Reverse Engineering Structures

Introduction

In this part of the tutorial, we'll take a look at how we can figure out a structure when reverse engineering a binary. First, we must write a C++ program that declares and uses the structure, so that we'll be able to reverse engineer it. The basic difference between arrays and structures is the fact that we're using an index to address consecutive elements of the array, whereas with structures we're using named members to access specific data within the structure.

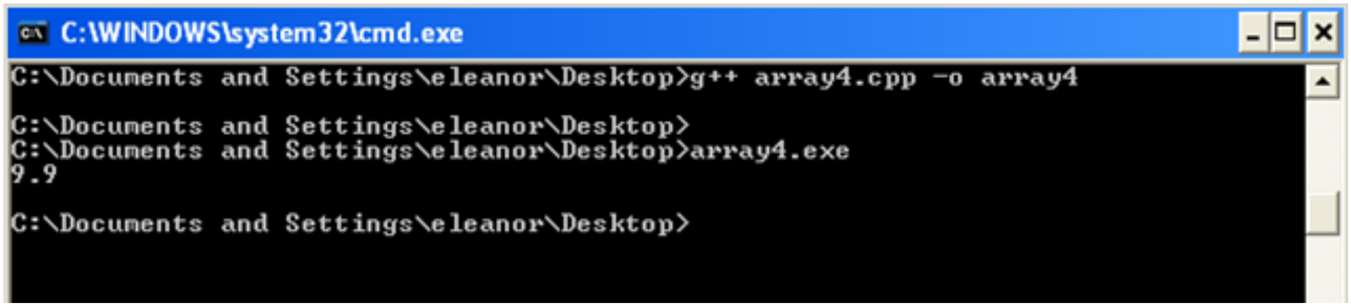
When working with structures, we must keep in mind that the size of the structure is declared as the sum of all its data members aligned on word boundary in memory. What does that mean? It means that the compiler will align each data structure on a 4-byte boundary, so it can read and write member values from memory more efficiently.

Global Structures

The program written in C++ that uses global structures can be seen below:

```
#include <iostream>

struct s {
  int x;
  int y;
  int z;
  double value;
} mys;

int main(int argc, char **argv) {
  mys.x = 1;
  mys.y = 2;
  mys.z = 3;
  mys.value = 9.9;
  std::cout << mys.value << std::endl;

  return 0;
}
```

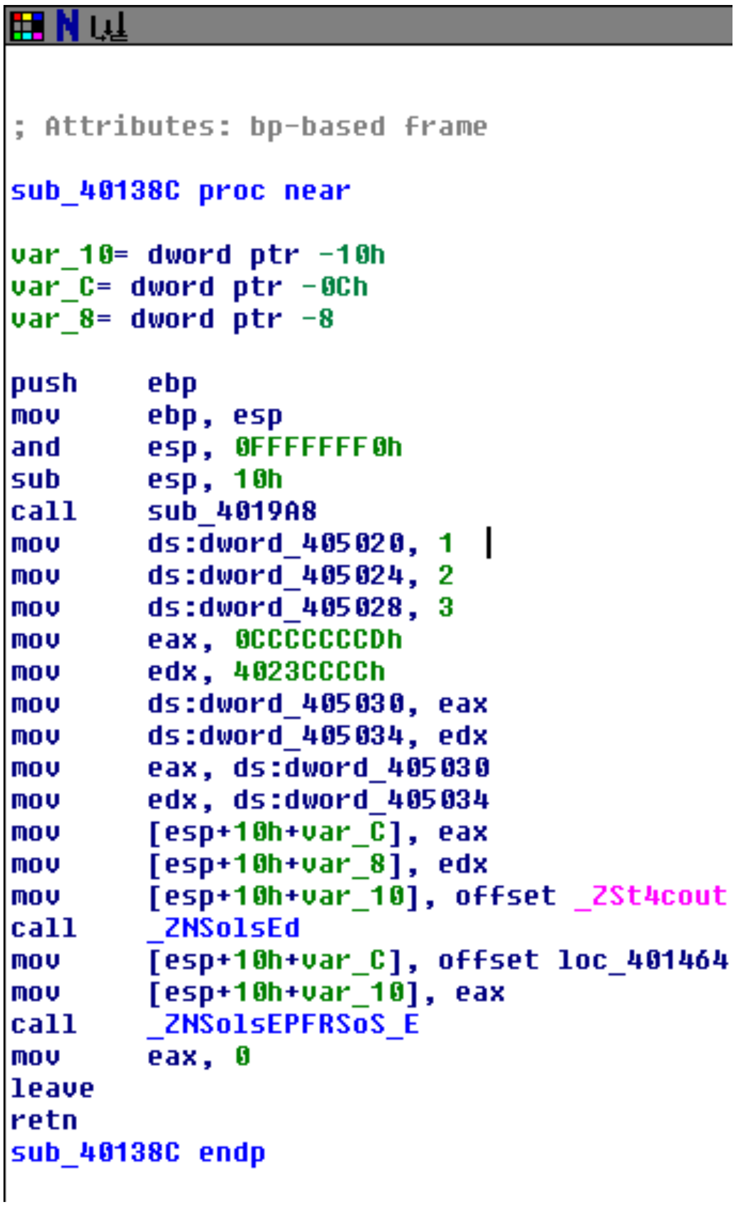
Let's compile and run the program to see what it does. We can do that by downloading the MinGW software package in Windows and issue the two commands that can be seen on the picture below:



```
C:\Documents and Settings\eleanor\Desktop>g++ array4.cpp -o array4

C:\Documents and Settings\eleanor\Desktop>
C:\Documents and Settings\eleanor\Desktop>array4.exe
9.9

C:\Documents and Settings\eleanor\Desktop>
```

We compiled the program with the g++ compiler and after running it, the program outputted the number 9.9. In the source code of the program, we're first defining a structure that has four members: variable x, an integer; variable y, an integer; variable z, an integer; and variable value, a double. The structure can represent points and their values in a three-dimensional space. We're also defining an instance of the structure named mys when declaring the structure: note that this is just a shortcut to declaring the structure in a normal way like "struct s mys."

If we load the program in Ida, we can quickly find the following disassembly that initialized the numbers 1, 2, 3 to the x, y, z members of a structure and which defined the value of member 'value' to be 9.9. The disassembly can be seen on the picture below:



```
; Attributes: bp-based frame

sub_40138C proc near

var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 10h
call    sub_4019A8
mov     ds:dword_405020, 1
mov     ds:dword_405024, 2
mov     ds:dword_405028, 3
mov     eax, 0CCCCCCCDh
mov     edx, 4023CCCCh
mov     ds:dword_405030, eax
mov     ds:dword_405034, edx
mov     eax, ds:dword_405030
mov     edx, ds:dword_405034
mov     [esp+10h+var_C], eax
mov     [esp+10h+var_8], edx
mov     [esp+10h+var_10], offset _ZSt4cout
call    _ZNSolsEd
mov     [esp+10h+var_C], offset loc_401464
mov     [esp+10h+var_10], eax
call    _ZNSolsEPFRSoS_E
mov     eax, 0
leave
retn
sub_40138C endp
```

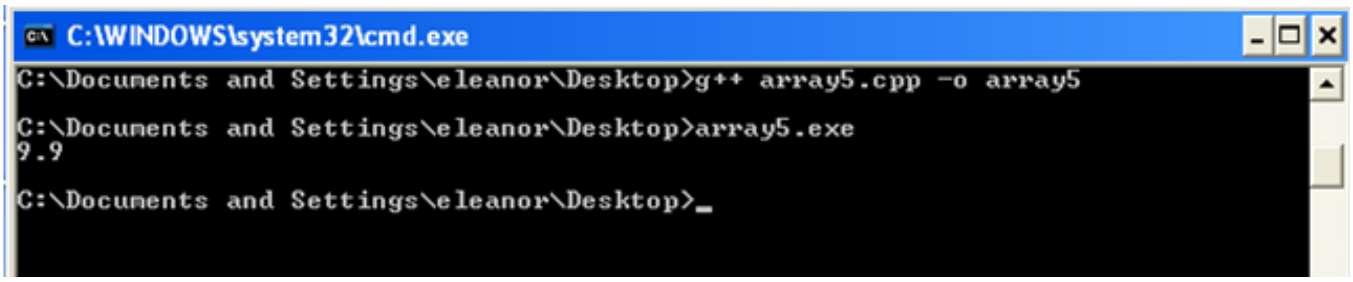
In the assembly code, we can see the direct assignment of values 1, 2, 3 and 9.9 to a certain memory location by using the variables dword_405020, dword_405024, dword_405028 for variables x, y and z and dword_405030, dword_405034 for variable 'value'. In the assembly code, there is no math involved at all, so we really can't be sure if the structure is involved or not. The way we see it, the program references a few global variables rather the members of the structure.

Local Structures

First let's present the C++ program that allocates the structure locally and declares its members. Basically, the program is the same as with the global structures, except that the structure is declared locally; all the rest is the same. The whole C++ program is as follows:

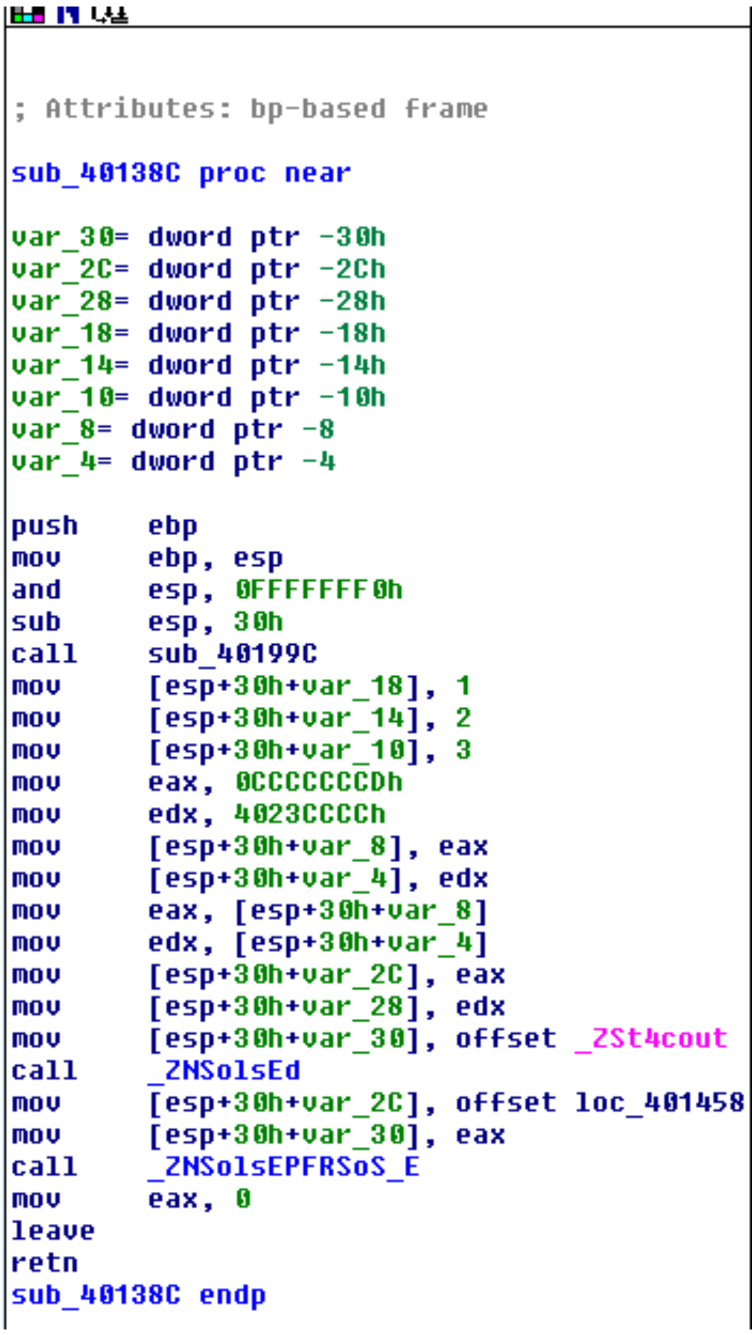
```
#include <iostream>

struct s {
  int x;
  int y;
  int z;
  double value;
};

int main(int argc, char **argv) {
  struct s mys;
  mys.x = 1;
  mys.y = 2;
  mys.z = 3;
  mys.value = 9.9;

  std::cout << mys.value << std::endl;

  return 0;
}
```

We can see that we're first declaring the structure with four members: variable x, y and z that are of type int and variable 'value' that is of type double. We can copy the program to Windows executable, compiling it and running it. We can see that on the picture below:



```
C:\Documents and Settings\eleanor\Desktop>g++ array5.cpp -o array5

C:\Documents and Settings\eleanor\Desktop>array5.exe
9.9

C:\Documents and Settings\eleanor\Desktop>_
```

Okay, so the program works as expected, because it outputs the number 9.9. But we're interested in the disassembled version of the program that we can obtain very quickly by opening up the executable in Ida Pro and finding the appropriate section of the executable. The disassembly listing can be seen below:

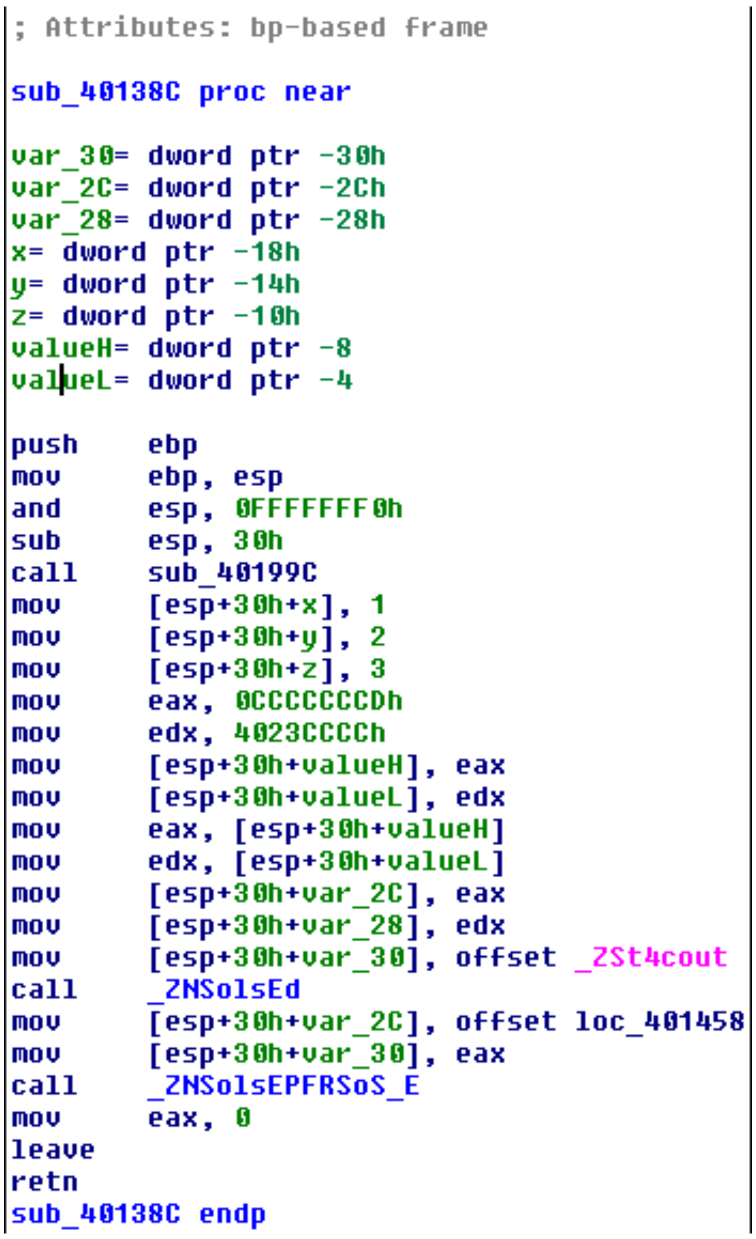


```
; Attributes: bp-based frame

sub_40138C proc near

var_30= dword ptr -30h
var_2C= dword ptr -2Ch
var_28= dword ptr -28h
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 30h
call    sub_40199C
mov     [esp+30h+var_18], 1
mov     [esp+30h+var_14], 2
mov     [esp+30h+var_10], 3
mov     eax, 0CCCCCCCDh
mov     edx, 4023CCCCh
mov     [esp+30h+var_8], eax
mov     [esp+30h+var_4], edx
mov     eax, [esp+30h+var_8]
mov     edx, [esp+30h+var_4]
mov     [esp+30h+var_2C], eax
mov     [esp+30h+var_28], edx
mov     [esp+30h+var_30], offset _ZSt4cout
call    _ZNSolsEd
mov     [esp+30h+var_2C], offset loc_401458
mov     [esp+30h+var_30], eax
call    _ZNSolsEPFRSoS_E
mov     eax, 0
leave
retn
sub_40138C endp
```

In the disassembly function, we can quickly figure out that we're using 0x30 bytes for local variables, which is the fact that we're declaring the structure locally. In the previous example where we declared the structure globally, we only used 0x10 bytes on the stack for local variables.

We can also see that this time, we're now assigning the values 1, 2, 3 and 9.9 to different global variables inside the assembly, yet we're actually using the stack pointer ESP with the right offsets to access certain members of the structure. The x variable from the C++ code lies at the address [esp+30h+var_18], which means that the local variable var_18 is used to reference the x member of the structure. The same goes for other members, where var_14 is used for member y, var_10 is used for member z and var_8 and var_4 are used for member 'value'. This gives us the picture that the function is using different local variables to hold the values assigned to them, but in reality we're defining the members of the previously defined structure 's'.

When we know how a certain function uses a structure, we can rename the local variables to define the structure more clearly. This also presents another useful feature of Ida: renaming the variables automatically generated by Ida itself. The disassembly of the renamed local variables could look like the picture below:



```
; Attributes: bp-based frame

sub_40138C proc near

var_30= dword ptr -30h
var_2C= dword ptr -2Ch
var_28= dword ptr -28h
x= dword ptr -18h
y= dword ptr -14h
z= dword ptr -10h
valueH= dword ptr -8
valueL= dword ptr -4

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 30h
call    sub_40199C
mov     [esp+30h+x], 1
mov     [esp+30h+y], 2
mov     [esp+30h+z], 3
mov     eax, 0CCCCCCCDh
mov     edx, 4023CCCCh
mov     [esp+30h+valueH], eax
mov     [esp+30h+valueL], edx
mov     eax, [esp+30h+valueH]
mov     edx, [esp+30h+valueL]
mov     [esp+30h+var_2C], eax
mov     [esp+30h+var_28], edx
mov     [esp+30h+var_30], offset _ZSt4cout
call    _ZNSolsEd
mov     [esp+30h+var_2C], offset loc_401458
mov     [esp+30h+var_30], eax
call    _ZNSolsEPFRSoS_E
mov     eax, 0
leave
retn
sub_40138C endp
```

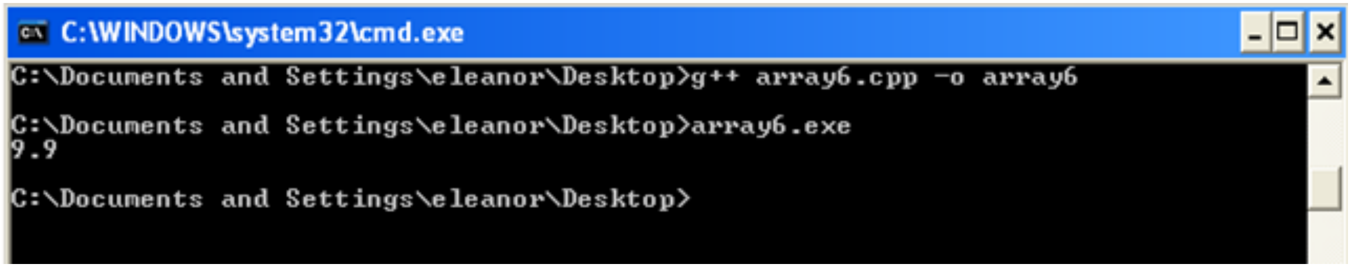
Notice that the local variables that used the offset into the structure are not renamed to define their real members x, y, z and 'value'? This can be a valuable help if we would like to share our work with others; maybe putting a few comments in there wouldn't be such a bad idea.

Heap Structures

Heap structures are basically the same as local or global structures, except that they are defined on heap. I guess we should first present the program written in C++ that does exactly that: it allocates the structure on the heap and then allocates certain values to its members and prints the value stored in memory 'value'. Such a C++ code can be seen below:

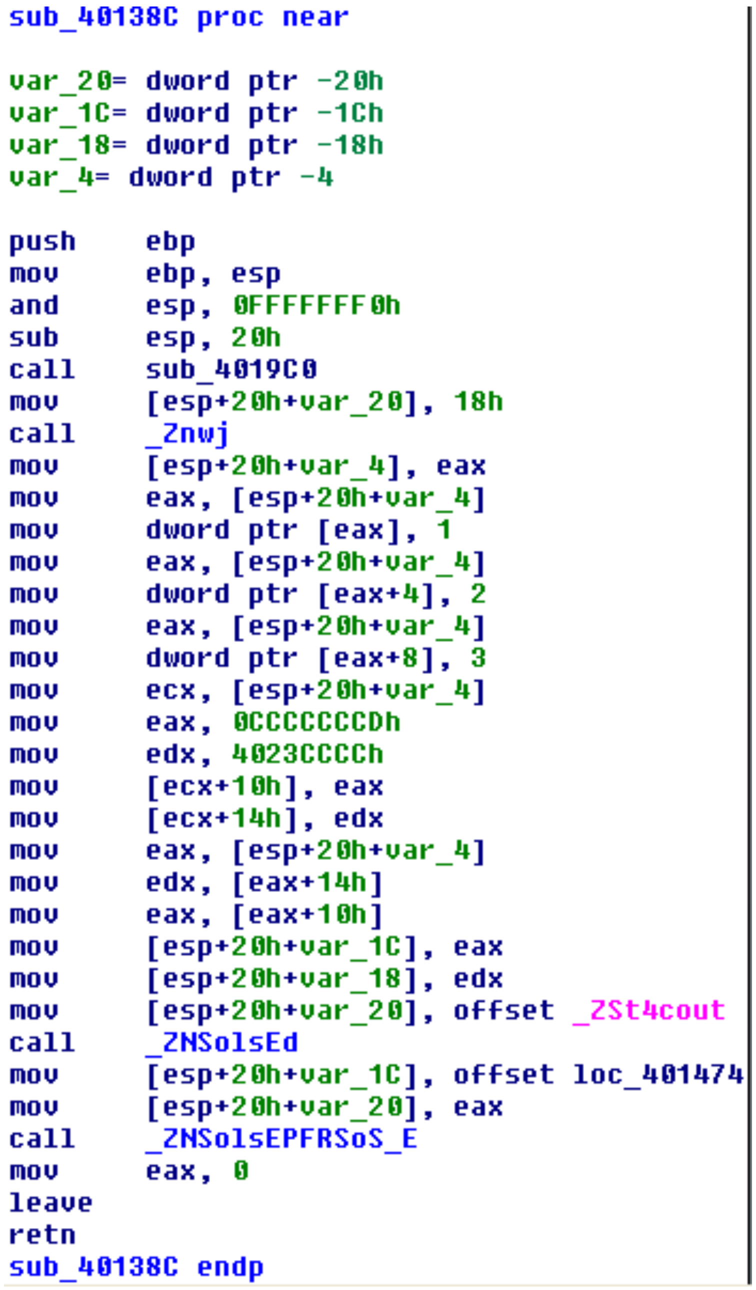
```
#include <iostream>

 struct s {
 int x;
 int y;
 int z;
 double value;
 };

 int main(int argc, char **argv) {
 s *mys = new s;
 mys-&gt;x = 1;
 mys-&gt;y = 2;
 mys-&gt;z = 3;
 mys-&gt;value = 9.9;

 std::cout << mys-&gt;value << std::endl;

 return 0;
 }
```

Upon compiling and running the example code above, the program will print the value 9.9 as we can see on the picture below:



```
C:\Documents and Settings\eleanor\Desktop>g++ array6.cpp -o array6

C:\Documents and Settings\eleanor\Desktop>array6.exe
9.9

C:\Documents and Settings\eleanor\Desktop>
```

When we load the program with Ida Pro, we can quickly find the relevant code the above program was compiled into. The assembly version of the above program can be seen on the picture below:



```
sub_40138C proc near

var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_4= dword ptr -4

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 20h
call    sub_4019C0
mov     [esp+20h+var_20], 18h
call    _Znwj
mov     [esp+20h+var_4], eax
mov     eax, [esp+20h+var_4]
mov     dword ptr [eax], 1
mov     eax, [esp+20h+var_4]
mov     dword ptr [eax+4], 2
mov     eax, [esp+20h+var_4]
mov     dword ptr [eax+8], 3
mov     ecx, [esp+20h+var_4]
mov     eax, 0CCCCCCCDh
mov     edx, 4023CCCCh
mov     [ecx+10h], eax
mov     [ecx+14h], edx
mov     eax, [esp+20h+var_4]
mov     edx, [eax+14h]
mov     eax, [eax+10h]
mov     [esp+20h+var_1C], eax
mov     [esp+20h+var_18], edx
mov     [esp+20h+var_20], offset _ZSt4cout
call    _ZNSolsEd
mov     [esp+20h+var_1C], offset loc_401474
mov     [esp+20h+var_20], eax
call    _ZNSolsEPFRSoS_E
mov     eax, 0
leave
retn
sub_40138C endp
```

Notice that we're first calling the Znwj function that equals the new function in C++. That function creates a new struct on the heap and stores the pointer to the structure in eax, which we're writing to the address on stack [esp+20h+var_4]. Afterwards, we're using this pointer to get access to various structure members by using the appropriate offset into the structure: [eax], [eax+4], [eax+8], [eax+10] and [eax+14]. We're also passing the 0x18 constant to the new function, which means that the struct's size is 0x18 (24 bytes).

Defining Structures Manually in Ida

In the preceding examples we saw how the structures from C++ were translated into assembly code. Let's summarize how the structure members were accessed in each of the three examples. When we declared the structure globally, the structure was accessed as follows:

```
mov     ds:dword_405020, 1
mov     ds:dword_405024, 2
mov     ds:dword_405028, 3
```

When we declared the structure locally, the structure was accessed as follows:

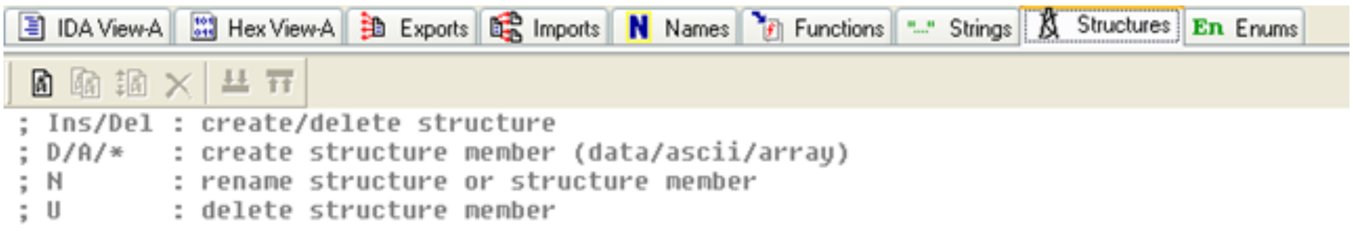
```
mov     [esp+30h+var_18], 1
mov     [esp+30h+var_14], 2
mov     [esp+30h+var_10], 3
```

When we declared the structure on the heap, the structure was accessed as follows:

```
mov     eax, [esp+20h+var_4]
mov     dword ptr [eax], 1
mov     dword ptr [eax+4], 2
mov     dword ptr [eax+8], 3
```

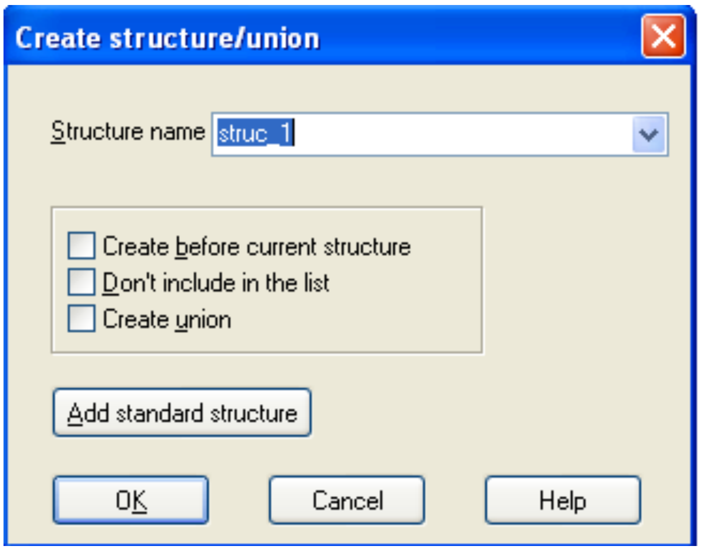
We can see that in the second and third case, we're using offsets to access certain members of a structure. We should tell Ida that we're dealing with a structure since Ida can only detect the use of a known structure by itself, but it certainly can't detect using the custom structure as we did in the above cases. We can open the Structures window by going to View – Open Subviews – Structures to see if Ida has detected the use of any structure in the program.

Currently there are no structures in this executable as we can see below:



```
; Ins/Del  : create/delete structure
; D/A/*    : create structure member (data/ascii/array)
; N        : rename structure or structure member
; U        : delete structure member
```

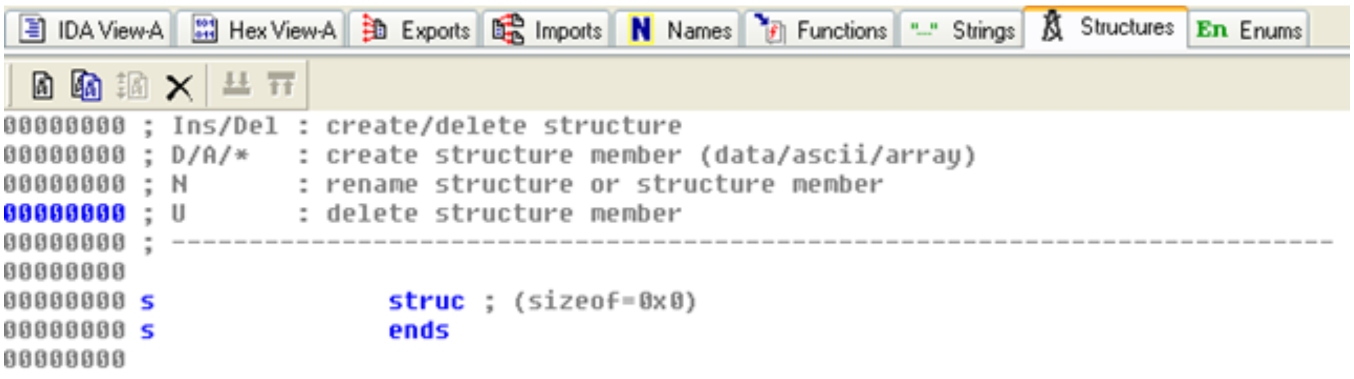
There are some comments presented in the structures view that informs us of how we can use the structures window. To create a structure, we can press the Insert key, while the Del key deletes a structure. We want to press the Insert key to insert a new structure. Upon doing that, the following dialog box will pop-up:


Create structure/union

Structure name: struc_1

- [ ] Create before current structure
- [ ] Don't include in the list
- [ ] Create union

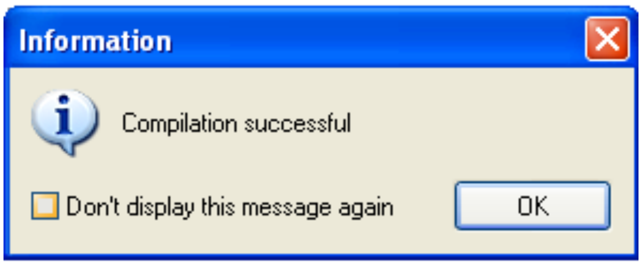
We need to enter the name of the structure, which is only the letter 's' and press OK. A new empty structure will be added to the Structures windows as can be seen on the picture below:



```
00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*   : create structure member (data/ascii/array)
00000000 ; N       : rename structure or structure member
00000000 ; U       : delete structure member
00000000 ; ---------------------------------------------------------------------------
00000000
00000000 s                struc ; (sizeof=0x0)
00000000 s                ends
00000000
```

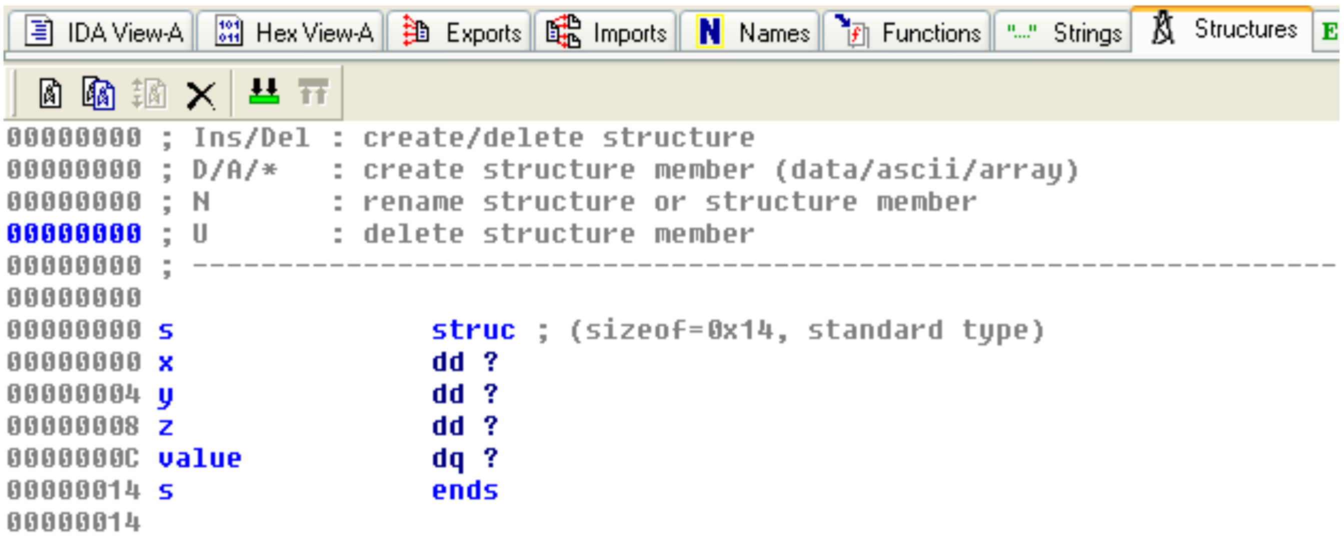
To add members to the data structure, we must position our cursor to where we want the structure to be and press the letter ‘d’. We must then press the letter ‘d’ as long as the added member isn’t the required size as it should be. Afterwards, we can right-click the name, which is by default field_0 and change its name to something else. Using that approach, we must add all members of a certain structure, which in our case are the variables x, y, z and ‘value’. We must also ensure the proper alignment of all the fields in the array. At the end, we can even collapse the structure with the minus ‘-‘ sign to represent it in one line; the opposite operation of that is to expand the structure by pressing the plus ‘+’ sign.

There is an easier way to create a structure in Ida – we can import the structure written in C/C++ programming language itself by importing the header file into Ida. We can do that by first creating the header file, which will contain only the defined structure itself and nothing else. Then we need to go to File – Load File – Parse C Header File and choose the created header file. Ida will parse and import it and then display the following notification window:



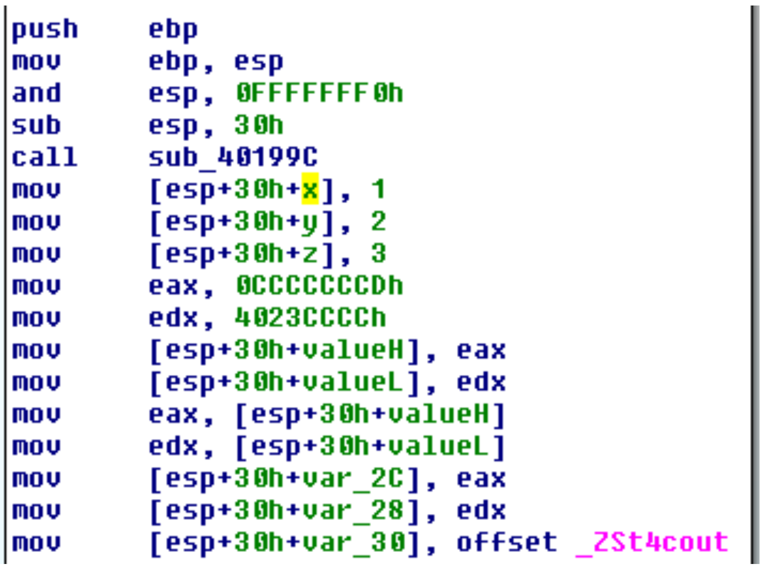
Compilation successful

The window tells us that the structure was successfully imported. After that, we should go to the Structures window and Insert a new structure with the same name as what we imported from the header file. This will actually add the structure among all the structures in the executable. We can see the structure ‘s’ being added to the structure window below:



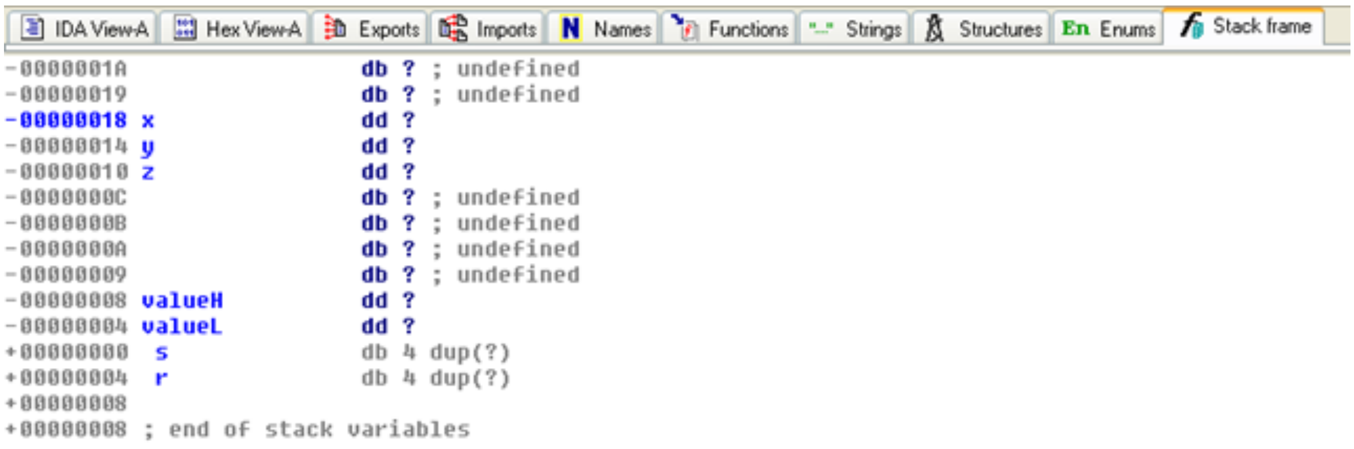
```
00000000 ; Ins/Del  : create/delete structure
00000000 ; D/A/*    : create structure member (data/ascii/array)
00000000 ; N        : rename structure or structure member
00000000 ; U        : delete structure member
00000000 ; ----------------------------------------------------------------
00000000
00000000 s               struc ; (sizeof=0x14, standard type)
00000000 x               dd ?
00000004 y               dd ?
00000008 z               dd ?
0000000C value           dq ?
00000014 s               ends
00000014
```

To use the structure in the disassembly listing, we have to double click on the offset that is being used to reference different members of the structure. Let’s take a look at the following program disassembly:



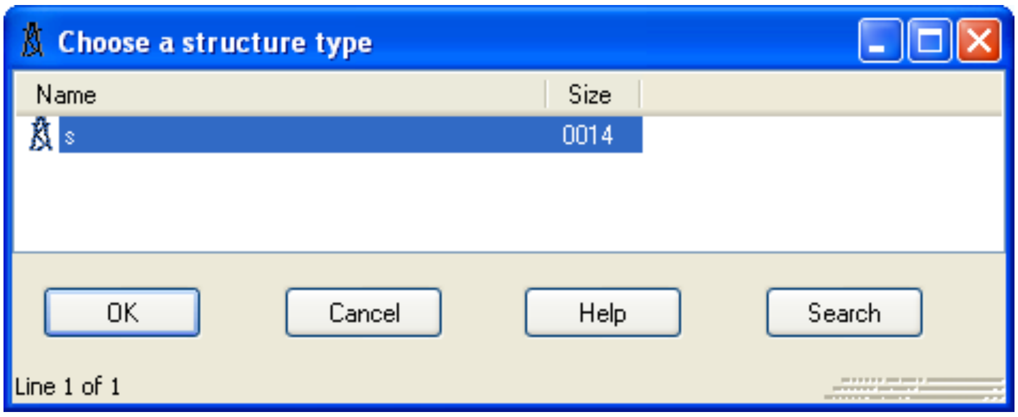
```
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 30h
call    sub_40199C
mov     [esp+30h+x], 1
mov     [esp+30h+y], 2
mov     [esp+30h+z], 3
mov     eax, 0CCCCCCCDh
mov     edx, 4023CCCCh
mov     [esp+30h+valueH], eax
mov     [esp+30h+valueL], edx
mov     eax, [esp+30h+valueH]
mov     edx, [esp+30h+valueL]
mov     [esp+30h+var_2C], eax
mov     [esp+30h+var_28], edx
mov     [esp+30h+var_30], offset _ZSt4cout
```

In the picture above, it’s already evident that we renamed the offsets that reference different members of the structure into x, y, z as valueH and valueL. After that, we should double-click on the variable x to be thrown at an actual stack frame memory address (this can be a memory allocated in any section) as follows:



```
-0000001A                 db ? ; undefined
-00000019                 db ? ; undefined
-00000018 x               dd ?
-00000014 y               dd ?
-00000010 z               dd ?
-0000000C                 db ? ; undefined
-0000000B                 db ? ; undefined
-0000000A                 db ? ; undefined
-00000009                 db ? ; undefined
-00000008 valueH          dd ?
-00000004 valueL          dd ?
+00000000  s              db 4 dup(?)
+00000004  r              db 4 dup(?)
+00000008
+00000008 ; end of stack variables
```

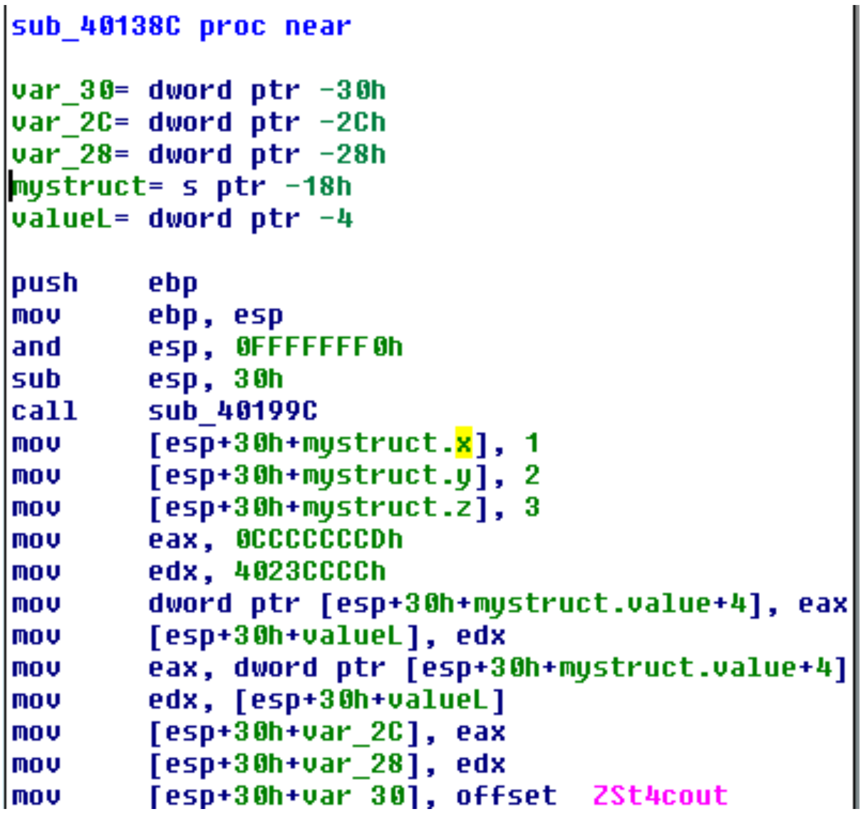
Then we should select the first variable of the structure, in this case variable x, and select Edit – Struct Var. This will display a list of known structures within the executable. In our case, only the imported structure 's' is known, as we can see on the picture below:



Name	Size
s	0014

Line 1 of 1

The structure will be applied to the current address and will consume as many bytes as the size of the structure. This is why we must always select the first member of the structure, because the structure will be applied to that memory address and its corresponding higher memory addresses. After the structure is applied to the current memory address, the disassembly will look like the picture below:



```
sub_40138C proc near

var_30= dword ptr -30h
var_2C= dword ptr -2Ch
var_28= dword ptr -28h
mystruct= s ptr -18h
valueL= dword ptr -4

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 30h
call    sub_40199C
mov     [esp+30h+mystruct.x], 1
mov     [esp+30h+mystruct.y], 2
mov     [esp+30h+mystruct.z], 3
mov     eax, 0CCCCCCCDh
mov     edx, 4023CCCCh
mov     dword ptr [esp+30h+mystruct.value+4], eax
mov     [esp+30h+valueL], edx
mov     eax, dword ptr [esp+30h+mystruct.value+4]
mov     edx, [esp+30h+valueL]
mov     [esp+30h+var_2C], eax
mov     [esp+30h+var_28], edx
mov     [esp+30h+var_30], offset _ZSt4cout
```

We can see that it was worth it, because now the disassembly view is much clearer and easier to read. Notice that we now have a local variable named mystruct that is used later by the function to reference different members inside it.

Conclusion

We've seen how structures are reverse engineered in Ida debugger and how to recognize them. But what is more important is the fact that we've looked at how to import the structures in Ida and apply them to memory locations, which automatically updates the disassembly view to make it more readable and easier to understand.

We should also keep in mind that Ida applies known structures from various system libraries to the executable by default when being analyzed. Usually, different structures are used in different API functions that are part of the system. All the recognized structures will also be added to the structures window, which we can use throughout the program analysis.

References

[1] Chris Eagle, The IDA Pro Book: The unofficial guide to the world's most popular disassembler.