

Estudio de un Sistema Operativo

Diseño de Sistemas Operativos

Problemas de aula 2. Punteros en C

Juan Carlos Pérez & Sergio Sáez

Sergio Sáez ssaez@disca.upv.es

Juan Carlos Pérez jcperez@disca.upv.es

2.1 Introducción

- En esta sesión practicaremos el uso de punteros y estructuras de datos que los usan.
- El uso eficiente y seguro de los punteros es fundamental e inevitable en C.
Plantearemos unos cuantos ejercicios comenzando desde los usos más simples de los punteros, con forma de vectores, hasta el uso de punteros a funciones y estructuras de datos complejas.
- Empezaremos por la definición:

Un puntero es **una variable que contiene** (apunta a) **una dirección de memoria**.

Lleva asociado un **tipo** para que el compilador sepa **cómo debe interpretar** la información contenida en la posición de memoria a la que el puntero apunta. Si el tipo es **void**, hablaremos de un "puntero genérico", es decir que representa simplemente una dirección de memoria y puede asignársele un puntero de cualquier tipo

2.2 Un ejemplo

Copiad este programa simple en un editor (por ejemplo **emacs**). Trabajad en un subdirectorio de vuestro **home** que puede llamarse **punteros**.

```
/* punt1.c */
#include <stdio.h>

int main(int argc, char *argv[]) {
    int v[3]={0,0,0}, *p=0;

    printf("v= %p ; Valores v: %d %d %d\n",v, v[0], v[1], v[2]);
    printf("p= %p ; Valores p: %d %d %d\n",p, p[0], p[1], p[2]);

    v[0]=9; v[1]=8; v[2]=7; v[3]=6;
    printf("v= %p ; Valores v: %d %d %d\n",v, v[0], v[1], v[2]);

    p=v;
    printf("p= %p ; Valores p: %d %d %d\n",p, p[0], p[1], p[2]);
} /* end main */
```

- Compila y ejecuta el programa. ¿Qué ocurre?






- Analiza cuál es el problema, entiende bien lo que ha ocurrido y elimina la línea que lo provoca.







- Ahora el programa funciona, ilustrando el hecho de que un puntero en C se puede manejar automáticamente como un vector. De hecho, un vector también se puede usar como un puntero, es decir un puntero y un vector en C se manejan igual.
- Cambia los dos últimos **printf** del programa para que impriman los mismos valores usando sintaxis de punteros. Recuerda: El contenido de un puntero **a** se expresa como ***a** y el de la posición siguiente como ***(a+1)**, teniendo en cuenta que la dirección dentro del paréntesis se incrementa en tantos bytes como ocupe el tipo al que apunta el puntero.





2.3 El operador &

- El operador **&** devuelve un puntero al objeto al que se le aplica (la dirección de memoria donde se halla ese objeto).
- Para comprobar cómo se usa, define tres variables enteras **v1**, **v2** y **v3** dentro de **main()**, inicializándolas con tres valores enteros. Por ejemplo con 10, 15 y 20 respectivamente.
- Ahora asigna a **p** la **dirección** de **v1** e imprime los valores de **p** y ***p** (dirección y contenido). Haz lo mismo para **v2** y **v3**.
- Compila y ejecuta el nuevo programa. ¿Qué resultado obtienes? Deberás ver las direcciones y los valores de las tres variables. Si te fijas, verás que el compilador las ha alojado en memoria consecutivamente (en la pila). 
- Ahora asigna a **p** la dirección al elemento 2 del vector **v**, utilizando la notación de vector y el operador **&**. Imprime **p** y ***p**. 
- Haz lo mismo utilizando la aritmética de punteros y el vector **v** como un puntero al primer elemento. 

- Asigna ahora a **p** la dirección de **v1** y recupera la línea en la que imprimías los tres valores del vector **v** usando el puntero **p**. ¿Imprime los valores de las tres variables? ¿Por qué? ¿Están seguidas en memoria? ¿Cuál de ellas tiene la dirección de memoria más baja? 
- Imprime los valores de **v1**, **v2** y **v3** usando el puntero **p** con sintaxis de puntero y de vector. 

- Haz lo mismo usando la expresión ***(p++)** para ir incrementando el puntero. Prueba a cambiar ***(p++)** por **(*p)++**. ¿Qué ocurre? 
- Vuelve a asignar a **p** la dirección de **v3** y a continuación cambia el contenido de la posición de memoria apuntada por **p**, **(p + 1)** y **(p + 2)**. Para ello puedes utilizar expresiones del tipo ***p=...;**.
- Imprime los valores de **v1**, **v2** y **v3** usando las propias variables. 

2.4 Punteros a punteros

- Los punteros son variables que ocupan una posición de memoria para almacenar una dirección. Así pues, puede declararse un puntero que apunte a dicha posición de memoria, es decir, un puntero a puntero.
- La sintaxis de la declaración es idéntica: **tipo * puntero;** solo que ahora el tipo es otro un puntero, quedando algo similar a: **tipo **p_puntero;**
- Los punteros a punteros, o dobles punteros, se utilizan normalmente para modificar el contenido de un puntero sin saber el nombre de la variable puntero modificada. Muy útil a la hora de crear listas enlazadas.
- Declara un puntero a puntero a un entero **ppi** y asígnale la dirección de **p** (**¡Ojo!** La dirección de **p**, no su contenido). Después asígnale a **p** la dirección de **v** utilizando el puntero **ppi**. Imprime **p** y ***p**. 
- ¿Podrías asignarle a **ppi** la dirección del puntero **v**? ¿Por qué? 

2.5 Punteros a funciones

Una capacidad muy útil que no podía faltar en C es la de manejar punteros a funciones. Veamos primero la sintaxis asociada a ese manejo:

- Declarar un puntero a función es como declarar la función pero poniendo el nombre precedido de un asterisco y entre paréntesis.

```
/* Declaración de una función */
int func(int a, float b);
```

```
/* Declaración de un puntero a función */
int (*p_func)(int a, float b);
```

- Los paréntesis distinguen la declaración de un puntero a función de la de una función que devuelve un puntero:

```
/* Declaración de una función que devuelve un puntero a int*/
int *func(int a, float b);
```

Copia el siguiente programa en tu editor y pruébalo.

```
/* puntfunc.c */
#include <stdio.h>

int suma1(int x1, int x2);
void suma2(int x1, int x2, int *x3);

int main(int argc, char *argv[]) {
    int a=6, b=2, c, d;


    c= suma1(a, b); printf("a + b = %d\n",c);
    suma2(a, b, &d); printf("a + b = %d\n",d);
} /* end main */


int suma1(int x1, int x2){
    return (x1+x2);
} /* end suma1 */

void suma2(int x1, int x2, int *x3){
    *x3= x1 + x2;
} /* end suma2 */
```

- Declara dentro de **main** dos punteros **p_oper1** y **p_oper2** a las funciones **suma1** y **suma2** respectivamente.
- No basta con definir los punteros. Hay que darles un valor. Para asignar un valor a un puntero a función basta con usar el nombre de la función, sin más. En una expresión, el nombre de una función representa su dirección (un puntero), del mismo modo que el nombre de un vector representa también un puntero en una expresión en C.
- Para utilizar un puntero a función, es decir, para invocar a la función apuntada se pueden usar dos sintaxis alternativas:

(*p_func)(a,b) ó **p_func(a,b)**

- Asigna las dos funciones a los punteros que has definido y usa los punteros en lugar de las funciones. Comprueba que el resultado es el mismo. 
- Imprime los punteros a las funciones y los punteros a las variables locales y comprueba que se encuentran en zonas diferentes de memoria. Prueba a imprimir también la **dirección de los punteros** (usando el operador &) y comprobarás que, lógicamente, están en la pila junto al resto de las variables locales.


- Define ahora una función **resta1()** igual a **suma1()** pero que realice la resta en lugar de la suma.
- Asigna al puntero **p_oper1** la dirección de la función **resta1()**, invoca a la función usando el puntero **p_oper1** y comprueba que cambia el resultado que obtienes. 
- Por supuesto, no puedes hacer lo mismo con el puntero **p_oper2()** ya que son tipos diferentes.
- Un puntero a función sólo puede apuntar a funciones con parámetros y valor de retorno de los mismos tipos. No se puede tampoco asignar el valor de un puntero de un tipo a un puntero de otro tipo. Los resultados no estarían definidos.
- Prueba a intercambiar los tipos. Asigna por ejemplo **suma1** al puntero **p_oper2** ignorando la advertencia del compilador y observa el resultado. ¿Por qué ocurre esto?



2.6 Vectores y estructuras de punteros a funciones

- Por último, resulta también muy útil definir estructuras y vectores de funciones. Observa este ejemplo:

```
int (*las4reglas[])(int, int) =  
    {suma1, resta1, mult1, div1};
```

- Utiliza el ejemplo en tu programa definiendo las funciones que faltan. 
- Ten en cuenta que lo que se ha definido es ya una variable de tipo vector a punteros a función y que se ha inicializado directamente en la declaración, luego puedes usar sus cuatro elementos sin más.

- Una estructura con campos de tipo puntero a función sería:

```
struct s_4reglas {  
    int (*suma)(int x1, int x2);  
    int (*resta)(int x1, int x2);  
    int (*multiplicacion)(int x1, int x2);  
    int (*division)(int x1, int x2);  
};
```

- Declárala fuera de `main()` y define una variable `mis4reglas` dentro del `main()`:

```
struct s_4reglas mis4reglas =  
    {suma1, resta1, mult1, div1};
```

- Usa la estructura como has usado el vector. En este caso, la suma se invocaría así:

```
mis4reglas.suma(a,b)
```



- En este caso todos los campos de la estructura son punteros a función del mismo tipo. Esto es siempre así en un vector, pero en una estructura, obviamente, no tiene por qué.

2.7 Listas

Vamos a llevar a cabo ahora la implementación de una lista doblemente enlazada. Partiremos para ello del siguiente programa:

```
#include <stdio.h>
```

```


struct elemento {
    int entero;  float real;
};

int main(int argc, char *argv[]) {
    int i;
    struct elemento vector[3]= { {1, 1.5}, {2, 2.5}, {3, 3.5} };

    for (i=0; i<3; i++) {
        printf("Elemento %d: entero %d, real %.2f\n",
            i, vector[i].entero, vector[i].real);
    } /* endfor */

    return 0;
} /* end main */

```

- Compila, ejecuta el programa y comprueba su resultado. 
- Modifica el programa para utilizar una lista doblemente enlazada en vez de un vector. Para ello deberás:
 - Declarar una estructura **lista** que contenga los campos **cabeza** y **cola** para apuntar al primer y último elemento.
 - Modifica la estructura **elemento** para que contenga un puntero al elemento anterior y al siguiente.
 - Haz una función para añadir elementos por la cabeza y otra por la cola. Deberán recibir un puntero a la estructura **lista** y un puntero al elemento a añadir.
 - Para insertar un nuevo elemento en la lista primero habrá que reservar un área de memoria para él. Esto se puede llevar a cabo con la función **malloc(3)** (**p = malloc(sizeof(double));** nos devuelve en **p** un puntero a zona de memoria del tamaño de un **double**, es decir, 8 bytes) y con el operador **sizeof** (nos devuelve el tamaño de un tipo de datos).

2.8 Problema

Vamos a llevar a cabo la implementación de una lista genérica de figuras. Para ello crearemos varios tipos de figuras y las funciones asociadas.

- Crear las estructuras de datos para almacenar la información sobre las dimensiones de un cuadrado, un triángulo y un círculo.
- Implementar funciones asociadas a cada figura que reciban como único parámetro un puntero a una figura genérica (**void ***) y devuelvan el área de dicha figura.
- Crear una estructura de tipo **nodo** que tenga un puntero al siguiente, al anterior, un puntero genérico a una figura y un puntero a la función que calcula el área.

Hacer un programa que construya una lista con varias figuras y que posteriormente imprima información sobre el área de cada figura. Se puede ampliar el programa

añadiendo una función que imprima información sobre cada figura (nombre, dimensiones, etc.).