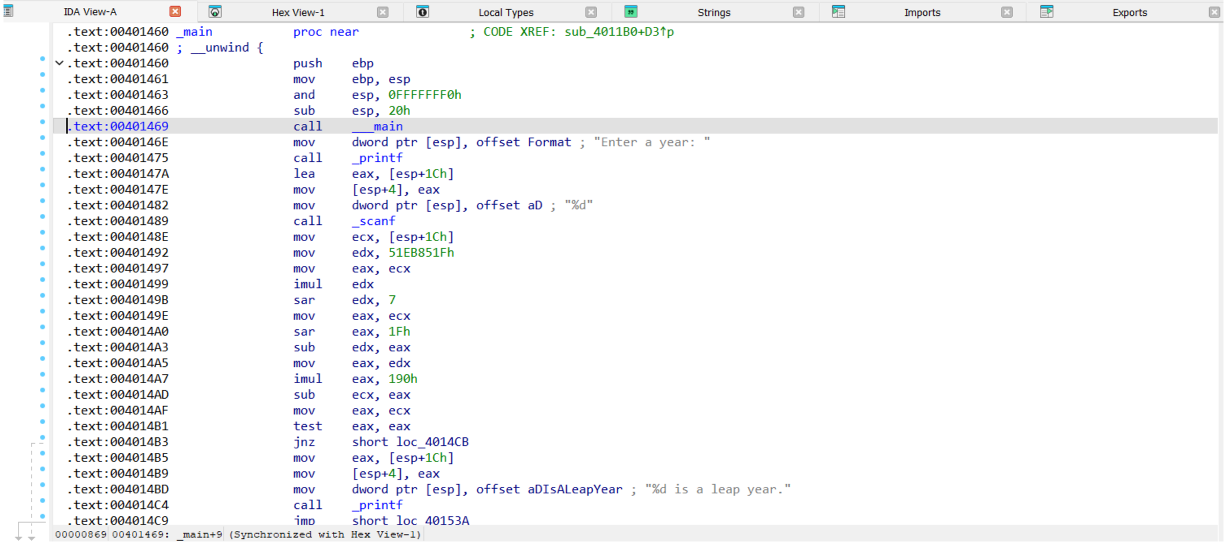
```
mov     eax, [esp+1Ch]
mov     [esp+4], eax
mov     dword ptr [esp], offset aDIsNotALeapYea ; "%d is not a leap year."
call    _printf
jmp     short loc_40153A
```

```
mov     eax, [esp+1Ch]
mov     [esp+4], eax
mov     dword ptr [esp], offset aDIsALeapYear ; "%d is a leap year."
call    _printf
jmp     short loc_40153A
```

```
loc_401526:
mov     eax, [esp+1Ch]
mov     [esp+4], eax
mov     dword ptr [esp], offset aDIsNotALeapYea ; "%d is not a leap year."
call    _printf
```

```
loc_40153A:
mov     eax, 0
leave
retn
; } // starts at 401460
_main endp
```

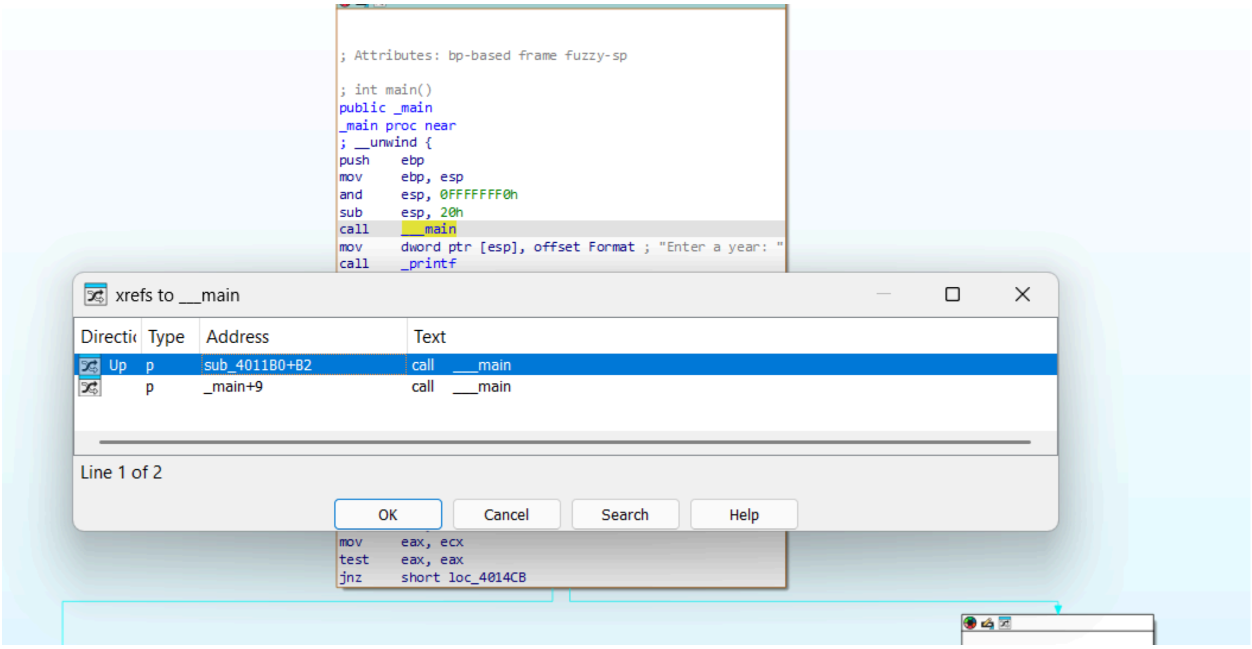
- We can toggle between Linear view and graph view using the space bar.



```
.text:00401460 _main           proc near               ; CODE XREF: sub_4011B0+D3↑p
.text:00401460 ; __unwind {
.text:00401460                 push    ebp
.text:00401461                 mov     ebp, esp
.text:00401463                 and     esp, 0FFFFFFF0h
.text:00401466                 sub     esp, 20h
.text:00401469                 call    ___main
.text:0040146E                 mov     dword ptr [esp], offset Format ; "Enter a year: "
.text:00401475                 call    _printf
.text:0040147A                 lea     eax, [esp+1Ch]
.text:0040147E                 mov     [esp+4], eax
.text:00401482                 mov     dword ptr [esp], offset aD ; "%d"
.text:00401489                 call    _scanf
.text:0040148E                 mov     ecx, [esp+1Ch]
.text:00401492                 mov     edx, 51EB851Fh
.text:00401497                 mov     eax, ecx
.text:00401499                 imul    edx
.text:0040149B                 sar     edx, 7
.text:0040149E                 mov     eax, ecx
.text:004014A0                 sar     eax, 1Fh
.text:004014A3                 sub     edx, eax
.text:004014A5                 mov     eax, edx
.text:004014A7                 imul    eax, 190h
.text:004014AD                 sub     ecx, eax
.text:004014AF                 mov     eax, ecx
.text:004014B1                 test    eax, eax
.text:004014B3                 jnz     short loc_4014CB
.text:004014B5                 mov     eax, [esp+1Ch]
.text:004014B9                 mov     [esp+4], eax
.text:004014BD                 mov     dword ptr [esp], offset aDIsALeapYear ; "%d is a leap year."
.text:004014C4                 call    _printf
.text:004014C9                 jmp     short loc_40153A
00000869 00401469: _main+9 (Synchronized with Hex View-1)
```

Cross-referencing Identification

- In IDA view-A, we will hover the cursor on the function we wish to investigate.
- By pressing X, we can view the cross-references in the window.
- In the window, it will display the location of the function being called.



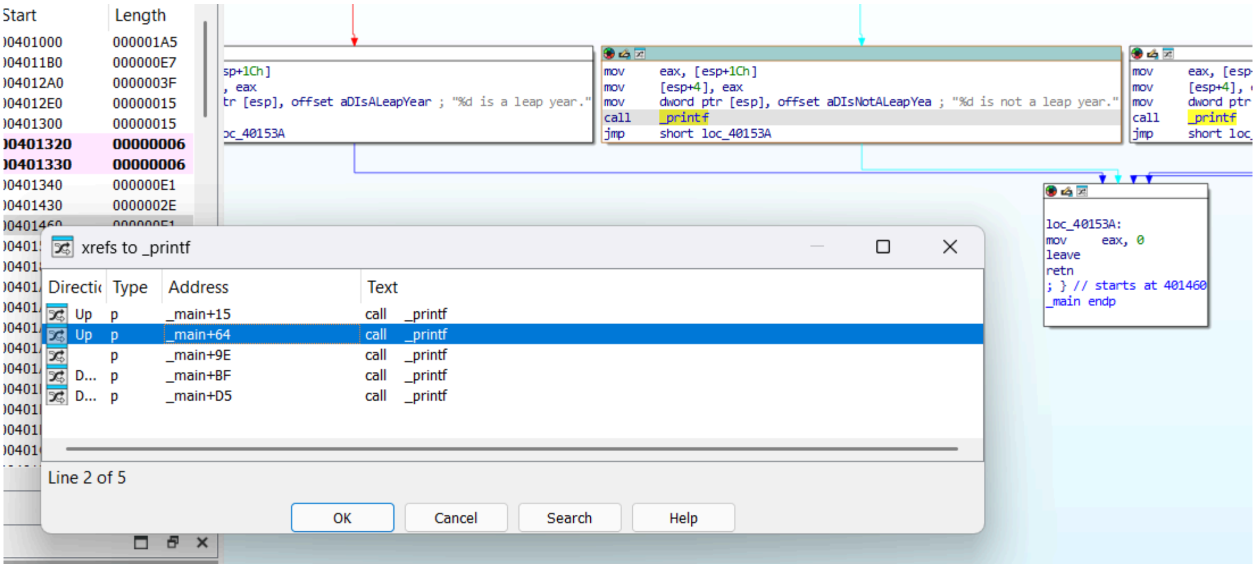
```
; Attributes: bp-based frame fuzzy-sp

; int main()
public _main
_main proc near
; __unwind {
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 20h
call    ___main
mov     dword ptr [esp], offset Format ; "Enter a year: "
call    _printf
```

xrefs to ___main

Directi	Type	Address	Text
Up	p	sub_4011B0+B2	call ___main
	p	_main+9	call ___main

Line 1 of 2

```
mov     eax, ecx
test    eax, eax
jnz     short loc_4014CB
```



Start	Length
)0401000	000001A5
)04011B0	000000E7
)04012A0	0000003F
)04012E0	00000015
)0401300	00000015
)0401320	**00000006**
)0401330	**00000006**
)0401340	000000E1
)0401430	0000002E

```
sp+1Ch]
, eax
tr [esp], offset aDIsALeapYear ; "%d is a leap year."
oc_40153A
```

```
mov     eax, [esp+1Ch]
mov     [esp+4], eax
mov     dword ptr [esp], offset aDIsNotALeapYea ; "%d is not a leap year."
call    _printf
jmp     short loc_40153A
```

```
mov     eax, [esp
mov     [esp+4],
mov     dword ptr
call    _printf
jmp     short loc_
```

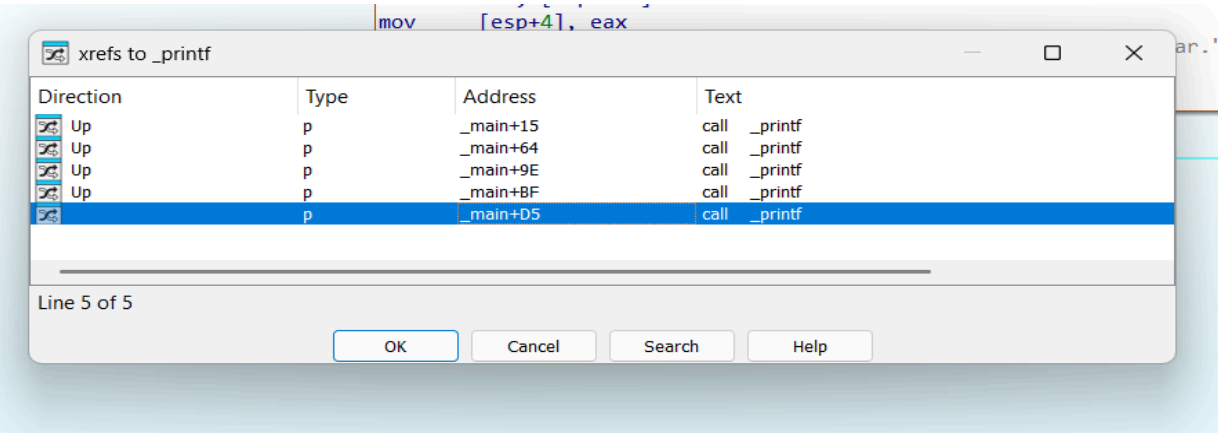
```
loc_40153A:
mov     eax, 0
leave
retn
; } // starts at 401460
_main endp
```

xrefs to _printf

Directi	Type	Address	Text
Up	p	_main+15	call _printf
Up	p	_main+64	call _printf
	p	_main+9E	call _printf
D...	p	_main+BF	call _printf
D...	p	_main+D5	call _printf

Line 2 of 5

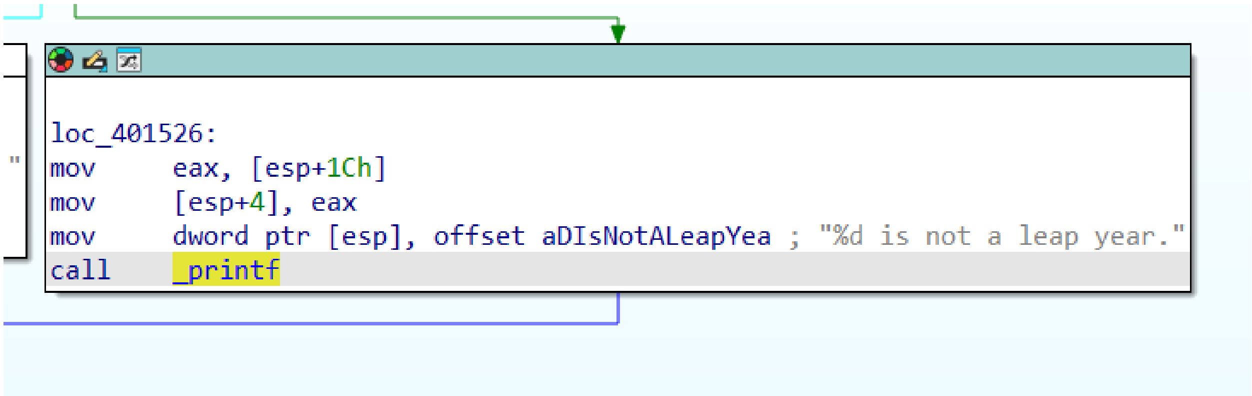
- Double-clicking any entry in the list to navigate directly to the usage.



xrefs to _printf

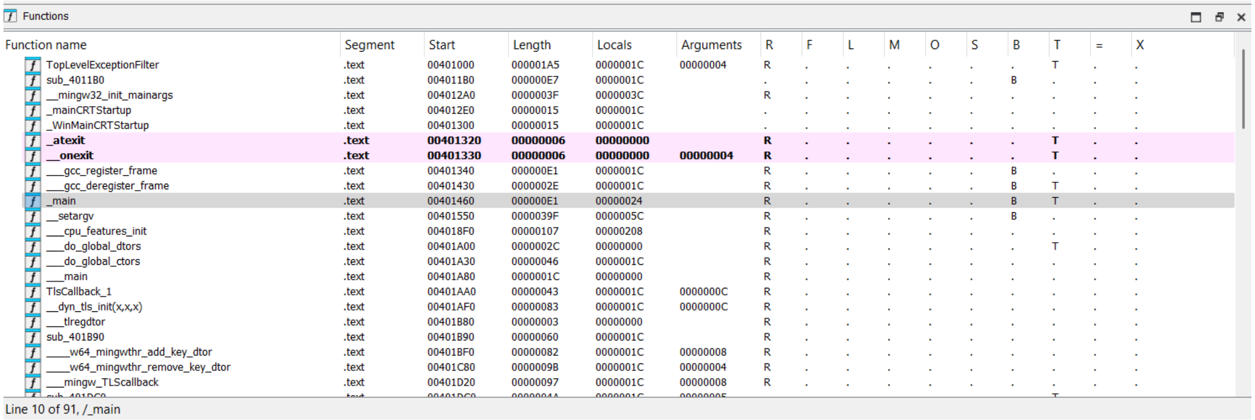
Direction	Type	Address	Text
Up	p	_main+15	call _printf
Up	p	_main+64	call _printf
Up	p	_main+9E	call _printf
Up	p	_main+BF	call _printf
	p	_main+D5	call _printf

Line 5 of 5



```
loc_401526:
mov     eax, [esp+1Ch]
mov     [esp+4], eax
mov     dword ptr [esp], offset aDIsNotALeapYea ; "%d is not a leap year."
call    _printf
```

The Functions window



Functions

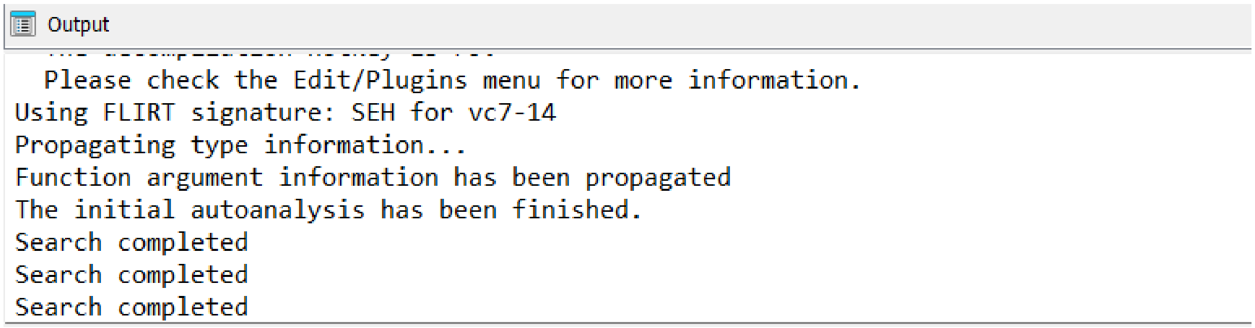
Function name	Segment	Start	Length	Locals	Arguments	R	F	L	M	O	S	B	T	=	X
TopLevelExceptionFilter	.text	00401000	000001A5	0000001C	00000004	R	.	.	.	.	.	.	T	.	.
sub_4011B0	.text	004011B0	000000E7	0000001C		.	.	.	.	.	.	B	.	.	.
__mingw32_init_mainargs	.text	004012A0	0000003F	0000003C		R	.	.	.	.	.	.	.	.	.
_mainCRTStartup	.text	004012E0	00000015	0000001C		.	.	.	.	.	.	.	.	.	.
_WinMainCRTStartup	.text	00401300	00000015	0000001C		.	.	.	.	.	.	.	.	.	.
_atexit	**.text**	**00401320**	**00000006**	**00000000**		**R**	.	.	.	.	.	.	**T**	.	.
__onexit	**.text**	**00401330**	**00000006**	**00000000**	**00000004**	**R**	.	.	.	.	.	.	**T**	.	.
___gcc_register_frame	.text	00401340	000000E1	0000001C		R	.	.	.	.	.	B	.	.	.
___gcc_deregister_frame	.text	00401430	0000002E	0000001C		R	.	.	.	.	.	B	T	.	.
_main	.text	00401460	000000E1	00000024		R	.	.	.	.	.	B	T	.	.
__setargv	.text	00401550	0000039F	0000005C		R	.	.	.	.	.	B	.	.	.
___cpu_features_init	.text	004018F0	00000107	00000208		R	.	.	.	.	.	.	.	.	.
___do_global_dtors	.text	00401A00	0000002C	00000000		R	.	.	.	.	.	.	T	.	.
___do_global_ctors	.text	00401A30	00000046	0000001C		R	.	.	.	.	.	.	.	.	.
___main	.text	00401A80	0000001C	00000000		R	.	.	.	.	.	.	.	.	.
TlsCallback_1	.text	00401AA0	00000043	0000001C	0000000C	R	.	.	.	.	.	.	.	.	.
__dyn_tls_init(x,x,x)	.text	00401AF0	00000083	0000001C	0000000C	R	.	.	.	.	.	.	.	.	.
___tlregdtor	.text	00401B80	00000003	00000000		R	.	.	.	.	.	.	.	.	.
sub_401B90	.text	00401B90	00000060	0000001C		R	.	.	.	.	.	.	.	.	.
____w64_mingwthr_add_key_dtor	.text	00401BF0	00000082	0000001C	00000008	R	.	.	.	.	.	.	.	.	.
____w64_mingwthr_remove_key_dtor	.text	00401C80	0000009B	0000001C	00000004	R	.	.	.	.	.	.	.	.	.
___mingw_TLScallback	.text	00401D20	00000097	0000001C	00000008	R	.	.	.	.	.	.	.	.	.

Line 10 of 91, /_main

Here we can see several columns in the “Functions” window. Here is what the columns represent:

- The first column Function names
- Second column signifies the segment
- Third column shows the start Memory locations, fourth column shows the whole length
- Arguments column shows the number of arguments the functions that in
- Rest of the columns represent different status(s) like referred, library, manual etc.

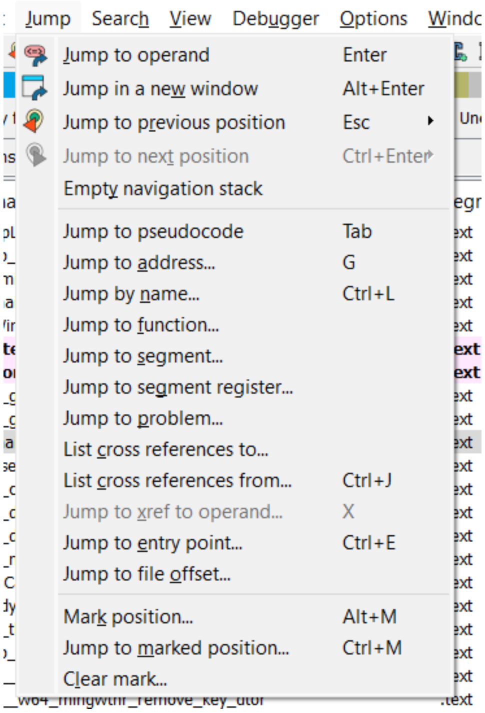
Output window

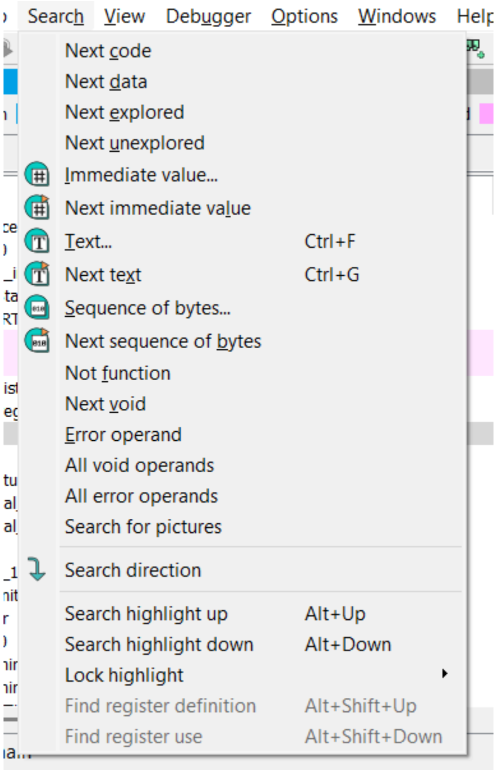


```
  Please check the Edit/Plugins menu for more information.
Using FLIRT signature: SEH for vc7-14
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
Search completed
Search completed
Search completed
```

The output window shows all the logs during our analysis of the code. It acts as a logger. It shows the sequence of operations and logs confirmation messages accordingly.

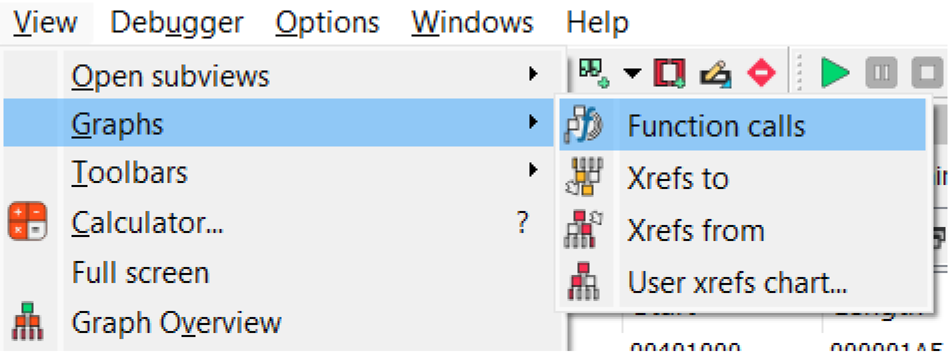
Jump, Search, View options


Jump Search View Debugger Options Windc
Jump to operand Enter
Jump in a new window Alt+Enter
Jump to previous position Esc
Jump to next position Ctrl+Enter
Empty navigation stack
Jump to pseudocode Tab
Jump to address... G
Jump by name... Ctrl+L
Jump to function...
Jump to segment...
Jump to segment register...
Jump to problem...
List cross references to...
List cross references from... Ctrl+J
Jump to xref to operand... X
Jump to entry point... Ctrl+E
Jump to file offset...
Mark position... Alt+M
Jump to marked position... Ctrl+M
Clear mark...



Here we can see several columns in the “Functions” window. Here is what the columns represent:

- Jump option lets one jump directly to an operand, address, memory location, segment, function etc.
- The search operation lets one search for some data in code, memory locations
- The view option lets one look at function calls, cross references, references from or to, user charts



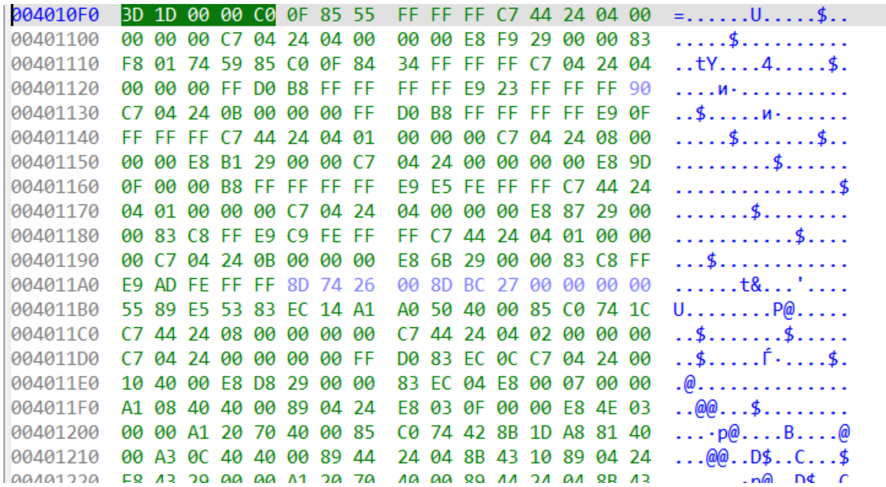
Hex View-1

Hex View-1 in IDA shows the raw bytes of the binary file. The format of the analysis of the compiled code in hex view-1 is as follows:

```
<Memory Address>   <Hex Bytes>   <ASCII equivalent (not always shown)>
```

The raw bytes represent the low-level assembly instructions, not the actual code. But these low-level instructions map to the high-level C code that the original code is written in.

Below is a small part of the hex byte breakdown for our code:

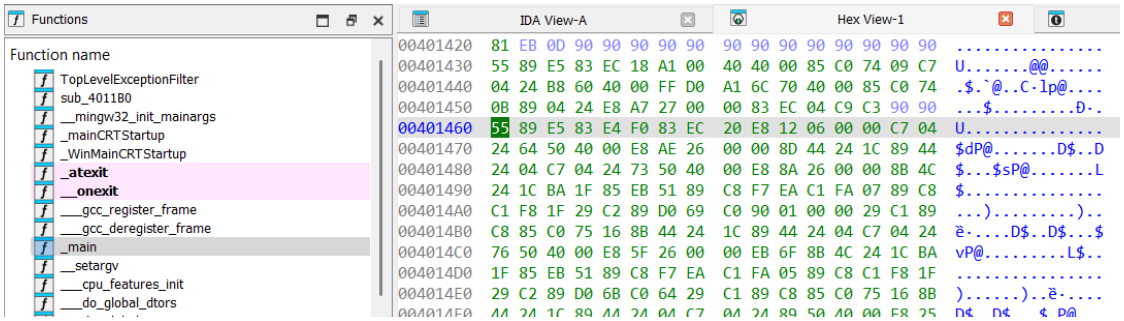


```
004010F0  3D 1D 00 00 C0 0F 85 55  FF FF FF C7 44 24 04 00  =......U.....$..
00401100  00 00 00 C7 04 24 04 00  00 00 E8 F9 29 00 00 83  .....$..........
00401110  F8 01 74 59 85 C0 0F 84  34 FF FF FF C7 04 24 04  ..tY....4.....$.
00401120  00 00 00 FF D0 B8 FF FF  FF FF E9 23 FF FF FF 90  ....и·..........
00401130  C7 04 24 0B 00 00 00 FF  D0 B8 FF FF FF FF E9 0F  ..$.....и·......
00401140  FF FF FF C7 44 24 04 01  00 00 00 C7 04 24 08 00  .....$.......$..
00401150  00 00 E8 B1 29 00 00 C7  04 24 00 00 00 00 E8 9D  .........$......
00401160  0F 00 00 B8 FF FF FF FF  E9 E5 FE FF FF C7 44 24  ...............$
00401170  04 01 00 00 00 C7 04 24  04 00 00 00 E8 87 29 00  .......$........
00401180  00 83 C8 FF E9 C9 FE FF  FF C7 44 24 04 01 00 00  ...........$....
00401190  00 C7 04 24 0B 00 00 00  E8 6B 29 00 00 83 C8 FF  ...$............
004011A0  E9 AD FE FF FF 8D 74 26  00 8D BC 27 00 00 00 00  ......t&...'....
004011B0  55 89 E5 53 83 EC 14 A1  A0 50 40 00 85 C0 74 1C  U........P@.....
004011C0  C7 44 24 08 00 00 00 00  C7 44 24 04 02 00 00 00  ..$.......$.....
004011D0  C7 04 24 00 00 00 00 FF  D0 83 EC 0C C7 04 24 00  ..$.....ŕ·....$.
004011E0  10 40 00 E8 D8 29 00 00  83 EC 04 E8 00 07 00 00  .@..............
004011F0  A1 08 40 40 00 89 04 24  E8 03 0F 00 00 E8 4E 03  ..@@...$........
00401200  00 00 A1 20 70 40 00 85  C0 74 42 8B 1D A8 81 40  ....·p@....B....@
00401210  00 A3 0C 40 40 00 89 44  24 04 8B 43 10 89 04 24  ...@@..D$..C...$
00401220  E8 43 29 00 00 A1 20 70  40 00 89 44 24 04 8B 43
```

From this breakdown, we can figure out some instructions, for example:

```
00401155: 00 00 E8 B1 29 00 00 C7
```

- **Instruction**: `jne 0x00401050`
- **Meaning**: Jump to address `00401155` if the comparison above was not equal.
- So this is a **conditional jump** that is `if` or `else if` logic.

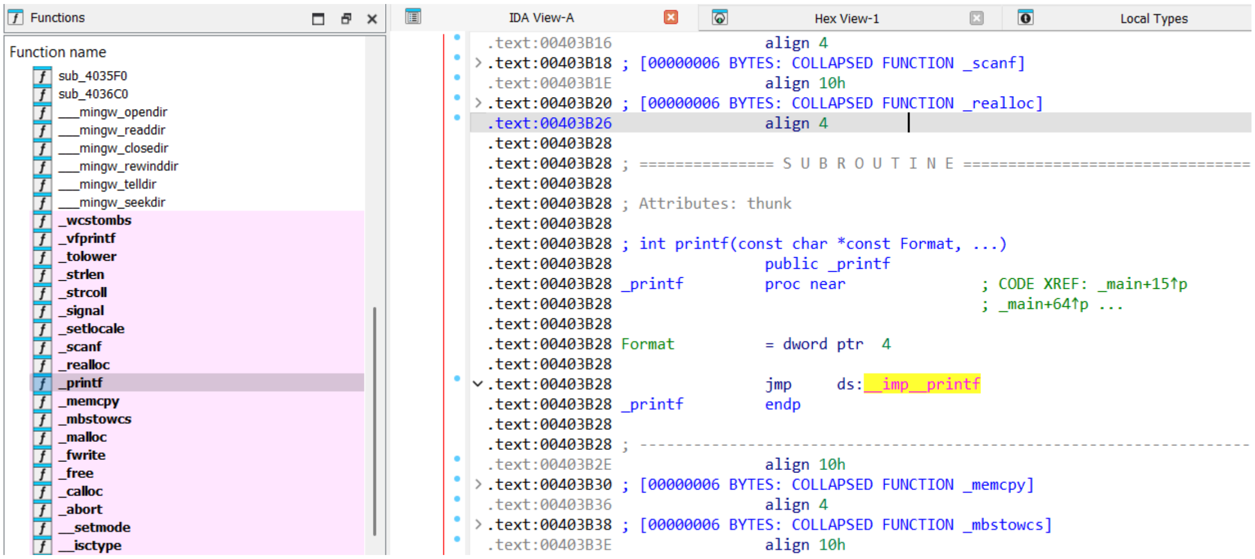


Function name
- TopLevelExceptionFilter
- sub_4011B0
- __mingw32_init_mainargs
- _mainCRTStartup
- _WinMainCRTStartup
- **_atexit**
- **__onexit**
- ___gcc_register_frame
- ___gcc_deregister_frame
- _main
- __setargv
- ___cpu_features_init
- ___do_global_dtors

```
00401420  81 EB 0D 90 90 90 90 90  90 90 90 90 90 90 90 90  ................
00401430  55 89 E5 83 EC 18 A1 00  40 40 00 85 C0 74 09 C7  U.......@@......
00401440  04 24 B8 60 40 00 FF D0  A1 6C 70 40 00 85 C0 74  .$.`@..C·lp@....
00401450  0B 89 04 24 E8 A7 27 00  00 83 EC 04 C9 C3 90 90  ...$.........Đ·.
00401460  55 89 E5 83 E4 F0 83 EC  20 E8 12 06 00 00 C7 04  U...............
00401470  24 64 50 40 00 E8 AE 26  00 00 8D 44 24 1C 89 44  $dP@.......D$..D
00401480  24 04 C7 04 24 73 50 40  00 E8 8A 26 00 00 8B 4C  $...$sP@.......L
00401490  24 1C BA 1F 85 EB 51 89  C8 F7 EA C1 FA 07 89 C8  $...............
004014A0  C1 F8 1F 29 C2 89 D0 69  C0 90 01 00 00 29 C1 89  ...)........)..
004014B0  C8 85 C0 75 16 8B 44 24  1C 89 44 24 04 C7 04 24  ȅ·....D$..D$...$
004014C0  76 50 40 00 E8 5F 26 00  00 EB 6F 8B 4C 24 1C BA  vP@.........L$..
004014D0  1F 85 EB 51 89 C8 F7 EA  C1 FA 05 89 C8 C1 F8 1F  ................
004014E0  29 C2 89 D0 6B C0 64 29  C1 89 C8 85 C0 75 16 8B  )......)..ȅ·....
004014F0  44 24 1C 89 44 24 04 C7  04 24 89 50 40 00 E8 25
```

If we click on a function name in the Functions panel, the respective address in the Hex View-1 panel would be shown. In the above example, we can see that upon clicking the `main` function we can see it is stored at `0x00401050`.

IDA View-A (Disassembled Code)



```
Function name
sub_4035F0
sub_4036C0
___mingw_opendir
___mingw_readdir
___mingw_closedir
___mingw_rewinddir
___mingw_telldir
___mingw_seekdir
_wcstombs
_vfprintf
_tolower
_strlen
_strcoll
_signal
_setlocale
_scanf
_realloc
_printf
_memcpy
_mbstowcs
_malloc
_fwrite
_free
_calloc
_abort
__setmode
__isctype
```

```
.text:00403B16                 align 4
.text:00403B18 ; [00000006 BYTES: COLLAPSED FUNCTION _scanf]
.text:00403B1E                 align 10h
.text:00403B20 ; [00000006 BYTES: COLLAPSED FUNCTION _realloc]
.text:00403B26                 align 4
.text:00403B28
.text:00403B28 ; =============== S U B R O U T I N E ================================
.text:00403B28
.text:00403B28 ; Attributes: thunk
.text:00403B28
.text:00403B28 ; int printf(const char *const Format, ...)
.text:00403B28                 public _printf
.text:00403B28 _printf         proc near               ; CODE XREF: _main+15↑p
.text:00403B28                                         ; _main+64↑p ...
.text:00403B28
.text:00403B28 Format          = dword ptr  4
.text:00403B28
.text:00403B28                 jmp     ds:__imp__printf
.text:00403B28 _printf         endp
.text:00403B28
.text:00403B28 ; ---------------------------------------------------------------------------
.text:00403B2E                 align 10h
.text:00403B30 ; [00000006 BYTES: COLLAPSED FUNCTION _memcpy]
.text:00403B36                 align 4
.text:00403B38 ; [00000006 BYTES: COLLAPSED FUNCTION _mbstowcs]
.text:00403B3E                 align 10h
```

The code in the screenshot shows a tiny helper function called `_printf`. It's **not** the real `printf` function.

Instead, this helper just **jumps to the real `printf`**, which lives in a system file like a DLL (for example, `msvcrt.dll`).

This is how programs usually work when they use common functions like `printf` — they don't include the whole code for `printf` inside the program. They just **link to it** and call it when needed.

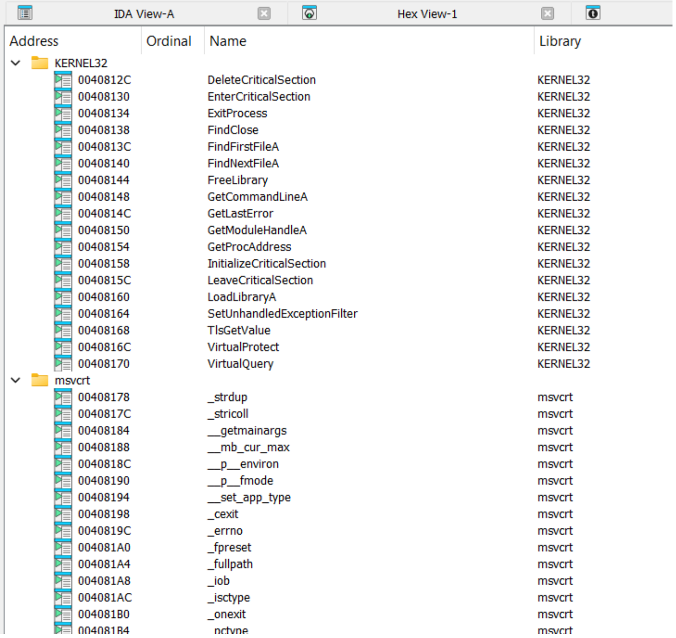
So, this `_printf` function is like a call to the `printf` function in the system library.

Imports

The imported functions can be viewed and analysed in the Imports tab. It shows all functions the program uses from external libraries in this case:

- `printf`
- `scanf`

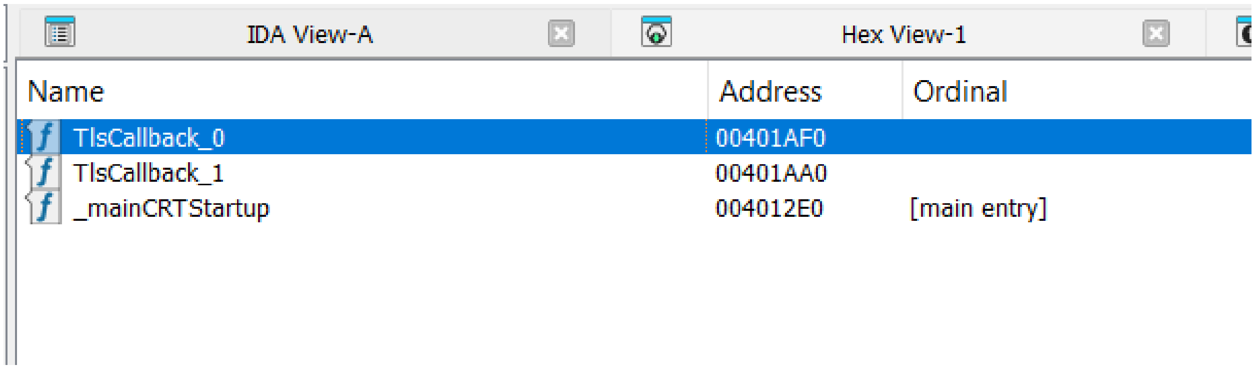
The functions are linked from (like msvcrt.dll or kernel32.dll). The addresses or pointers used to jump to those functions. You can double-click any import to see where it's used in the disassembly.



Address	Ordinal	Name	Library
KERNEL32			
0040812C		DeleteCriticalSection	KERNEL32
00408130		EnterCriticalSection	KERNEL32
00408134		ExitProcess	KERNEL32
00408138		FindClose	KERNEL32
0040813C		FindFirstFileA	KERNEL32
00408140		FindNextFileA	KERNEL32
00408144		FreeLibrary	KERNEL32
00408148		GetCommandLineA	KERNEL32
0040814C		GetLastError	KERNEL32
00408150		GetModuleHandleA	KERNEL32
00408154		GetProcAddress	KERNEL32
00408158		InitializeCriticalSection	KERNEL32
0040815C		LeaveCriticalSection	KERNEL32
00408160		LoadLibraryA	KERNEL32
00408164		SetUnhandledExceptionFilter	KERNEL32
00408168		TlsGetValue	KERNEL32
0040816C		VirtualProtect	KERNEL32
00408170		VirtualQuery	KERNEL32
msvcrt			
00408178		_strdup	msvcrt
0040817C		_stricoll	msvcrt
00408184		__getmainargs	msvcrt
00408188		__mb_cur_max	msvcrt
0040818C		__p__environ	msvcrt
00408190		__p__fmode	msvcrt
00408194		__set_app_type	msvcrt
00408198		_cexit	msvcrt
0040819C		_errno	msvcrt
004081A0		_fpreset	msvcrt
004081A4		_fullpath	msvcrt
004081A8		_iob	msvcrt
004081AC		_isctype	msvcrt
004081B0		_onexit	msvcrt
004081B4		_pctype	msvcrt

Exports

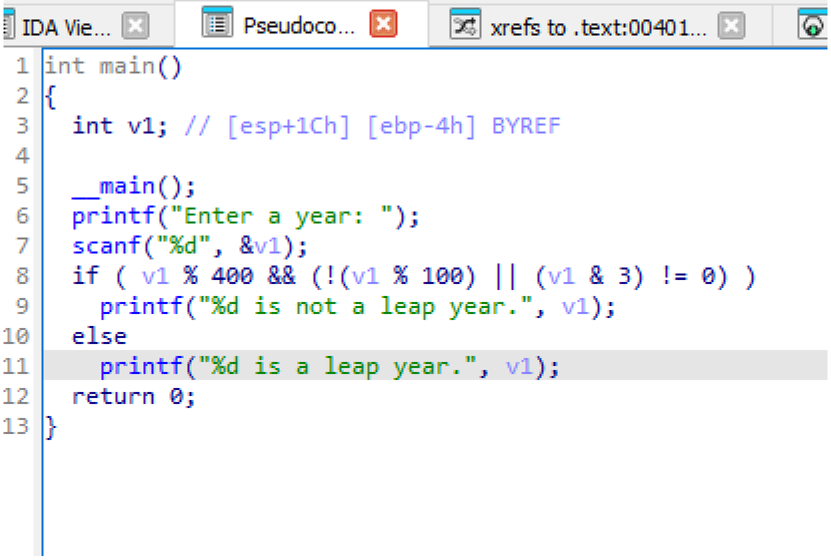
Shows all the functions that we export from our code for use in other programs. We have no export function in our code as shown except main:



Name	Address	Ordinal
TlsCallback_0	00401AF0	
TlsCallback_1	00401AA0	
_mainCRTStartup	004012E0	[main entry]

`_mainCRTStartup` is stored at the Address shown (`004012E0`) that is the main function of our C program.

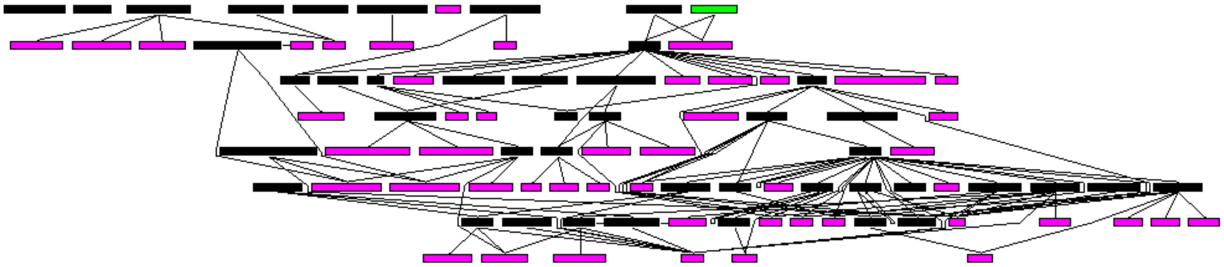
PseudoCode Generation



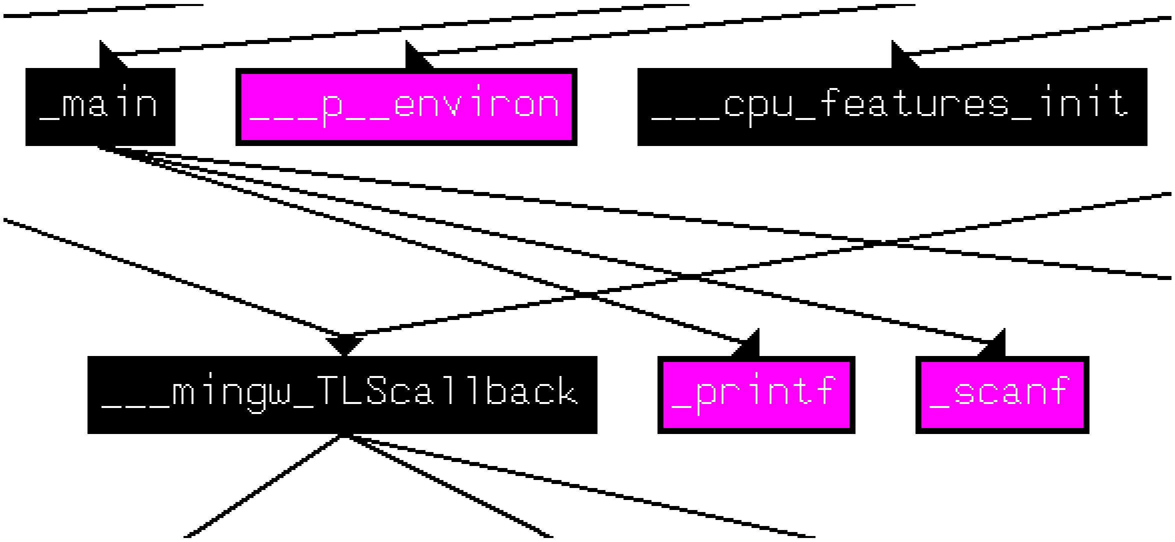
```
int main()
{
  int v1; // [esp+1Ch] [ebp-4h] BYREF

  __main();
  printf("Enter a year: ");
  scanf("%d", &v1);
  if ( v1 % 400 && (!(v1 % 100) || (v1 & 3) != 0) )
    printf("%d is not a leap year.", v1);
  else
    printf("%d is a leap year.", v1);
  return 0;
}
```

IDA tool also provides an option to view the pseudocode of a binary through its **Hex-Rays Decompiler plugin**. The pseudocode, although not exactly the original source code, is much more readable and helps to quickly figure out what the program is doing — like how functions work, how data is being used, and what the overall logic looks like. The pseudocode helps analysts understand the logic and control flow of the program without having to interpret complex assembly instructions directly.

Graph View of Code Flow



The Graph View in IDA provides a clear visual representation of a function's control flow, displaying how the program executes from start to finish. It shows the function calls and branches in a top-to-bottom format, making it easier to follow the logic and structure. It also allows zooming in and out to explore the flow between different blocks and understand how one part of the function leads to another.



If the view is expanded, it shows `printf` and `scanf` function calls from main. This provides an understanding that the main function is directly calling these two functions, as well as other built-in functions. The view provides a clearer picture of the flow of execution within the program.