APÈNDIX: PROGRAMACIÓ EN C (CONVENCIONS I PUNTERS)

1. Convencions en C

Al llenguatge C existeixen una sèrie de convencions que cal saber per poder programar correctament les subrutines i especialment les que contenen codi en assemblador. Aquí les teniu, es important que les tingueu sempre a ma.

- Els paràmetres es pasan de dreta a esquerra
- Els vectors i matrius es pasan per referència
- Els structs es pasen per valor
- Els caràcters (1 byte) ocupen 4 bytes
- Els short (2 bytes) ocupen 4 bytes
- Els registres % ebx, % esi, % edi se han de salvar si son modificats
- Els registres % eax, % ecx, %edx es poden modificar a l'interior d'una subrutina. Se ès necessari, el codi que crida la rutina els ha de salvar
- Els resultats sempre es retornen en %eax

2. Tutorial de punters en C

Aqui teniu una adreça web on podeu consultar un tutorial per utilitzar els punters en C (http://www.publispain.com/supertutoriales/programacion/c_y_cplus/cursos/1/cursoc9.html). El document que teniu a continuacio és una copia del contingut d'aquesta adreça. L'autor és Gorka Urrutia (link).

2.1. Introducción

Punteros!!! uff. Si hasta ahora te había parecido complicado prepárate. Este es uno de los temas que más suele costar a la gente al aprender C. Los punteros son una de las más potentes características de C, pero a la vez uno de sus mayores peligros. Los punteros nos permites acceder directamente a cualquier parte de la memoria. Esto da a los programas C una gran potencia. Sin embargo son una fuente ilimitada de errores. Un error usando un puntero puede bloquear el sistema (si usamos ms-dos o win95, no en Linux) y a veces puede ser difícil detectarlo.
Otros lenguajes no nos dejan usar punteros para evitar estos problemas, pero a la vez nos quitan parte del control que tenemos en C. No voy a entrar a discutir si es mejor o no poder usar punteros, aunque pienso que es mejor. Yo me voy a limitar a explicar cómo funcionan.
A pesar de todo esto no hay que tenerles miedo. Casi todos los programas C usan punteros. Si aprendemos a usarlos bien no tendremos mas que algún problema esporádico. Así que atención, valor y al toro.

2.2. La memoria del ordenador

Si tienes bien claro lo que es la memoria del ordenador puedes saltarte esta sección. Pero si confundes la memoria con el disco duro o no tienes claro lo que es no te la pierdas. A lo largo de mi experiencia con ordenadores me he encontrado con mucha gente que no tiene claro cómo funciona un ordenador. Cuando hablamos de memoria nos estamos refiriendo a la memoria RAM del ordenador. Son unas *pastillas* que se conectan a la placa base y nada tienen que ver con el disco duro. El disco duro guarda los datos permanentemente (hasta que se rompe) y la información se almacena como ficheros. Nosotros podemos decirle al ordenador cuándo grabar, borrar, abrir un

documento, etc. La memoria Ram en cambio, se borra al apagar el ordenador. La memoria Ram la usan los programas sin que el usuario de éstos se de cuenta.

Para hacernos una idea, hoy en dia la memoria se mide en MegaBytes (suelen ser 16, 32, 64, 128Mb) y los discos duros en GigaBytes (entre 3,4 y 70Gb, o mucho más). Hay otras memorias en el ordenador aparte de la mencionada. La memoria de video (que está en la tarjeta gráfica), las memorias caché (del procesador, de la placa...) y quizás alguna más que ahora se me olvida.

2.3. Direcciones de variables

Vamos a ir como siempre por partes. Primero vamos a ver qué pasa cuando declaramos una variable.

Al declarar una variable estamos diciendo al ordenador que nos reserve una parte de la memoria para almacenarla. Cada vez que ejecutemos el programa la variable se almacenará en un sitio diferente, eso no lo podemos controlar, depende de la memoria disponible y otros factores misteriosos. Puede que se almacene en el mismo sitio, pero es mejor no fiarse. Dependiendo del tipo de variable que declaremos el ordenador nos reservará más o menos memoria. Como vimos en el capítulo de tipos de datos cada tipo de variable ocupa más o menos bytes. Por ejemplo si declaramos un char, el ordenador nos reserva 1 byte (8 bits). Cuando finaliza el programa todo el espacio reservado queda libre.

Existe una forma de saber qué direcciones nos ha reservado el ordenador. Se trata de usar el operador **&** (operador de dirección). Vamos a ver un ejemplo: Declaramos la variable 'a' y obtenemos su valor y dirección.

```
#include <stdio.h>

void main()
        {
        int a;

        a = 10;
        printf( "Dirección de a = %p, valor de a = %i\n", &a, a
);
        }
```

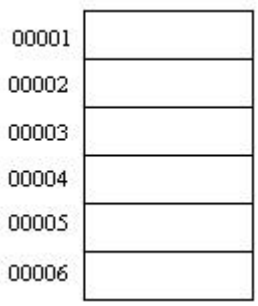
Para mostrar la dirección de la variable usamos %p en lugar de %i, sirve para escribir direcciones de punteros y variables. El valor se muestra en hexadecimal.

No hay que confundir el valor de la variable con la dirección donde está almacenada la variable. La variable 'a' está almacenada en un lugar determinado de la memoria, ese lugar no cambia mientras se ejecuta el programa. El valor de la variable puede cambiar a lo largo del programa, lo cambiamos nosotros. Ese valor está almacenado en la dirección de la variable. El nombre de la variable es equivalente a poner un nombre a una zona de la memoria. Cuando en el programa escribimos 'a', en realidad estamos diciendo, "el valor que está almacenado en la dirección de memoria a la que llamamos 'a'".

2.4. Que son los punteros

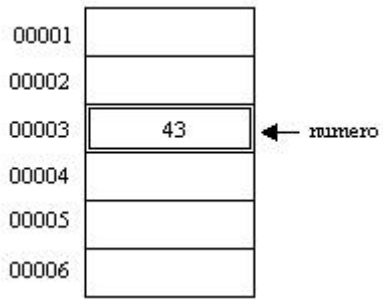
Ahora ya estamos en condiciones de ver lo que es un puntero. Un puntero es una variable un tanto especial. Con un puntero podemos almacenar direcciones de memoria. En un puntero podemos tener guardada la dirección de una variable.

Vamos a ver si cogemos bien el concepto de puntero y la diferencia entre éstos y las variables *normales*.



En el dibujo anterior tenemos una representación de lo que sería la memoria del ordenador. Cada casilla representa un byte de la memoria. Y cada número es su dirección de memoria. La primera casilla es la posición 00001 de la memoria. La segunda casilla la posición 00002 y así sucesivamente.

Supongamos que ahora declaramos una variable char: ***char numero = 43***. El ordenador nos guardaría por ejemplo la posición 00003 para esta variable. Esta posición de la memoria queda reservada y ya no la puede usar nadie más. Además esta posición a partir de ahora se le llama *numero*. Como le hemos dado el valor 43 a numero, el valor 43 se almacena en numero, es decir, en la posición 00003.



Si ahora usáramos el programa anterior:

```
#include <stdio.h>

void main()
        {
        int numero;

        numero = 43;
        printf( "Dirección de numero = %p, valor de numero =
%i\n", &numero, numero );
        }
```

El resultado sería:

```
Dirección de numero = 00003, valor de numero = 43
```

Creo que así ya está clara la diferencia entre el valor de una variable (43) y su dirección (00003). Ahora vamos un poco más allá, vamos a declarar un puntero.

Hemos dicho que un puntero sirve para almacenar la direcciones de memoria. Muchas veces los punteros se usan para guardar las direcciones de variables. Vimos en el capítulo Tipos de Datos que cada tipo de variable ocupaba un espacio distinto. Por eso cuando declaramos un puntero debemos especificar el tipo de datos cuya dirección almacenará. En nuestro ejemplo queremos que almacene la dirección de una variable char. Así que para declarar el puntero **punt** debemos hacer:

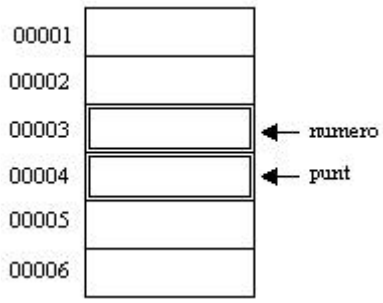
```
char *punt;
```

El * (asterisco) sirve para indicar que se trata de un puntero, debe ir justo antes del nombre de la variable, sin espacios. En la variable punt sólo se pueden guardar direcciones de memoria, no se pueden guardar datos. Vamos a volver sobre el ejemplo anterior un poco ampliado para ver cómo funciona un puntero:

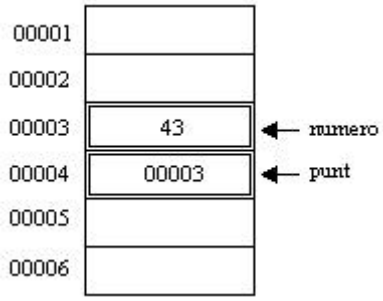
```
#include <stdio.h>

void main()
        {
        int numero;
        int *punt;

        numero = 43;
        punt = &numero;
        printf( "Dirección de numero = %p, valor de numero =
%i\n", &numero, numero );
        }
```

Vamos a ir línea a línea.

- En la primera *int numero* reservamos memoria para numero (supongamos que queda como antes, posición 00003). Por ahora numero no tiene ningún valor.
- Siguiente línea: *int *punt;*. Reservamos una posición de memoria para almacenar el puntero. Lo normal es que según se declaran variables se guarden en posiciones contiguas. De modo que quedaría en la posición 00004. Por ahora punt no tiene ningún valor, es decir, no apunta a ninguna variable. Esto es lo que tenemos por ahora:



- Tercera línea: *numero = 43;*. Aquí ya estamos dando el valor 43 a numero. Se almacena 43 en la dirección 00003, que es la de numero.
- Cuarta línea: *punt = №*. Por fin damos un valor a punt. El valor que le damos es la dirección de numero (ya hemos visto que & devuelve la dirección de una variable). Así que punt tendrá como valor la dirección de numero, 00003. Por lo tanto ya tenemos:



Cuando un puntero tiene la dirección de una variable se dice que ese puntero **apunta** a esa variable.

NOTA: La declaración de un puntero depende del tipo de dato al que queramos apuntar. En general la declaración es:

```
tipo_de_dato *nombre_del_puntero;
```

Si en vez de querer apuntar a una variable tipo char como en el ejemplo hubiese sido de tipo int:

```
int *punt;
```

2.5. Para que sirve un puntero y como se usa

Los punteros tienen muchas utilidades, por ejemplo nos permiten pasar argumentos (o parámetros) a una función y modificarlos. También permiten el manejo de cadenas y de arrays. Otro uso importante es que nos permiten acceder directamente a la pantalla, al teclado y a todos los componentes del ordenador. Pero esto ya lo veremos más adelante.

Pero si sólo sirvieran para almacenar direcciones de memoria no servirían para mucho. Nos deben dejar también la posibilidad de acceder a esas posiciones de memoria. Para acceder a ellas se usa el operador *, que no hay que confundir con el de la multiplicación.

```
#include <stdio.h>

void main()
        {
        int numero;
        int *punt;

        numero = 43;
        punt = &numero;
        printf( "Dirección de numero = %p, valor de numero = %i\n", &numero, *punt );
        }
```

Si nos fijamos en lo que ha cambiado con respecto al ejemplo anterior, vemos que para acceder al valor de número usamos *punt en vez de numero. Esto es así porque punt apunta a numero y *punt nos permite acceder al valor al que apunta punt.

```
#include <stdio.h>

void main()
        {
        int numero;
        int *punt;
```

```
                numero = 43;
                punt = &numero;
                *punt = 30;
                printf( "Dirección de numero = %p, valor de numero =
%i\n", &numero, numero );
                }
```

Ahora hemos cambiado el valor de numero a través de *punt.

En resumen, usando punt podemos apuntar a una variable y con *punt vemos o cambiamos el contenido de esa variable.

Un puntero no sólo sirve para apunta a una variable, también sirve para apuntar una dirección de memoria determinada. Esto tiene muchas aplicaciones, por ejemplo nos permite controlar el hardware directamente (en MS-Dos y Windows, no en Linux). Podemos escribir directamente sobre la memoria de video y así escribir directamente en la pantalla sin usar printf.

2.6. Usando punteros en una comparación

Veamos el siguiente ejemplo. Queremos comprobar si dos variables son iguales usando punteros:

```
        #include <stdio.h>

        void main()
                {
                int a, b;
                int *punt1, *punt2;

                a = 5; b = 5;
                punt1 = &a; punt2 = &b;

                if ( punt1 == punt2 )
                        printf( "Son iguales\n" );
                }
```

Alguien podría pensar que el *if* se cumple y se mostraría el mensaje *Son iguales* en pantalla. Pues no es así, el programa es erróneo. Es cierto que a y b son iguales. También es cierto que punt1 apunta a 'a' y punt2 a 'b'. Lo que queríamos comprobar era si a y b son iguales. Sin embargo con la condición estamos comprobando si punt1 apunta al mismo sitio que punt2, estamos comparando las direcciones donde apuntan. Por supuesto a y b están en distinto sitio en la memoria así que la condición es falsa. Para que el programa funcionara deberíamos usar los asteriscos:

```
        #include <stdio.h>

        void main()
                {
                int a, b;
                int *punt1, *punt2;

                a = 5; b = 5;
                punt1 = &a; punt2 = &b;

                if ( *punt1 == *punt2 )
                        printf( "Son iguales\n" );
                }
```

Ahora sí. Estamos comparando el contenido de las variables a las que apuntan punt1 y punt2. Debemos tener mucho cuidado con esto porque es un error que se cuela con mucha facilidad.

Vamos a cambiar un poco el ejemplo. Ahora 'b' no existe y punt1 y punt2 apuntan a 'a'. La condición se cumplirá porque apuntan al mismo sitio.

```
#include <stdio.h>

void main()
        {
        int a;
        int *punt1, *punt2;

        a = 5;
        punt1 = &a; punt2 = &a;

        if ( punt1 == punt2 )
                printf( "punt1 y punt2 apuntan al mismo
sitio\n" );
        }
```

2.7. Punteros como argumentos de funciones

Hemos visto en el capítulo de funciones cómo pasar parámetros y cómo obtener resultados de las funciones (con los valores devueltos con return). Pero tiene un inconveniente, sólo podemos tener un valor devuelto. Ahora vamos a ver cómo los punteros nos permites modificar varias variables en una función.

Hasta ahora para pasar una variable a una función hacíamos lo siguiente:

```
#include <stdio.h>

int suma( int a, int b )
        {
        return a+b:
        }

void main()
        {
        int var1, var2;

        var1 = 5; var2 = 8;
        printf( "La suma es : %i\n", suma(var1, var2) );
        }
```

Bien aquí hemos pasado a la función los parámetros 'a' y 'b' (que no podemos modificar) y nos devuelve la suma de ambos. Supongamos ahora que queremos tener la suma pero además queremos que var1 se haga cero dentro de la función. Para eso haríamos lo siguiente:

```
#include <stdio.h>

int suma( int *a, int b )
        {
        *a = 0;
        return a+b:
        }
```

```
        void main()
                {
                int var1, var2;

                var1 = 5; var2 = 8;
                printf( "La suma es: %i y a vale: %i\n", suma(&var1,
var2), var1 );
                }
```

Fijémonos en lo que ha cambiado (con letra en negrita): En la función suma hemos declarado 'a' como puntero. En la llamada a la función (dentro de main) hemos puesto & para pasar la dirección de la variable var1. Ya sólo queda hacer cero a var1 a través de *a=0.

Es importante no olvidar el operador & en la llamada a la función ya que sin el no estaríamos pasando la dirección de la variable sino cualquier otra cosa.

Podemos usar tantos punteros como queramos en la definición de la función.