7.1.—Reconocer la estructura de datos que se utiliza

Mientras que los tipos de dato primitivos se ajustan con naturalidad al tamaño de los registros de la CPU o a los operandos de una instrucción, los tipos de datos compuestos como pueden ser arrays y estructuras, requieren secuencias de instrucciones más complejas a fin de acceder a los elementos de datos que contienen. Pues bien, antes de estudiar las características que IDA nos proporciona para mejorar la comprensión del código utilizado por los tipos de dato complejos, antes necesitamos repasar cómo se nos mostrará el código.

7.1.1.—Acceso a un elemento de un array (conjunto)

Los conjuntos son la estructura más simple desde el punto de vista del esquema de memoria. Normalmente los conjuntos son bloques contiguos de memoria los cuales contienen, consecutivamente, elementos del mismo tipo de dato. Como ya vimos, calcular el tamaño de un conjunto es fácil, es el **producto del número total de elementos del conjunto por el tamaño de cada elemento**. Utilizando la notación C, el número mínimo de bytes utilizados por el siguiente array: **int array_demo [100]**, se calcularía como **int bytes = 100 * sizeof (int).**

Para tener acceso individualmente a los elementos de dicho array, lo haríamos variando el valor del índice, el cual podría ser una variable o una constante, como mostramos a continuación:

```
array_demo[20] = 15;          //índice fijo en el array

for (int i = 0; i < 100; i++)
        {
                array_demo [i] = i;
        }
```

Asumiremos, en el ejemplo, que **sizeof (int)** es **4 bytes**, En la primera forma de acceder al conjunto, **array_demo[20] = 15**, sabremos que el valor entero estará en la posición **80 bytes** desde el inicio del conjunto, mientras que en la segunda forma de acceso se realizará por sucesivos offsets del **0, 4, 8, .. , 96 bytes** del conjunto. El offset en la primera forma de acceso puede calcularse y compilar una vez como **20 * 4**. En la mayoría de los casos la segunda forma de acceder al conjunto, el offset debe ser calculado en tiempo de ejecución ya que el valor del contador del bucle**, i,** no está fijado el efectuar la compilación. De esta forma cada pasada por el bucle, se deberá calcular el producto de **i * 4** para determinar el offset exacto en el conjunto. De momento para finalizar diremos, que la forma en la cual se accede a un elemento del conjunto, dependerá no sólo del tipo de índice usado sino también por la forma en que esté distribuido el conjunto en el espacio de memoria del programa. Veámoslo.

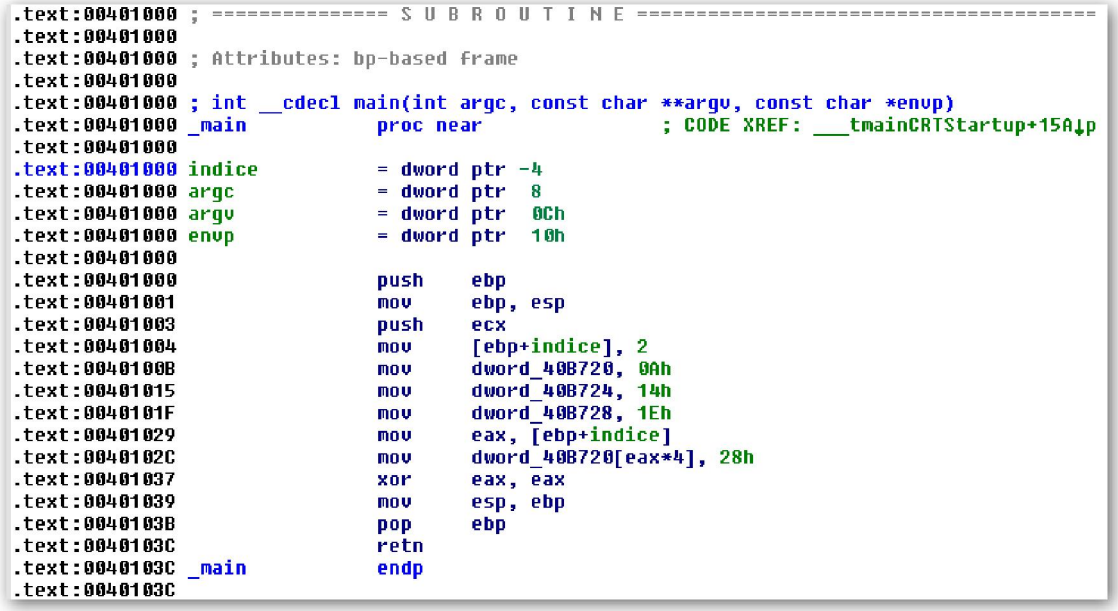
7.1.1.1.—Conjuntos distribuidos globalmente

Cuando un conjunto se distribuye en el área de datos globales de un programa, por ejemplo en la sección **.data** o **.bss**, cuando se compila la dirección base del conjunto es conocida por el compilador. La dirección base fija, hace posible que el compilador pueda calcular la dirección fija para cualquier elemento del conjunto ya que se accederá a él con un índice fijo. Veamos el siguiente programa en el cual se accede a un conjunto global utilizando las dos formas, con offset fijo y variable:

```
int global_array [3];

int main ( )
      {
            int indice = 2;
            global_array [0] = 10;
            global_array [1] = 20;
            global_array [2] = 30;
            global_array [indice] = 40;
      }
```

El desensamblado del programa sería el siguiente:



Aunque este programa tiene solamente una variable global, vemos que en las líneas

parecen indicar que existen tres variables globales. El cálculo del offset (indice * 4) en

,es lo único que nos muestra el indicio de la existencia de un conjunto global llamado **dword_40B720**, sin embargo este es el mismo nombre de la variable global que está en

Basándonos en los **dummy names** asignados por IDA, sabemos que el conjunto global está compuesto por **12 bytes** que empiezan en la dirección **0040B720**. Durante el

proceso de compilación, el compilador ha utilizado índices fijos (**0,1,2**) para calcular las direcciones reales de los elementos, **0040B720, 0040B724 y 0040B728**, del conjunto, las cuales son referenciadas utilizando variables globales en:

```
.text:0040100B                mov     dword_40B720, 0Ah
.text:00401015                mov     dword_40B724, 14h
.text:0040101F                mov     dword_40B728, 1Eh
```

Utilizando las operaciones para formatear conjuntos, explicadas en la parte 6 (**Edit > Array**), **dword_40B720** puede ser formateado como un conjunto de tres elementos lo cual nos produce las líneas de desensamblado alternativo mostradas a continuación. Observa que este particular formato remarca la utilización de offset en el conjunto:

```
.text:0040100B                mov     dword_40B720, 0Ah
.text:00401015                mov     dword_40B720+4, 14h
.text:0040101F                mov     dword_40B720+8, 1Eh
```

Hay dos cosas a observar en este ejemplo. Primera, cuando se utilizan índices constantes para el acceso a elementos de conjuntos globales, dichos elementos, aparecerán en el desensamblado como variables globales. En otras palabras, el desensamblado no mostrará ninguna evidencia de que existe un conjunto.
Segunda la utilización de índices variables nos llevará al principio del conjunto ya que se nos mostrará la dirección base, como vemos en la línea abajo, en el momento que el

```
.text:0040102C                mov     dword_40B720[eax*4], 28h
```

offset es calculado para añadirlo a ella y calcular la ubicación real a acceder. El cálculo realizado en, **0040102C**, nos proporciona una información significativa respecto al conjunto. Ya que si observamos la cantidad por la cual el índice del conjunto es multiplicado, **4** en este caso, sabremos el tamaño, pero no el tipo, de un elemento del conjunto.

7.1.1.2.—Conjuntos distribuidos en la pila

¿En qué cambia el acceso a un conjunto que está distribuido en la pila? Por instinto, podríamos pensar que la diferencia está en que el compilador no puede saber ninguna dirección absoluta en el momento de la compilación, seguramente para acceder a él se utilizarán índices constantes los cuales requerirán algún tipo de cálculo en tiempo de ejecución. En la práctica, sin embargo, los compiladores tratan a los conjuntos distribuidos en la pila casi de la misma forma que a los conjuntos distribuidos globalmente.

Consideremos el siguiente programa el cual utiliza un conjunto distribuido en la pila:

```
int main()
      {
         int stack_array[3];
         int indice = 2;
         stack_array[0] = 10;
         stack_array[1] = 20;
         stack_array[2] = 30;
         stack_array[indice] = 40;
      }
```

Las direcciones en las cuales **stack_array** estará distribuido son desconocidas en la compilación, pues no es posible para el compilador precalcular la dirección de **stack_array [1]** en el momento de compilar como sucedía en el ejemplo del conjunto global. Examinemos el listado de desensamblado de dicha función, veamos cómo se accede a los conjuntos distribuidos en la pila:

```
.text:00401000 ; =============== S U B R O U T I N E =======================================
.text:00401000
.text:00401000 ; Attributes: bp-based frame
.text:00401000
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char *envp)
.text:00401000 _main           proc near               ; CODE XREF: ___tmainCRTStartup+15A↓p
.text:00401000
.text:00401000 var_10          = dword ptr -10h
.text:00401000 var_C           = dword ptr -0Ch
.text:00401000 var_8           = dword ptr -8
.text:00401000 indice          = dword ptr -4
.text:00401000 argc            = dword ptr  8
.text:00401000 argv            = dword ptr  0Ch
.text:00401000 envp            = dword ptr  10h
.text:00401000
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 sub     esp, 10h
.text:00401006                 mov     [ebp+indice], 2
.text:0040100D                 mov     [ebp+var_10], 0Ah
.text:00401014                 mov     [ebp+var_C], 14h
.text:0040101B                 mov     [ebp+var_8], 1Eh
.text:00401022                 mov     eax, [ebp+indice]
.text:00401025                 mov     [ebp+eax*4+var_10], 28h
.text:0040102D                 xor     eax, eax
.text:0040102F                 mov     esp, ebp
.text:00401031                 pop     ebp
.text:00401032                 retn
.text:00401032 _main           endp
```

Al igual que en el ejemplo del conjunto global, la función aparece teniendo tres variables, **var_10, var_C y var_8**, mejor dicho un conjunto de tres enteros. Basándonos en las constantes utilizadas en las líneas siguientes, sabemos que lo que parece ser

```
.text:0040100D                 mov     [ebp+var_10], 0Ah
.text:00401014                 mov     [ebp+var_C], 14h
.text:0040101B                 mov     [ebp+var_8], 1Eh
```

referencias a las variables locales son en realidad referencias a los tres elementos del conjunto **stack_array** cuyo primer elemento estará en la variable local **var_10** con la dirección de memoria más baja.

Para comprender como el compilador resuelve las referencias a los otros elementos del conjunto, consideremos lo que el compilador examina cuidadosamente al tratar con la referencia **stack_array [1]**, cuyo elemento está distribuido en **4 bytes** del conjunto, cuatro bytes a partir de la ubicación de **var_10**. En el **stack frame**, el compilador ha elegido distribuir a **stack_array** desde **ebp – 0x10**. El compilador entiende que **stack_array [1]** está situado en la posición **ebp – 0x10 + 4**, lo cual simplifica como **ebp – 0x0C**. El resultado es que IDA muestra esto como una referencia a una variable local. Por lo tanto el efecto final es similar al de los conjuntos globales, la utilización de índices constantes tenderá a ocultar la presencia de un conjunto distribuido en la pila. Solamente el acceso al conjunto mostrado en la línea siguiente, nos da indicios que de

hecho **var_10** es el primer elemento del conjunto en vez de una variable con valor hexa. Además, la línea de desensamblado anterior nos facilita el poder saber, que el tamaño de los elementos del conjunto es de **4 bytes**.

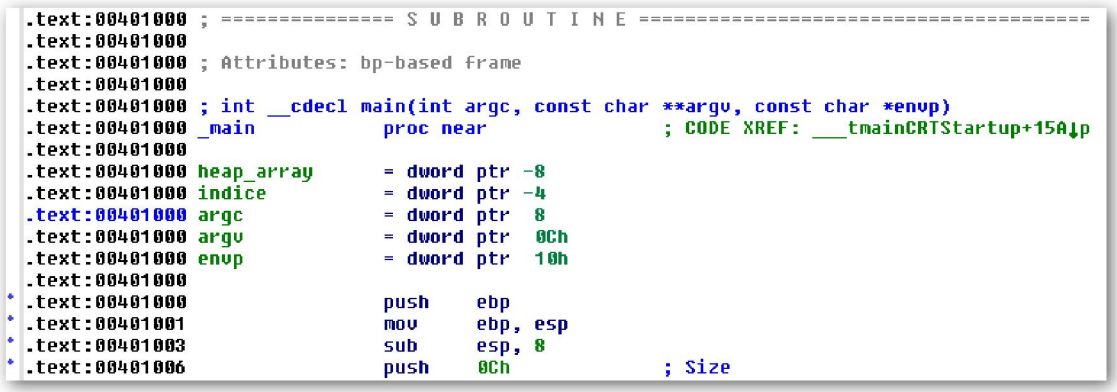
Resumiendo podemos decir, que tanto los conjunto distribuidos en pila como los globales son tratados similarmente por los compiladores. Sin embargo en el ejemplo del stack_array obtenemos más información la cual podemos deducir del desensamblado. Basándonos en la ubicación de **indice** en la pila, es posible llegar a a la conclusión que el conjunto que empieza con **var_10** no tiene más que **tres** elementos, ya que de otra forma se sobrescribiría **indice**. Si te dedicas a realizar explotaciones, esto puede ser muy útil para determinar exactamente qué cantidad de datos podrás encajar en un conjunto antes de desbordarlo (overflow) y empezar a corromper los datos que le siguen.

7.1.1.3.—Conjuntos distribuidos en el Heap (almacenamiento de pila)

Los conjuntos distribuidos en el **heap**, son distribuidos utilizando una función de distribución dinámica de memoria como **malloc** en C o **new** en C++. Desde la perspectiva del compilador, la principal diferencia al tratar con un conjunto distribuido en heap es que el compilador debe generar todas las referencias al conjunto basándose en el valor retornado por la función de distribución de memoria. Para poder compararlo con los anteriores, vamos a estudiar este pequeño programa, el cual distribuye un conjunto en el heap del programa:

```
int main()
      {
         int *heap_array = (int*)malloc(3 * sizeof(int));
         int indice = 2;
         heap_array[0] = 10;
         heap_array[1] = 20;
         heap_array[2] = 30;
         heap_array[indice] = 40;
      }
```

Estudiando su desensamblado correspondiente, veremos las similitudes o diferencias con los desensamblados anteriores.



```
.text:00401008                call    _malloc
.text:0040100D                add     esp, 4
.text:00401010                mov     [ebp+heap_array], eax
.text:00401013                mov     [ebp+indice], 2
.text:0040101A                mov     eax, [ebp+heap_array]
.text:0040101D                mov     dword ptr [eax], 0Ah
.text:00401023                mov     ecx, [ebp+heap_array]
.text:00401026                mov     dword ptr [ecx+4], 14h
.text:0040102D                mov     edx, [ebp+heap_array]
.text:00401030                mov     dword ptr [edx+8], 1Eh
.text:00401037                mov     eax, [ebp+indice]
.text:0040103A                mov     ecx, [ebp+heap_array]
.text:0040103D                mov     dword ptr [ecx+eax*4], 28h
.text:00401044                xor     eax, eax
.text:00401046                mov     esp, ebp
.text:00401048                pop     ebp
.text:00401049                retn
.text:00401049 _main          endp
```

La dirección de inicio del conjunto, retornada por **malloc** en el registro **EAX**, es guardada en la variable local **heap_array**. En este ejemplo, a diferencia de los anteriores, cada acceso al conjunto se inicia leyendo el contenido de **heap_array** para obtener la dirección base del conjunto antes de poder añadir cualquier offset, para poder calcular la dirección correcta de cualquier elemento del conjunto. Las referencias a **heap_array [0], heap_array [1] y heap_array [2]** requieren offset de **0,4 y 8** bytes respectivamente. Como se puede ver en las líneas siguientes

```
.text:0040101D                mov     dword ptr [eax], 0Ah
```

```
.text:00401026                mov     dword ptr [ecx+4], 14h
```

```
.text:00401030                mov     dword ptr [edx+8], 1Eh
```

La operación que más se asemeja a los ejemplos anteriores es la referencia a **heap_array [indice]** en la línea siguiente, en la cual se muestra que el offset al conjunto, se continúa calculando con la multiplicación del índice del conjunto por el tamaño del elemento del conjunto.

```
.text:0040103D                mov     dword ptr [ecx+eax*4], 28h
```

Una caracteristica particular de los conjuntos distribuidos en el heap es que, cuando puede determinarse el tamaño total del conjunto y el tamaño de cada elemento, es fácil calcular el numero de elementos del conjunto. En estos conjuntos el parámetro pasado a la función de distribución de memoria, **0x0C** pasado a **malloc** en la línea 00401006,

```
.text:00401006                push    0Ch             ; Size
```

representa el número total de bytes distribuidos en el conjunto. Dividiendo este tamaño por el tamaño de un elemento, **4 bytes** en este ejemplo como hemos observado en los offsets de las líneas **0040101D, 00401026 y 00401030**, nos dará como resultado el número de elementos del conjunto. En el ejemplo, han sido distribuidos tres elementos.

Para finalizar podemos llegar a la conclusión respecto a la utilización de conjuntos, que son fáciles de reconocer cuando se utilizan índices variable como índice del conjunto. La operación de acceso al conjunto requiere que el índice se escale con el tamaño de un elemento del conjunto antes de añadir el offset resultante a la dirección base del conjunto. Por desgracia, como veremos en la siguiente sección, cuando se utiliza un índice constante para acceder a los elementos del conjunto, no nos sugiere la presencia

de un conjunto y se parece más al código utilizado para acceder a los elementos de una estructura.

7.1.2.—Acceso a un elemento de estructura

Las **structs** de estilo C, referidas genericamente como estructuras, son recopilaciones de datos heterogeneos que permiten la agrupación de elementos de distintos tipos de dato en un sólo tipo de dato compuesto. Una caracteristica principal que distingue a las estructuras es que los campos de datos en una estructura son accedidos por nombre en vez de por índice, como ocurre en los conjuntos. Por desgracia, los nombres de campos son convertidos en offset numéricos por el compilador, así pues cuando miremos un desensamblado, la vista de acceso a un campo de estructura se parecerá al acceso de elementos de un conjunto que utiliza índices constantes.

Cuando un compilador se encuentra con la definición de una estructura, el compilador mantiene la ejecución total del número de bytes consumidos por los campos de la estructura a fin de determinar el offset de cada campo que resida dentro de la estructura. La siguiente definición de estructura se utilizará para los próximos ejemplos:

```
struct ch8_struct
      {
                          //Size     Minimum offset     Default offset
         int field1;      //  4            0                  0
         short field2;    //  2            4                  4
         char field3;     //  1            6                  6
         int field4;      //  4            7                  8
         double field5;   //  8            11                 16
      };                  //Minimum total size: 19   Default size: 24
```

El espacio mínimo para distribuir una estructura se determina con la suma del espacio requerido para distribuir cada campo en la estructura. Sin embargo asumamos que nunca un compilador utilizará el espacio mínimo necesario para distribuir una estructura. Por defecto, los compiladores tratan de alinear los campos de estructura en las direcciones de memoria teniendo en cuenta la eficiencia para leer y escribir en esos campos. Por ejemplo, los campos enteros de 4 bytes serán alineados con offset divisibles por cuatro, mientras que los de 8 bytes serán alineados con offset divisibles por ocho. Dependiendo de la composición de la estructura, las necesidades de alineación puede requerir la inserción de bytes de relleno, causando que el tamaño real de la estructura pase a ser mayor que la suma de sus campos. Los offset por defecto y el tamaño resultante de la estructura del ejemplo anterior puede verse en la columna **Default offset**.

Las estructuras pueden ser comprimidas en el mínimo espacio requerido utilizando opciones del compilador que piden las alineaciones específicas de los elementos. Los compiladores **Microsoft Visual C/C++ y GNU gcc/g++**, reconocen el atributo **pack** como un medio de controlar el alineamiento de los campos de una estructura. El compilador **GNU** además, reconoce el atributo **packed** como medio para controlar el alineamiento de una estructura, con la base predominante en la estructura. Pedir un alineamiento de **1 byte** para los campos de una estructura, causa que el compilador ajuste la estructura en el mínimo espacio requerido. En nuestro ejemplo, esto produce los offset y el tamaño de estructura mostrados en la columna **Minimum Offset**. Tengamos en cuenta que algunas CPU realizan mejor la ejecución cuando los datos

están alineados según su tipo, mientras que en otras pueden generar excepciones si los datos no están alineados entre límites específicos.

Bien pues atendiendo a lo dicho hasta aquí, podemos empezar a echar una mirada a cómo se tratan las estructuras en el código compilado, lo haremos de forma comparativa con los conjuntos, al igual que estos el acceso a los elementos de una estructura se ejecuta sumando a la dirección base de la estructura el offset del elemento deseado. Sin embargo, mientras que en los conjuntos los offset pueden calcularse en tiempo de ejcución gracias a un valor de índice, esto es debido a que en el conjunto cada elemento tiene el mismo tamaño, en la estructura los offset deben de precalcularse y suministrarse al código compilado como offset fijos de la estructura, siendo casi idéntico a las referencias de conjunto que se hacen con índices constantes.

7.1.2.1.—Estructuras distribuidas globalmente

Al igual que los conjuntos distribuidos globalmente, las direcciones de las estructuras distribuidas globalmente se conocen al realizar la compilación. Esto permite al compilador calcular la dirección de cada elmento de la estructura cuando se compila y eliminar la necesidad de realizar cualquier cálculo matemático en tiempo de ejecución. Consideremos el siguiente programa para acceder a una estructura distribuida global:

```
struct global_struct;

int main()
      {
         global_struct.field1 = 10;
         global_struct.field2 = 20;
         global_struct.field3 = 30;
         global_struct.field4 = 40;
         global_struct.field5 = 50.0;
      }
```

Si el programa se compila con las opciones de alineamiento de estructura por defecto, podremos ver algo parecido a lo siguiente cuando se desensambla:

```
.text:00401000 ; =============== S U B R O U T I N E =======================================
.text:00401000
.text:00401000 ; Attributes: bp-based frame
.text:00401000
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char *envp)
.text:00401000 _main           proc near               ; CODE XREF: ___tmainCRTStartup+15A↓p
.text:00401000
.text:00401000 argc            = dword ptr  8
.text:00401000 argv            = dword ptr  0Ch
.text:00401000 envp            = dword ptr  10h
.text:00401000
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 mov     dword_40EA60, 0Ah
.text:0040100D                 mov     word_40EA64, 14h
.text:00401016                 mov     byte_40EA66, 1Eh
.text:0040101D                 mov     dword_40EA68, 28h
.text:00401027                 fld     ds:dbl_40B128
.text:0040102D                 fstp    dbl_40EA70
.text:00401033                 xor     eax, eax
.text:00401035                 pop     ebp
.text:00401036                 retn
.text:00401036 _main           endp
.text:00401036
```

Como podemos observar, este desensamblado no contiene ninguna operación matemática para acceder a los elementos de la estructura, podríamos pensar que el código fuente es corrupto, pero no es así ya que si lo fuera no se nos mostraría como

una estructura que se está utilizando. No existen operaciones matemáticas debido a que el compilador ha ejecutado todos los cálculos de los offset en la compilación, examinándo este programa parece referenciar a cinco variables locales, en vez de a cinco campos de una estructura. Con lo cual vemos una similitud con los conjuntos distribuidos globalmente que utilizan valores de índice constantes.

7.1.2.2.—Estructuras distribuidas en la pila

Al igual que los conjuntos distribuidos en la pila, ver sección 7.1.1.2. Las estructuras distribuidas en la pila tienen la misma dificultad para reconocerlas basándonos solamente en el esquema de la pila. Si modificamos el programa anterior para utilizar una estructura distribuida en la pila, declarado en **main**, tendremos el siguiente desensamblado:

```
.text:00401000 ; =============== S U B R O U T I N E =======================================
.text:00401000
.text:00401000 ; Attributes: bp-based frame
.text:00401000
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char *envp)
.text:00401000 _main           proc near               ; CODE XREF: ___tmainCRTStartup+15A↓p
.text:00401000
.text:00401000 var_18          = dword ptr -18h
.text:00401000 var_14          = word ptr -14h
.text:00401000 var_12          = byte ptr -12h
.text:00401000 var_10          = dword ptr -10h
.text:00401000 var_8           = qword ptr -8
.text:00401000 argc            = dword ptr  8
.text:00401000 argv            = dword ptr  0Ch
.text:00401000 envp            = dword ptr  10h
.text:00401000
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 sub     esp, 18h
.text:00401006                 mov     [ebp+var_18], 0Ah
.text:0040100D                 mov     [ebp+var_14], 14h
.text:00401013                 mov     [ebp+var_12], 1Eh
.text:00401017                 mov     [ebp+var_10], 28h
.text:0040101E                 fld     ds:dbl_40B128
.text:00401024                 fstp    [ebp+var_8]
.text:00401027                 xor     eax, eax
.text:00401029                 mov     esp, ebp
.text:0040102B                 pop     ebp
.text:0040102C                 retn
.text:0040102C _main           endp
```

Otra vez, nos fijamos en que no existen operaciones matemáticas para acceder a los campos de la estructura, ya que el compilador puede determinar los offset relativos de cada campo en el stack frame en la compilación. En este caso, nos aparece lo mismo, se están utilizando cinco variables en vez de una variable para conseguir el contenido de los cinco campos distintos. En realida **var_18** debe ser el inicio de una estructura de **24 byte (18h)**, y las otras variables de algún modo serán formateadas para reflejar que de hecho son los campos de la estructura.

7.1.2.3.—Estructuras distribuidas en el heap

Las estructuras distribuidas en el heap, además del tamaño de la estructura y el esquema de sus campos nos proporcionan más información. Cuando tenemos una estructura distribuida en el heap, el compilador no tiene ninguna alternativa al generar el código, de poder calcular el offset apropiado para poder acceder a un campo de la estructura. Esto es el resultado de desconocerse la dirección de la estructura en el momento de la compilación. En las estructuras distribuidas globalmente, el compilador es capaz de calcular una dirección fija de inicio. En las estructuras distribuidas en la pila, el compilador puede calcular un valor relativo fijo entre el inicio de la estructura y el

frame pointer incluido en el stack frame. Finalmente cuando una estructura ha sido distribuida en el heap, la única referencia a la estructura disponible por el compilador es el puntero a la dirección de inicio de la estructura.

Si modificamos otra vez nuestra estructura de ejemplo, para que utilice una estructura distribuida en el heap, tendremos el siguiente desensamblado. Similarmente al ejemplo de conjunto distribuido en el heap, sección 7.1.1.3, declaramos un puntero en **main** y le asignamos la dirección de un bloque de memoria lo suficientemente grande para que soporte nuestra estructura:

```
.text:00401000 ; =============== S U B R O U T I N E =======================================
.text:00401000
.text:00401000 ; Attributes: bp-based frame
.text:00401000
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char *envp)
.text:00401000 _main           proc near               ; CODE XREF: ___tmainCRTStartup+15A↓p
.text:00401000
.text:00401000 heap_struct     = dword ptr -4
.text:00401000 argc            = dword ptr  8
.text:00401000 argv            = dword ptr  0Ch
.text:00401000 envp            = dword ptr  10h
.text:00401000
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 push    ecx
.text:00401004                 push    18h             ; Size
.text:00401006                 call    _malloc
.text:0040100B                 add     esp, 4
.text:0040100E                 mov     [ebp+heap_struct], eax
.text:00401011                 mov     eax, [ebp+heap_struct]
.text:00401014                 mov     dword ptr [eax], 0Ah
.text:0040101A                 mov     ecx, [ebp+heap_struct]
.text:0040101D                 mov     word ptr [ecx+4], 14h
.text:00401023                 mov     edx, [ebp+heap_struct]
.text:00401026                 mov     byte ptr [edx+6], 1Eh
.text:0040102A                 mov     eax, [ebp+heap_struct]
.text:0040102D                 mov     dword ptr [eax+8], 28h
.text:00401034                 mov     ecx, [ebp+heap_struct]
.text:00401037                 fld     ds:dbl_40B128
.text:0040103D                 fstp    qword ptr [ecx+10h]
.text:00401040                 xor     eax, eax
.text:00401042                 mov     esp, ebp
.text:00401044                 pop     ebp
.text:00401045                 retn
.text:00401045 _main           endp
```

En este ejemplo distinto al distribuido global y de pila, vemos que somos capaces de discernir el tamaño exacto y el esquema de la estructura. El tamaño de la estructura nos es mostrado en la cantidad de memoria pedida por **malloc** que es de **24 bytes (18h)**.

```
.text:00401004                 push    18h             ; Size
```

La estructura contiene los siguientes campos con sus offset indicados:

Un campo dword (4-byte) con offset 0

```
.text:00401014                 mov     dword ptr [eax], 0Ah
```

Un campo Word (2-byte) con offset 4

```
.text:0040101D                 mov     word ptr [ecx+4], 14h
```

Un campo byte (1-byte) con offset 6

```
.text:00401026                 mov     byte ptr [edx+6], 1Eh
```

Un campo dword (4-byte) con offset 8

```
.text:0040102D                  mov     dword ptr [eax+8], 28h
```

Un campo qword (8-byte) con offset 16 (10h)

```
.text:0040103D                  fstp    qword ptr [ecx+10h]
```

El mismo programa compilado con la opción **pack** y con alineamiento de **1-byte** nos daría el siguiente desensamblado:

```
.text:00401000 ; =============== S U B R O U T I N E =======================================
.text:00401000
.text:00401000 ; Attributes: bp-based frame
.text:00401000
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char *envp)
.text:00401000 _main           proc near               ; CODE XREF: ___tmainCRTStartup+15A↓p
.text:00401000
.text:00401000 heap_struct     = dword ptr -4
.text:00401000 argc            = dword ptr  8
.text:00401000 argv            = dword ptr  0Ch
.text:00401000 envp            = dword ptr  10h
.text:00401000
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 push    ecx
.text:00401004                 push    13h             ; Size
.text:00401006                 call    _malloc
.text:0040100B                 add     esp, 4
.text:0040100E                 mov     [ebp+heap_struct], eax
.text:00401011                 mov     eax, [ebp+heap_struct]
.text:00401014                 mov     dword ptr [eax], 0Ah
.text:0040101A                 mov     ecx, [ebp+heap_struct]
.text:0040101D                 mov     word ptr [ecx+4], 14h
.text:00401023                 mov     edx, [ebp+heap_struct]
.text:00401026                 mov     byte ptr [edx+6], 1Eh
.text:0040102A                 mov     eax, [ebp+heap_struct]
.text:0040102D                 mov     dword ptr [eax+7], 28h
.text:00401034                 mov     ecx, [ebp+heap_struct]
.text:00401037                 fld     ds:dbl_40B128
.text:0040103D                 fstp    qword ptr [ecx+0Bh]
.text:00401040                 xor     eax, eax
.text:00401042                 mov     esp, ebp
.text:00401044                 pop     ebp
.text:00401045                 retn
.text:00401045 _main           endp
```

El único cambio producido en la estructura es su tamaño que se ha reducido a **19 bytes** y el ajuste del valor de los offset debido al alineamiento de cada campo de la estructura. A pesar de la alineación utilizada al compilar un programa, si nos encontramos con una estructura distribuida en el heap del programa es la forma más rápida para determinar el tamaño y el esquema de una estructura de datos. Sin embargo hay que tener presente que muchas funciones no te ayudarán a poder acceder a los elementos de la estructura para que así puedas entender su esquema. Llegado a este caso, necesitarás seguir la utilización del puntero a la estructura y tomar nota de los offset utilizados siempre que sea referenciado dicho puntero. Así, al final podrás juntar todas tus anotaciones para poder deducir el esquema completo de la estructura.

7.1.2.4.—Estructuras de conjuntos

Podríamos decir que lo bonito de las estructuras de datos compuestas, es que nos permiten construir estructuras complejas las cuales envuelven a otras estructuras más pequeñas dentro de ellas. De entre otras posibilidades, elegimos la capacidad que nos permiten las estructuras de conjuntos de tener estructuras dentro de estructuras y estructuras, las cuales contengan conjuntos como elementos. Lo que ya hemos estudiado

en las secciones anteriores con respecto a conjuntos y estructuras, se aplicará aquí de la misma forma pero tratandolo de forma anidada. Como ejemplo, consideremos un conjunto de estructuras en el siguiente programa en el cual **heap_struct** apunta a un conjunto **tora_struct** de cinco elementos:

```
int main()
       {
           int indice = 1;
           struct tora_struct *heap_struct;
           heap_struct = (struct tora_struct*)malloc(sizeof(struct tora_struct) * 5);
           heap_struct[indice].field1 = 10;
       }
```

Las operaciones necesarias para acceder a **field1** según la línea mostrada, son;

heap_struct (indice) . field1 = 10;

Multiplicar el valor del índice por el tamaño del elemento del conjunto, en este caso el tamaño de la estructura, y añadir el offset del campo deseado. El desensamblado correspondiente sería el siguiente:

```
.text:00401000 ; =============== S U B R O U T I N E =======================================
.text:00401000
.text:00401000 ; Attributes: bp-based frame
.text:00401000
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char *envp)
.text:00401000 _main           proc near               ; CODE XREF: ___tmainCRTStartup+15A↓p
.text:00401000
.text:00401000 indice          = dword ptr -8
.text:00401000 heap_struct     = dword ptr -4
.text:00401000 argc            = dword ptr  8
.text:00401000 argv            = dword ptr  0Ch
.text:00401000 envp            = dword ptr  10h
.text:00401000
.text:00401000                 push    ebp
.text:00401001                 mov     ebp, esp
.text:00401003                 sub     esp, 8
.text:00401006                 mov     [ebp+indice], 1
.text:0040100D                 push    78h             ; Size
.text:0040100F                 call    _malloc
.text:00401014                 add     esp, 4
.text:00401017                 mov     [ebp+heap_struct], eax
.text:0040101A                 mov     eax, [ebp+indice]
.text:0040101D                 imul    eax, 18h
.text:00401020                 mov     ecx, [ebp+heap_struct]
.text:00401023                 mov     dword ptr [ecx+eax], 0Ah
.text:0040102A                 xor     eax, eax
.text:0040102C                 mov     esp, ebp
.text:0040102E                 pop     ebp
.text:0040102F                 retn
.text:0040102F _main           endp
```

El desensamblado nos indica que se pedirán **120 bytes (78h)** para el heap.

```
.text:0040100D                 push    78h             ; Size
```

El índice del conjunto es multiplicado por **24 (18h)**, antes de sumarlo a la dirección de

```
.text:0040101D                 imul    eax, 18h
```

inicio del conjunto, en la línea abajo. Vemos que no se añade ningún offset, con el fin

de generar una dirección final para ser referenciada. De ahí que podemos deducir que el tamaño del elemento del conjunto es de **24**, que el número de elementos en el conjunto es de **120 / 24 = 5**, y que en realidad cada elemento del conjunto es un campo **dword (4-byte)** con inicio de offset **0.**

Performance Bigundill@