LISTAS ENLAZADAS

Voy a copiar la teoría del curso de Cabanes ya que esta mucho mejor explicado de lo que lo haría yo y continuare con el ejemplo en el IDA.

Estructuras dinámicas habituales 1: las listas enlazadas

Ahora vamos a ver un tipo de estructura totalmente dinámica (que puede aumentar o disminuir realmente de tamaño durante la ejecución del programa). Estas son las llamadas listas.

Ahora "el truco" consistirá en que dentro de cada dato almacenaremos todo lo que nos interesa, pero también una referencia que nos dirá dónde tenemos que ir a buscar el siguiente.

Sería algo como:

(Posición: 1023).
Nombre : 'Nacho Cabanes'
Web : 'www.nachocabanes.com'
SiguienteDato : 1430

Este dato está almacenado en la posición de memoria número 1023. En esa posición guardamos el nombre y la dirección (o lo que nos interese) de esta persona, pero también una información extra: la siguiente ficha se encuentra en la posición 1430.

Así, es muy cómodo recorrer la lista de forma secuencial, porque en todo momento sabemos dónde está almacenado el siguiente dato. Cuando lleguemos a uno para el que no esté definido cual es el siguiente dato, quiere decir que se ha acabado la lista.

Por tanto, en cada dato tenemos un enlace con el dato siguiente. Por eso este tipo de estructuras recibe el nombre de "listas simplemente enlazadas" o listas simples. Si tuviéramos enlaces hacia el dato siguiente y el posterior, se trataría de una "lista doblemente enlazada" o lista doble, que pretende hacer más sencillo el recorrido hacia delante o hacia atrás.

Con este tipo de estructuras de información, hemos perdido la ventaja del acceso directo: ya no podemos saltar directamente a la ficha número 500. Pero, por contra, podemos tener tantas fichas como la memoria nos permita, y eliminar una (o varias) de ellas cuando queramos, recuperando inmediatamente el espacio que ocupaba.

Para añadir una ficha, no tendríamos más que reservar la memoria para ella, y el compilador de C nos diría "le he encontrado sitio en la posición 4079". Entonces nosotros iríamos a la última ficha y le diríamos "tu siguiente dato va a estar en la posición 4079".

Esa es la "idea intuitiva". Ahora vamos a concretar cosas en forma de programa en C.

Primero veamos cómo sería ahora cada una de nuestras fichas:

```
struct f { /* Estos son los datos que guardamos: */
  char nombre[30]; /* Nombre, hasta 30 letras */
  char direccion[50]; /* Direccion, hasta 50 */
  int edad; /* Edad, un numero < 255 */
  struct f* siguiente; /* Y dirección de la siguiente */
};
```

La diferencia con un **“struct”** normal está en el campo **“siguiente”** de nuestro registro, que es el que indica donde se encuentra la ficha que va después de la actual, y por tanto será otro puntero a un registro del mismo tipo, un **“struct f *”**.

Un puntero que “no apunta a ningún sitio” tiene el valor NULL (realmente este identificador es una constante de valor 0), que nos servirá después para comprobar si se trata del final de la lista: todas las fichas “apuntarán” a la siguiente, menos la última, que “no tiene siguiente”, y apuntará a NULL.

Entonces la primera ficha definiríamos con

```
struct f *dato1; /* Va a ser un puntero a ficha */
```

y la comenzaríamos a usar con

```
dato1 = (struct f*) malloc (sizeof(struct f)); /* Reservamos memoria */
strcpy(dato1->nombre, "Pepe"); /* Guardamos el nombre, */
strcpy(dato1->direccion, "Su casa"); /* la dirección */
dato1->edad = 40; /* la edad */
dato1->siguiente = NULL; /* y no hay ninguna más */
```

(No debería haber nada nuevo: ya sabemos cómo reservar memoria usando “malloc” y como acceder a los campos de una estructura dinámica usando ->).

Ahora que ya tenemos una ficha, podríamos añadir otra ficha detrás de ella. Primero guardamos espacio para la nueva ficha, como antes:

```
struct f *dato2;

dato2 = (struct f*) malloc (sizeof(struct f)); /* Reservamos memoria */
strcpy(dato2->nombre, "Juan"); /* Guardamos el nombre, */
strcpy(dato2->direccion, "No lo sé"); /* la dirección */
dato2->edad = 35; /* la edad */
dato2->siguiente = NULL; /* y no hay ninguna más */
```

y ahora enlazamos la anterior con ella:

```
dato1->siguiente = dato2;
```

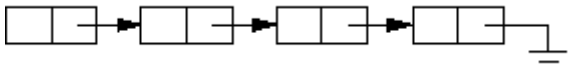
Si quisiéramos introducir los datos ordenados alfabéticamente, basta con ir comparando cada nuevo dato con los de la lista, e insertarlo donde corresponda. Por ejemplo, para insertar un nuevo dato entre los dos anteriores, haríamos:

```
struct f *dato3;

dato3 = (struct f*) malloc (sizeof(struct f)); /* La tercera */
strcpy(dato3->nombre, "Carlos");
strcpy(dato3->direccion, "Por ahí");
dato3->edad = 14;
dato3->siguiente = dato2; /* enlazamos con la siguiente */
dato1->siguiente = dato3; /* y la anterior con ella */
printf("La lista inicialmente es:\n");
```

La estructura que hemos obtenido es la siguiente

Dato1 - Dato3 - Dato2 - NULL

Gráficamente:



Es decir: cada ficha está enlazada con la siguiente, salvo la última, que no está enlazada con ninguna (apunta a NULL).

Si ahora quisiéramos borrar Dato3, tendríamos que seguir dos pasos:

1.- Enlazar Dato1 con Dato2, para no perder información.
2.- Liberar la memoria ocupada por Dato3.

Esto, escrito en "C" sería:

dato1->siguiente = dato2; /* Borrar dato3: Enlaza Dato1 y Dato2 */
free(dato3); /* Libera lo que ocupó Dato3 */

Hemos empleado tres variables para guardar tres datos. Si tenemos 20 datos, ¿necesitaremos 20 variables? ¿Y 3000 variables para 3000 datos?

Sería tremendamente ineficiente, y no tendría mucho sentido. Es de suponer que no sea así. En la práctica, basta con dos variables, que nos indicarán el principio de la lista y la posición actual, o incluso sólo una para el principio de la lista.

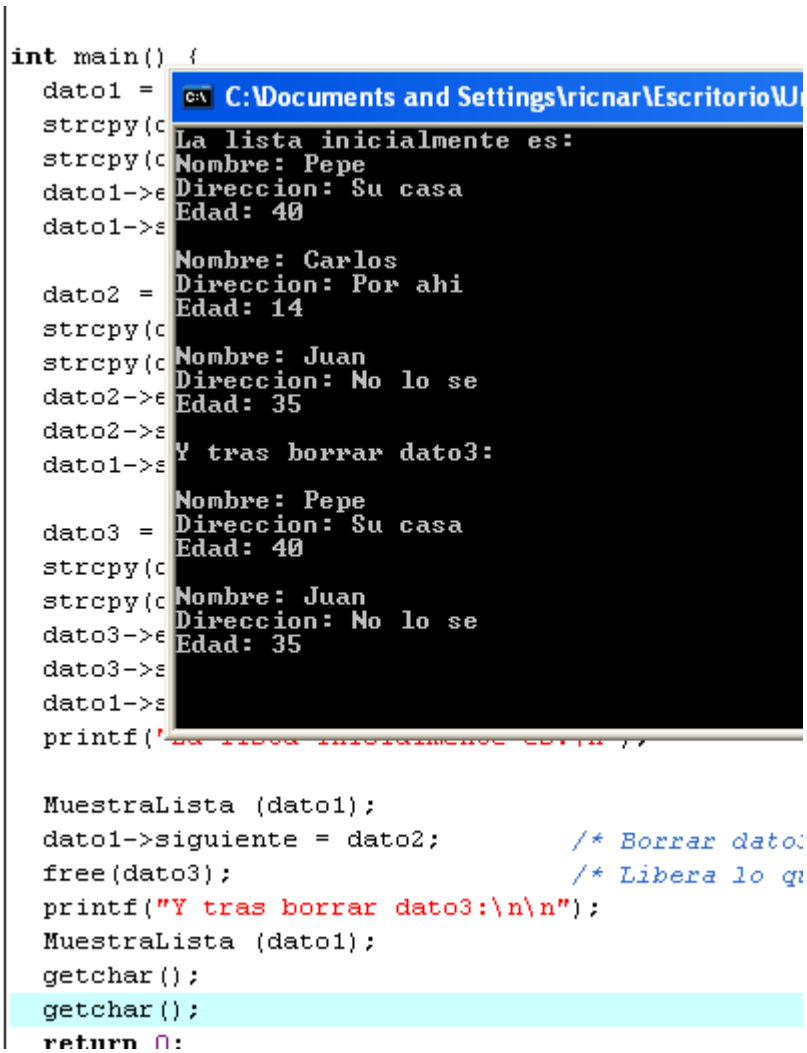
Por ejemplo, una rutina que muestre en pantalla toda la lista se podría hacer de forma recursiva así:

```
void MuestraLista ( struct f *inicial ) {
  if (inicial!=NULL) { /* Si realmente hay lista */
    printf("Nombre: %s\n", inicial->nombre);
    printf("Dirección: %s\n", inicial->direccion);
    printf("Edad: %d\n\n", inicial->edad);
    MuestraLista ( inicial->siguiente ); /* Y mira el siguiente */
  }
}
```

Lo llamaríamos con "**MuestraLista(dato1)**", y a partir de ahí el propio procedimiento se encarga de ir mirando y mostrando los siguientes elementos hasta llegar a NULL, que indica el final.

Antes de seguir, vamos a juntar todo esto en un programa, para comprobar que realmente funciona: añadimos los 3 datos y decimos que los muestre desde el primero; luego borramos el del medio y los volvemos a mostrar:

Hasta aquí la explicación teórica que creo se entiende perfectamente, así que vayamos al código fuente a explicarlo y luego a verlo en IDA.



Veamoslo en IDA ahora veremos que no es tan complicado como parece.

```
_main proc near

var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h

push    ebp
mov     ebp, esp
sub     esp, 18h
and     esp, 0FFFFFFF0h
mov     eax, 0
add     eax, 0Fh
add     eax, 0Fh
shr     eax, 4
shl     eax, 4
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
call    ___chkstk
call    _main
mov     dword ptr [esp], 58h ; size_t
call    malloc
mov     ds:dword_404080, eax
```

Vemos que en el main no tenemos variables locales, ya que las que estan allí son las que siempre crea y usa el procesador y no son nuestras, si recordamos el código fuente veamos que no hay variables locales definidas dentro de main, las que usa son todas globales.

```
struct f {            /* Estos son los datos que guardamos: */
  char nombre[30];          /*   Nombre, hasta 30 letras */
  char direccion[50];       /*   Direccion, hasta 50 */
  int edad;                 /*   Edad, un numero < 255 */
  struct f* siguiente;      /*   Y dirección de la siguiente */
};

struct f *dato1;            /* Va a ser un puntero a ficha */
struct f *dato2;            /* Otro puntero a ficha */
struct f *dato3;            /* Y otro más */

void MuestraLista ( struct f *inicial ) {
  if (inicial!=NULL) {                      /* Si realmente hay lista */
    printf("Nombre: %s\n", inicial->nombre);
    printf("Direccion: %s\n", inicial->direccion);
    printf("Edad: %d\n\n", inicial->edad);
    MuestraLista ( inicial->siguiente );    /* Y mira el siguiente */
  }
}

int main() {
  dato1 = (struct f*) malloc (sizeof(struct f)); /* Reservamos memoria */
  strcpy(dato1->nombre, "Pepe");            /* Guardamos el nombre, */
```

Aquí comienza lo nuestro, reserva 58 hexa de tamaño que es 88 decimal para la primera ficha o ficha1, recordamos que era 30 para el primer campo, 50 para el segundo, 4 para la edad y 4 para el puntero al siguiente lo que da 88 decimal.

```
call    ___main
mov     dword ptr [esp], 58h ; size_t
call    malloc
mov     ds:dword_404080, eax
```

El puntero que en nuestro código llamábamos **dato1** lo guarda en la variable global **404080**, así que renombramos allí a **dato1**.

```
mov     ds:dato1, eax
mov     dword ptr [esp+4], offset aPepe ; "Pepe"
mov     eax, ds:dato1
mov     [esp], eax      ; char *
call    strcpy
```

Luego llama a **strcpy** pasandole el puntero a la string **“Pepe”** como fuente y como destination el puntero **dato1**, para que inicialice copiando allí el primer campo de la ficha.

Luego corre el puntero desde el inicio **1Eh** o sea 30 decimal mas adelante para escribir la **dirección** que era el segundo campo.

```
mov     dword ptr [esp+4], offset aSuCasa ; "Su casa"
mov     eax, ds:dato1
add     eax, 1Eh
mov     [esp], eax      ; char *
call    strcpy
```

Podemos ponerlo en decimal usando el menú del botón derecho allí.

```
call    strcpy
mov     dword ptr [esp+4], offset aSuCasa ; "Su casa"
mov     eax, ds:dato1
add     eax, 30
mov     [esp], eax      ; char *
```

Luego guarda la **edad** y el cero en el campo **siguiente** ya que es la ultima ficha creada en esta lista, sumándole 50 al puntero **dato1** y escribiendo en el contenido que sera el campo **edad** y luego sumándole 54 al puntero **dato1** y escribiendo en el contenido que sera el campo **siguiente**.

```
mov     eax, ds:dato1
mov     dword ptr [eax+50h], 28h
mov     eax, ds:dato1
mov     dword ptr [eax+54h], 0
```

Podemos pasarlo a decimal así se ve correctamente la edad y apretando punto y coma puedo agregar algún comentario adicional como que el 40 es la edad y que el cero es el que marca la condición de ultimo.

```
mov     eax, ds:dato1
mov     dword ptr [eax+50h], 40 ; edad
mov     eax, ds:dato1
mov     dword ptr [eax+54h], 0 ; cero de ultima en la lista
mov     dword ptr [esp], 58h ; size_t
```

```
mov     dword ptr [esp], 58h ; size_t
call    malloc
mov     ds:dato2, eax
mov     dword ptr [esp+4], offset aJuan ; "Juan"
mov     eax, ds:dato2
mov     [esp], eax      ; char *
call    strcpy
mov     dword ptr [esp+4], offset aNoLoSe ; "No lo se'
mov     eax, ds:dato2
add     eax, 30
mov     [esp], eax      ; char *
call    strcpy
mov     eax, ds:dato2
mov     dword ptr [eax+50h], 35 ; edad
mov     eax, ds:dato2
mov     dword ptr [eax+54h], 0 ; nuevo cero final
```

Vemos que hace exactamente lo mismo con la segunda ficha o ficha2 en la lista, reserva la memoria le copia los datos y pone el cero en el campo siguiente, lo que queda ahora es ver como arregla el campo siguiente de la ficha1 para quitar el cero y poner el puntero a la ficha2.

```
mov     edx, ds:dato1
mov     eax, ds:dato2
mov     [edx+54h], eax
```

Ahí esta mueve a EDX **dato1** el puntero a la ficha1 y a EAX **dato2** el puntero a ficha2 y luego escribe en el contenido de **dato1** mas 54h o sea en el campo **siguiente** de la ficha1, el puntero **dato2** para que apunte a la ficha2 y se mantenga la lista enlazada.

Luego realiza el mismo trabajo con la ficha3.

```
mov     dword ptr [esp], 58h ; size_t
call    malloc
mov     ds:dato3, eax
mov     dword ptr [esp+4], offset aCarlos ; "Carlos"
mov     eax, ds:dato3
mov     [esp], eax      ; char *
call    strcpy
mov     dword ptr [esp+4], offset aPorAhi ; "Por ahi"
mov     eax, ds:dato3
add     eax, 30
mov     [esp], eax      ; char *
call    strcpy
mov     eax, ds:dato3
mov     dword ptr [eax+50h], 14 ; edad
```

Y luego arregla los punteros haciendo que esta ficha3 se incluya en medio de las dos existentes, quede como la segunda en la lista enlazada, vemos primero que al siguiente de la ficha3 le mueve el puntero a la ficha2, y luego al siguiente de la ficha1 hace que apunte a ficha3 para que queden en la lista en el orden

ficha1--> ficha3 → ficha2

```
mov     edx, ds:dato3
mov     eax, ds:dato2
mov     [edx+54h], eax
mov     edx, ds:dato1
mov     eax, ds:dato3
mov     [edx+54h], eax
```

mov dword ptr [esp], offset aLaListaInicial ; "La lista inicialmente es:\n"
call printf

Luego de imprimir el mensaje anterior llama a la funcion que en nuestro código se llamaba **MuestraLista**, así que la renombramos.

```
call    printf
mov     eax, ds:dato1
mov     [esp], eax
call    sub_401290
```

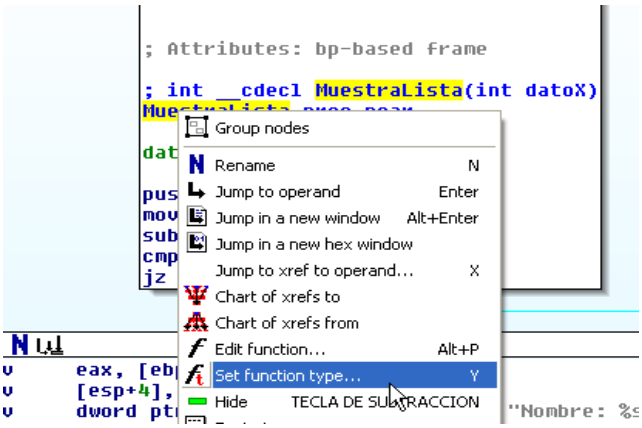
Entrando en ella vemos que tiene un argumento ya que se le pasaba el puntero a la ficha1 llamado **dato1.**

```
mov     eax, ds:dato1
mov     [esp], eax
call    MuestraLista
```

```
; Attributes: bp-based frame

MuestraLista proc near

arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 8
cmp     [ebp+arg_0], 0
jz      short locret_4012E9
```

Sabemos que dentro de la funcion tomara los valores de **dato1, dato2 o dato3** lo llamaremos **datoX** para hacerlo bien genérico.

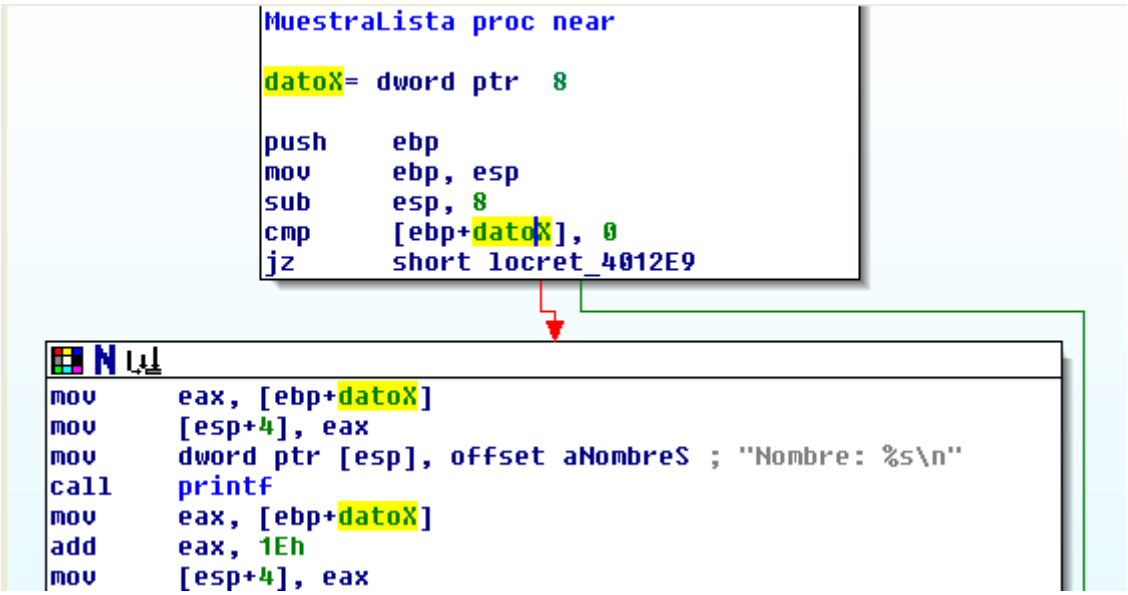
Luego propagamos la definición de la funcion con el argumento **datoX**, yendo a Set Function Type.



Así propago el nombre del argumento hacia main, veo que en el main aparece, como que le paso **dato1** como argumento y que la funcion lo recibe como **datoX** ya que es genérico para los diferentes llamados que hay a la misma.

```
call    printf
mov     eax, ds:dato1
mov     [esp], eax      ; datoX
call    MuestraLista
```

Aquí se fija si **datoX** es cero eso solo puede ocurrir si no hay lista o si la ficha es la ultima, en este caso **datoX** vale **dato1** y es distinto de cero.



Así que usando ese puntero a dato1 llama a **printf** haciendo format string **%s** imprimiendo el campo **nombre**, luego le suma 30 para hallar el puntero al campo **dirección** y mediante format string imprimir la misma.

```
mov     eax, [ebp+datoX]
add     eax, 30
mov     [esp+4], eax
mov     dword ptr [esp], offset aDireccionS ; "Direccion: %s\n"
call    printf
```

Luego le suma 50h al puntero **dato1** y lee la edad y hace format string usando **%d** para imprimir la misma.

```
mov     eax, [ebp+datoX]
mov     eax, [eax+50h]
mov     [esp+4], eax
mov     dword ptr [esp], offset aEdadD ; "Edad: %d\n\n"
call    printf
```

Luego llama a la misma funcion recursivamente, pasandole como argumento el campo **siguiente** que esta **54h** a partir de **dato1**, y vuelve a repetir el mismo proceso para imprimir los datos de la siguiente ficha de la lista y así sucesivamente hasta que llegue a la ultima que tiene valor cero en el campo **siguiente** y por eso sale y vuelve al main.

```
mov     eax, [ebp+datoX]
mov     eax, [eax+54h]
mov     [esp], eax      ; datoX
call    MuestraLista
```

```
datoX= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 8
cmp     [ebp+datoX], 0
jz      short locret_4012E9
```

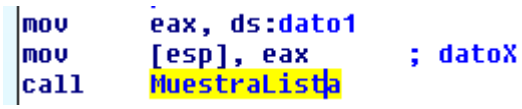
SALIDA

```
mov     eax, [ebp+datoX]
mov     [esp+4], eax
mov     dword ptr [esp], offset aNombreS ; "Nombre: %s\n"
```

Luego elimina la ficha3, para ello arregla los punteros y al **siguiente** de la ficha1, le guarda el puntero **dato2**, y hace **free()** de **dato3** con lo cual la elimina.

```
mov     edx, ds:dato1
mov     eax, ds:dato2
mov     [edx+54h], eax
mov     eax, ds:dato3
mov     [esp], eax      ; void *
call    free
```

Luego imprime nuevamente toda la lista pasandole el puntero a ficha1, ahora solo imprimirá la **ficha1**, y la **ficha2**, ya que la **ficha3** desapareció por el **free()** y el arreglo de los punteros **siguiente**.



Eso es todo sobre el ejemplo de lista simplemente enlazada veremos si alguno es guapo y reversea el ejercicio que es una lista simplemente enlazada mas compleja que este jeje.

Hasta la parte siguiente:

ricnar