Programación I

Práctica #6 **Curso 2024-2025** **1º curso**

Punteros. Paso de parámetros por referencia

Introducción

En esta práctica introduciremos el concepto de gestión dinámica de memoria, utilizada en aquellas situaciones en las que, en tiempo de compilación, no sabemos la cantidad de memoria que necesitamos para almacenar alguna información.

Esta forma de gestión de la memoria se realiza a través de los punteros. Estudiaremos su declaración, su sintaxis y su uso.

Además, aprenderemos la estrecha relación existente entre los "arrays" y los punteros.

Finalmente, también aprenderemos a usar las diferentes formas de pasar parámetros a una función: por **valor** y por **referencia**.

Objetivos

Esta práctica tiene como objetivos fundamentales:

- Empezar a utilizar punteros.
- Trabajar con el concepto de gestión dinámica de memoria.
- Entender la relación entre arrays y punteros en C.
- Comprender las diferencias entre el paso de parámetros por valor y por referencia.

Planificación

Sesión 6 de laboratorio (3 horas):

1. Realización de los ejercicios 1 a 3.
2. Realización de los ejercicios 4 y 5.
3. Realización de los ejercicios 6 a 8.

Trabajo fuera del aula:

- Terminar los ejercicios no completados en el aula.

Punteros

Un puntero es una variable que, en vez de contener el valor de un dato relevante a nivel de aplicación, contiene la dirección de la memoria que ocupa otra variable. El tipo de esa otra variable se llama tipo base del puntero.

Para utilizarlos, disponemos de dos operadores:

- Operador de dirección (`&`): devuelve la dirección de memoria que ocupa una determinada variable.
- Operador de indirección (`*`): accede al contenido de una variable de la que conocemos su dirección.

Ejercicio 1:

Ve al directorio pr06.

Crea un programa en C (`ej1.c`) con el siguiente código. Ejecútalo. Explica los resultados obtenidos.

```
#include <stdio.h>

int main() {
  int a;
  int *b;

  b = &a;
  a = 5;
  printf ("a vale: %d\n", a);
  *b = 7;
  printf ("a vale: %d\n", a);
  return 0;
}
```

Gestión dinámica de memoria

Cuando definimos una variable de cualquier tipo, sea básico o estructurado, el compilador se encarga de reservar el espacio de memoria necesario para almacenar el valor correspondiente.

Sin embargo, existen situaciones en las que no podemos saber la cantidad de memoria que necesitamos para almacenar alguna información en el momento en que se compila nuestro programa. Por ejemplo, porque dicha cantidad depende de un dato que introduce el usuario cuando se ejecuta el programa.

En esos casos, debemos reservar la memoria que necesitemos en tiempo de ejecución del programa. Esto es lo que llamamos gestión dinámica de memoria, y se basa en el uso de punteros.

Para realizar una gestión dinámica de memoria se utilizan dos funciones básicas:

- `malloc(<expresión>)`: Esta función permite reservar memoria. La expresión debe evaluar a un valor entero que indicará el número de bytes de memoria que queremos reservar. Si esta función se ejecuta correctamente devuelve la dirección del primer byte del bloque de memoria reservado (es decir, un puntero).

- `free(<puntero>)`: Esta función libera un espacio de memoria **previamente** reservado cuando ya no es necesario. El puntero apunta a la dirección del comienzo del espacio de memoria que se quiere liberar.

En la expresión utilizada para calcular la cantidad de memoria a reservar se suele utilizar el operador:

- `sizeof(<tipo>)`: devuelve el número de bytes necesarios para almacenar un elemento del tipo indicado.

Aritmética de punteros

Cuando se incrementa (o decrementa) un puntero, este contendrá la dirección de memoria del siguiente (o anterior) elemento de su tipo base:

```
float *p;
p++; // desplaza p al siguiente float (4 bytes después)
```

Al sumar (o restar) un entero `n` a un puntero `p`, éste avanza (o retrocede) un número de posiciones de memoria equivalente a `n` veces el tamaño de su tipo base. Si `p` apunta a un `float`, `p+2` apunta 2 floats más adelante (8 bytes), y no 2 bytes más adelante:

```
p+=5; // desplaza p el equivalente a 5 float (20 bytes después)
```

La resta de punteros permite obtener el número de elementos del tipo base que hay entre ambos; NO es la diferencia entre sus valores:

```
float *p, *t;
p – t es el número de floats entre p y t
*p - *t es la diferencia entre el valor almacenado en la dirección a la que apunta p y el
valor almacenado en la dirección a la que apunta t
```

Arrays y punteros

A modo de ejemplo, para declarar una variable dinámica llamada "`tabla`", formada por un número **indeterminado** de enteros (es decir, el tipo base es `int`), lo haríamos con la siguiente instrucción:

```
int  *tabla;
```

Para reservar tanta memoria dinámica como sea necesaria para almacenar `N` enteros en dicha variable "`tabla`", lo haríamos con la siguiente instrucción:

```
tabla = (int *) malloc (N*sizeof(int));
```

Y, para asignar a una variable entera "`a`" el contenido de la variable que ocupa la 7ª posición de dicha "`tabla`", podemos usar tanto la notación de array como la de puntero; es decir, lo podríamos hacer con cualquiera de las siguientes instrucciones:

```
a = tabla[6];
```

o

```
a = *(tabla + 6);
```

Ejercicio 2:

Crea un programa en C (`ej2.c`) con el siguiente código.

Antes de ejecutarlo, ¿qué crees que debe mostrar por pantalla?

Ejecútalo. Explica los resultados obtenidos.

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

int main (void) {

    char nombre[] = "Jacobo";
    char *pc1 = &nombre[0];
    char *pc2 = nombre;
    int *pi1, *pi2, i;

    if (pc1 == pc2) printf ("pc1 y pc2 son iguales\n");
    else printf ("pc1 y pc2 son distintos\n");
    pc1[0] = 'X';
    *(pc2+5) = 'e';
    printf ("El nombre es: %s\n", nombre);

    pi1 = malloc (5 * sizeof(int));
    pi2 = pi1;
    for (i=0; i<5; i++) pi1[i] = i*10+1;
    for (i=0; i<5; i++) printf ("%d ", pi1[i]);
    printf ("\n");
    *(pi2+2) *= 5;
    pi2 += 3;

    printf ("pi1[2] = %d\n", pi1[2]);
    printf ("pi2 apunta a %d\n", *pi2);
    printf ("Entre pi1 y pi2 hay %ld enteros\n", pi2 - pi1);
    printf ("*pi2 - *pi1 vale: %d\n", *pi2 - *pi1);

    return 0;
}
```

Ejercicio 3:

Copia el ejercicio 2 de la Práctica #4:

```
usuario@maquina:~/pr06$ cp ../pr04/ej2.c ej3.c
```

Vamos a modificarlo, de manera que, ahora, utilice gestión dinámica de memoria.

Es decir, en vez de guardar siempre exactamente 5 elementos de tipo entero en un `array` de 5 enteros, ahora, cada vez que se ejecute, el programa dejará que sea el usuario el que decida el número de elementos que se van a guardar.

Para ello:

1º) El programa pedirá al usuario que introduzca un valor entero (**`N`**) entre 1 y 10. El programa deberá repetir la solicitud anterior hasta que se introduzca un valor correcto (pista: ya tienes el código que hace esto en el programa que escribiste para el ejercicio 4 de la Práctica #3).

2º) Una vez hemos obtenido un valor correcto, el programa deberá reservar **dinámicamente** memoria para almacenar **`N`** valores de tipo entero (empleando, para ello, la anteriormente mencionada función `malloc()`) y almacenará en esa memoria **`N`** valores aleatorios entre -10 y +10. Recuerda: al asignar los valores, puedes usar notación de array o de puntero.

3º) Por último, el programa imprimirá en pantalla los valores almacenados en memoria. Recuerda: al imprimir los valores, también puedes usar notación de array o de puntero.

```
Dame un entero [1-10]: 0
Dame un entero [1-10]: 15
Dame un entero [1-10]: 7
  3
  3
 -4
  9
  4
-10
 -7
```

Conceptos sobre parámetros de funciones

Parámetros por referencia

Cuando definimos un parámetro de una función, el compilador crea una variable local a dicha función (parámetro formal) y la inicializa con el valor (parámetro actual) que indicamos para dicho parámetro cuando llamamos a la función. Este comportamiento se denomina paso de parámetros **por valor**, e implica que los parámetros son sólo de entrada para las funciones: puesto que la variable correspondiente es local a la función, cualquier modificación que la función haga sobre el parámetro formal no será visible desde fuera de la función.

Veamos esto con un ejemplo:

```
void funcion (int a) {
    a += 10;
    return;
}

int main() {
    int x = 7;
    funcion(x);
    printf ("%d\n", x);  // Pinta un 7
    return 0;
}
```

Esto nos plantea dos problemas:

- qué hacer si queremos que la función modifique una variable recibida como parámetro, y
- qué hacer si queremos que una función devuelva varios resultados. Con lo que sabemos hasta ahora, nuestras funciones sólo pueden devolver un único valor, mediante la sentencia `return`.

La solución a estos dos problemas se denomina paso de parámetros **por referencia**, y consiste en pasar como parámetro un puntero a una variable donde queremos almacenar un resultado. Como antes, el compilador creará una copia del puntero que será local a la función (parámetro formal), pero como se inicializa al valor que pasamos como parámetro (en este caso, el parámetro actual es la dirección de una variable que no es local a la función), mediante ese puntero local, la función puede modificar la variable donde queremos dejar el resultado.

En el ejemplo anterior:

```
void funcion (int *a) {
  *a += 10;
  return;
}

int main() {
  int x = 7;
  funcion(&x);
  printf ("%d\n", x);  // Pinta un 17
  return 0;
}
```

Ejercicio 4:

Escribe una función que a partir de la base y la altura de un rectángulo (ojo: **NO** un triángulo rectángulo), calcule su área y su perímetro.
La función tendrá la siguiente declaración:

```
float rectangulo (float base, float altura, float *dir_area);
```

El último parámetro es un puntero que contendrá la dirección de la variable donde queremos que la función almacene el área del rectángulo. Además, la función entrega, como valor de retorno, el perímetro del rectángulo.

Escribe un programa que:
- pida por teclado la base y la altura de un rectángulo
- utilizando la función `rectangulo()`, calcule su área y su perímetro,
- y presente en pantalla su base, altura, área y perímetro.

```
Dame la base: 15.4
Dame la altura: 9.5

Altura:       9.50 cm
Base:        15.40 cm
Area:       146.30 cm x cm
Perimetro:   49.80 cm
```

Ejercicio 5:

Copia aquí el ejercicio 7 de la Práctica #5, y añádele la declaración y la definición de la función:

```
void take_string (const char *mensaje, char *palabra);
```

su funcionamiento es el del ejercicio 5 de la Práctica #3, salvo que:
- en vez de mostrar: `"Dame una cadena"`, mostrará el `mensaje` recibido como parámetro, y
- tras leer una cadena correcta, en vez de calcular y mostrar su longitud, copiará a la variable apuntada por `palabra` el contenido de la cadena leída.

Haz ahora que tu función `p_new()`, tras mostrar el mensaje `"New"`:
-llame a `take_string()`, pasándole la cadena `"Name"` como primer parámetro, y,
- después muestre la cadena que `take_string()` ha dejado en su segundo parámetro.

Ejercicio 6:

Haz una copia del ejercicio 5, y añádele la declaración y la definición de la función:

```
int take_int (const char *mensaje, int maximo);
```

su funcionamiento es el del ejercicio 4 de la Práctica #3, salvo que:
- en vez de mostrar: `"Dame un entero"`, mostrará el `mensaje` recibido como parámetro,
- tras leer un entero, comprobará si su valor está entre `0` y el valor recibido en el segundo parámetro (`máximo`), y
- cuando tenga un valor correcto, en vez de calcular y mostrar su raíz, retornará como **resultado** el valor del entero leído.

Haz ahora que tu función `p_vote()`, tras mostrar el mensaje `"Vote"`:
- llame a `take_int()`,
- pasándole la cadena `"Book"` como primer parámetro, y `20` como segundo parámetro,
- y, después muestre el entero retornado por `take_int()`.

Ejercicio 7:

Haz ahora una copia del ejercicio 6, y añádele la declaración y la definición de la función:

```
char take_char (const char *mensaje, const char *pattern);
```

su funcionamiento es el del ejercicio 6 de la Práctica #3, salvo que:
- en vez de mostrar: `"Dame un caracter"`, mostrará el `mensaje` recibido como 1º parámetro, y
- en vez de mostrar: `"NAVCSX"`, mostrará el `pattern` recibido como 2º parámetro, y
- tras leer un carácter correcto, en vez de mostrarlo, retornará como resultado dicho carácter pasado a mayúsculas.

Haz ahora que tu función `validate()`, en vez de:
- mostrar repetitivamente su parámetro
- y leer un carácter de la entrada de usuario mediante `scanf()`,

ahora:
-llame solo una vez a `take_char()`, para leer el carácter,
  - pasándole, como 1º parámetro, el parámetro que `validate()` ha recibido, y
  - pasándole la cadena `"YN"` como 2º parámetro, y
- después retorne 0 o 1, en función del valor de retorno de `take_char()`.

Ejercicio 8:

Haz ahora una copia del ejercicio 7, y
modifica tu función `main()` para que, en vez de:
- mostrar la cadena `"Enter operation"`
- y leer la opción elegida por el usuario mediante `scanf()`,

ahora:
-llame solo una vez a `take_char()`, para leer la opción,
  - pasándole la cadena `"Enter operation"` como 1º parámetro, y
  - pasándole la cadena `"NAVCSX"` como 2º parámetro, y
- después use el valor de retorno de `take_char()` como el selector para el `switch`.

Resumen

Los principales resultados esperados de esta práctica son:

- Manejar los punteros.
- Comprender el concepto de asignación dinámica de memoria
- Comprender las diferencias entre el paso de parámetros por valor y por referencia

Como posible línea de trabajo adicional del alumno, se propone la siguiente:

- Terminar los ejercicios no completados en clase.