Punteros, Arreglos y Cadenas en C

Edgardo Hames

Versión 1.1

Punteros y Direcciones

Organización de la memoria: Una computadora típica tiene un arreglo de celdas de memoria numeradas o direccionables que pueden ser manipuladas individualmente o en grupos contiguos. Una situación común es que un byte sea un caracter (`char`), un par de celdas de un byte sea tratado como un entero corto (`short`) y cuatro bytes adyacentes formen un entero largo (`long`). Es conveniente notar que esta disposición depende de la arquitectura subyacente y del compilador, ya que el único requerimiento del lenguaje es que:

$sizeof(char) = 1$ y $sizeof(short) \leq size(int) \leq sizeof(long)$

Un *puntero* es una variable que contiene la dirección de una variable. El uso de punteros conduce usualmente a un código más compacto, pero si se usan sin cuidado, es fácil crear punteros que apunten a un lugar inesperado.

Operadores

- El operador unario `&` retorna la dirección de un objeto, por lo tanto

  ```
  p = &i;
  ```

  asigna la dirección de la variable `i` a la variable puntero `p` y se dice que `p` *apunta a* `i`. Sólo se puede aplicar a variables y elementos de arreglos. Ver Fig. 1.

- El operador unario `*` es el operador de *indirección* o de *dereferenciación*; cuando se aplica a un puntero, accede al contenido de la dirección de memoria apuntada por éste. Ver ejemplo ej1.c.

  ```
  j = *p; /* j tiene ahora el valor de i */
  ```

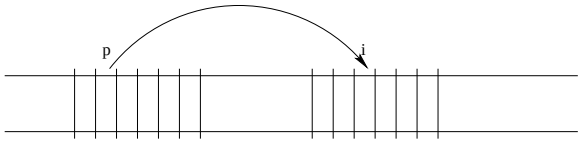

Figura 1: Organización de la memoria

- La declaración de `p` como variable puntero se obtiene anteponiéndole un `*`, por ejemplo:

  ```
  int *p;
  ```

  Esta notación intenta ser mnemónica, nos dice que la expresión `*p` es un `int`. Notar la restricción de que cada puntero apunta a un tipo particular de objeto: cada puntero apunta a un tipo de dato específico.[1]

- Como cualquier otra variable, es conveniente inicializar un puntero en su declaración. Si el valor no se conocerá hasta más adelante en el código, se debe usar `NULL`.

  ```
  int *p = NULL;
  ```

- Como los punteros son variables, pueden ser usados sin ser dereferenciados. Por ejemplo, si `p` y `q` son punteros a `int`,

  ```
  p = q;
  ```

  copia el contenido de `q` en `p`, haciendo que `p` apunte a lo mismo que apunta `q`.

- Los operadores `&` y `*` asocian con mayor precedencia que los operadores aritméticos. Por lo tanto, cualquiera de las siguientes instrucciones incrementa en 1 el valor apuntado por `p`.

  ```
  *p = *p + 1;
  *p += 1;
  ++*p;
  (*p)++; /* Notar el uso de paréntesis. */
  ```

[1] Esto es parcialmente cierto ya que un `puntero a void` se usa para contener cualquier tipo de punteros.

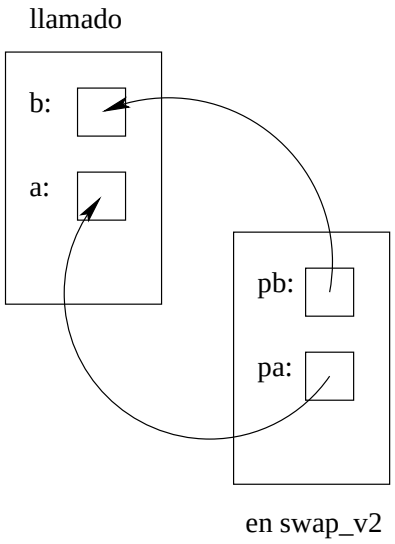

Figura 2: Paso de referencias a funciones.

¡Cuidado! Los operadores unarios `*` y `++` asocian de derecha a izquierda, por lo tanto, el uso de paréntesis suele ser necesario.

- El operador `++` aplicado a un puntero hace que apunte al elemento siguiente. Ver ejemplo ej2.c.
- Dado que en C, el paso de parámetros a funciones es por valor, cuando deseamos modificar un parámetro pasado a una función debemos usar punteros. Ver Fig. 2. Ver ejemplo ej3.c.

Punteros y Arreglos

En C, hay una relación muy fuerte entre punteros y arreglos. Cualquier operación que pueda ser lograda indexando un arreglo también puede ser conseguida con punteros.

La declaración

```
int a[10];
```

define un arreglo de tamaño 10, o sea un bloque de 10 objetos consecutivos nombrados `a[0]`, `a[1]`, ..., `a[9]`. Ver Fig. 3.

La notación `a[i]` hace referencia al `i`-ésimo elemento del arreglo. Supongamos que `pa` es un puntero a enteros, declarado como

```
int *pa;
```

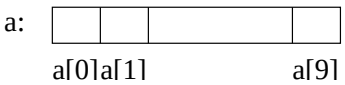

Figura 3: Arreglo de 10 caracteres.

entonces la asignación

```
pa = &a[0];
```

hace que `pa` apunte al elemento cero de `a`, o sea `pa` contiene la dirección de `a[0]`. El nombre de un arreglo es un puntero a su primer elemento.

Si `pa` apunta a un elemento particular de un arreglo, entonces por definición `pa+1` apunta al siguiente elemento, `pa+i` apunta `i` elementos más allá y `pa-1` apunta `i` elementos antes. Por lo tanto si `pa` apunta a `a[0]`,

```
*(pa + 1);
```

referencia al contenido de `a[1]`, `pa+i` es la dirección de `a[i]` y `*(pa+i)` es el contenido de `a[i]`.

Hagamos algunas cuentas sencillas ...

$$a[i] = *(pa + i) = *(i + pa) = i[a]$$

¿Será cierto lo que nos dicen nuestras clases de álgebra?

`Ver ejemplo ej4.c`.

Declaración de tipos y variables con punteros

C no tiene un tipo "puntero a". Algunos programadores parecen necesitar fuertemente la presencia de ese tipo y por lo tanto declaran sus variables puntero de la siguiente manera:

```
int* p = NULL;
```

Si bien no hay un error sintáctico en esa declaración, podría conducir a que el programador descuidado se equivoque al intentar declarar una nueva variable puntero `q`:

```
int* p = NULL, q = NULL; /* ERROR! q es un int !! */
```

Por esta misma razón, tampoco es una buena práctica, declarar tipos usando el comando `#define` del preprocesador. Veamos un ejemplo:

```
#define int * PINT;

PINT p = NULL, q = NULL;
```

El preprocesador transformará este texto[2] en:

```
int * p = NULL, q = NULL;
```

C provee una construcción para declaración de nuevos tipos: `typedef`. La sintaxis del comando es:

typedef tipo-anterior nuevo-tipo

Ahora sí podemos definir el tipo "puntero a int":

```
typedef int * pint;

pint p = NULL, q = NULL;
```

Tanto `p` como `q` son punteros a int. `typedef` hace más que un `#define`: construye un nuevo nombre para un tipo.

En nuestros programas, también nos interesará indicar que ciertos valores son constantes; es decir, no cambian durante la ejecución del programa. Para ello, usamos el calificador `const`.

```
const int V = 5; /* V es una constante con valor 5*/
```

La declaración de punteros y constantes da lugar a combinaciones que son tratados en las subsecciones siguientes.

Punteros a valores constantes

Los punteros a valores constantes nos sirven para indicar que no se harán modificaciones sobre lo apuntado. La declaración tiene esta forma:

*const tipo-apuntado * variable;*

He aquí un ejemplo:

```
int a = 5, b = 7;
const int *p = &a;
```

Podemos modificar el valor de `p`, pero no el valor apuntado:

```
p = &b; /* Ahora p apunta a b */
```

[2]Los comandos del preprocesador no son código C.

Pero...

```
*p = 3; /* Error! Lo apuntado por p es constante! */
```

Deberá notarse el carácter documental de const en las interfaces de programación de las bibliotecas.

```
size_t strlen(const char *str);
```

Con sólo ver esa declaración, el programador atento sabrá que `strlen` no modifica el valor apuntado por `str`. El argumento con que invocamos a esta función está a salvo de cualquier modificación, voluntaria o involuntaria, por parte del implementador de la biblioteca. Aprovechemos la capacidad del compilador de detectar algunos errores o descuidos!

Punteros constantes a valores variables

Ya que tenemos variables con valor constante y punteros a valores constantes, podemos preguntarnos cómo hacemos para declarar punteros con valor constante. El formato de la declaración es el siguiente:

*tipo-apuntado * const variable;*

Veamos un ejemplo:

```
int a = 1, b = 3;
int * const p = &a;
```

En este caso, no podemos modificar el valor de `p`, pero sí el de lo apuntado:

```
*p = 3;
```

Sin embargo...

```
p = &b; /* Error! p es constante! */
```

Punteros constantes a valores constantes

En el extremo de la constancia, encontramos a estos punteros. Su declaración sigue la forma:

*const tipo-apuntado * const variable;*

Y ya debería ser fácil darnos cuenta de un ejemplo.

```
int a = 7;
const int * const p = &a; /* Dale, cambialo si podés. */
```

Cadenas de caracteres

- Una variable de tipo *char* sólo puede almacenar un único caracter.

```
char c = 'A';
```

- Un *string* es una secuencia de caracteres. Por ej: "Hallo, Welt!"
- Una mala noticia: **C no soporta el tipo string**.
- En C un *string* se maneja como un array de caracteres muy especial.
- Generalmente, una variable con valor de tipo *string* se declara como un puntero de tipo `char *`. Veamos los siguientes ejemplos:

```
char saludo_arr[] = "Salut, mundi.";
char *saludo_ptr = "Salut, mundi.";

printf("Saludo con array: %s\n", saludo_arr);
printf("Saludo con puntero: %s\n", saludo_ptr);
```

La similitud entre ambas declaraciones y un uso semejante, puede conducirnos a pensar que son equivalentes. Sin embargo, hay una diferencia muy importante entre ellas.

La variable `saludo_arr` es un array de tamaño 13+1 (se suma 1 para el carácter terminador `'\0'`). La variable `saludo_ptr` es un puntero que apunta a una dirección fija de memoria resuelta en tiempo de compilación. Los caracteres del array pueden ser modificados, pero no así los caracteres apuntados por el puntero. Se puede asignar luego una nueva dirección al puntero, pero no así al array. Veamos unos ejemplos que nos ayudarán a entender más esta diferencia.

```
saludo_arr[0] = 's'; /* Reasigna el elemento 0. */
saludo_ptr[0] = 's'; /* Indefinido :( */

saludo_ptr = "Hello, World." /* Reasigna el puntero */
saludo_arr = "Hello, World." /* Incorrecto! */
```

- Por **convención**, el caracter nulo `'\0'` marca el fin del string.

```
char *mensaje = {'H','o','l','a','\0'};
char *mensaje = "Hola";
```

- El caracter nulo `'\0'` es conceptualmente distinto de `NULL`, aunque tengan el mismo valor! Además, `'\0'` es distinto de `'0'`.

```
char *ptr;
char c;

if (c == '\0') {
       /* c es el caracter nulo. */
}
if (!c) {
       /* c es el caracter nulo. */
}
if (*ptr == '\0') {
       /* p apunta al caracter nulo. */
}
if (!*ptr) {
       /* p apunta a un caracter nulo. */
}
```

parecido pero distinto de...

```
char *ptr;

if (!ptr) {
       /* p es un puntero nulo. */
}
```

Problemas con la librería de cadenas de C

- El uso de `'\0'` para denotar el fin del string implica que determinar la longitud de la cadena es una operación de orden lineal *O(n)* cuando puede ser constante *O(1)*.
- Impone una interpretación del valor del caracter `'\0'`. Por lo tanto, `'\0'` no puede formar parte de un string.

- `fgets` tiene la inusual semántica de ignorar los `'\0'` que encuentra antes de `'\n'`.
- No hay administración de memoria y las operaciones provistas (`strcpy`, `strcat`, `sprintf`, etc.) son lugares comunes para el desbordamiento de buffers.
- Pasar `NULL` a la librería de strings de C provoca un acceso a puntero nulo.
- No hay detección de *aliasing* (superposición o punteros autoreferenciantes)

Hay un canción que hace *referencia* a los strings en C, se llama *"Lo que ves es lo que hay"*. [3]

Para concluir esta sección, estudiaremos un ejemplo de la biblioteca de *"strings"* de C. El prototipo de strcpy es el siguiente:

```
char *strcpy(char *dest, const char *src);
```

En este caso, sin ver la implementación y con lo aprendido en secciones anteriores, podemos deducir que `src` es un puntero a char y su contenido es constante. Es decir, no se modificará el *"string"* apuntado por `src`. En cambio, si podría modificarse el valor del puntero (que recuperará su valor original cuando el programa retorne del llamado a función). Esto permite al programador hábil usar aritmetica de punteros en la implementacion en lugar de indexar los punteros, consiguiendo programas más compactos, pero menos simples para leer.

```
char *strcpy (char *dest, const char *src)
{
        char *p = dest;
        while((*dest++ = *src++));
        return p;
 }
```

Será conveniente notar que esta función tiene un resultado indefinido[4] cuando algunos de sus argumentos toma el valor `NULL`.[5]

[3] Charly García, Álbum "El aguante" :-)

[4] Es muy común encontrar programas que terminan con una *"violación de segmento"* por esta razón. El autor sueña con escribir un compilador más salvaje que borre el disco duro cuando se desreferencia un puntero nulo.

[5] Queda como ejercicio para el lector corregir la función para que chequee el valor de sus parámetros.

La destreza y la memoria son buenas si van en yunta[6]

- Arrays de caracteres declarados como `const` no pueden ser modificados. Por lo tanto, es de muy buen estilo declarar strings no modificables de tipo `const char *`.
- La memoria asignada para un array de caracteres puede extenderse más allá del caracter nulo.

```
char ptr[20];

strcpy(ptr, "Hola, Mundo");
/* Sobran 2^3 caracteres: ptr[12] - ptr[19]. */
```

- Una fuente muy común de *bugs* es intentar poner más caracteres de los que caben en el espacio asignado. Recordemos hacer lugar para `'\0'` !
- Las funciones de la biblioteca NO toman en cuenta el tamaño de la memoria asignada. C asume que el programador sabe lo que hace... :-) Ver ejemplo ej5.c.
- Para muchas funciones de la forma `strXXX` existe una versión `strnXXX` que opera sobre los `n` primeros caracteres del array. Es recomendable usar estas funciones para evitar problemas desbordando arreglos.
- Notar que podemos usar un índice mayor a la longitud del string sin que nos dé error. Ver ejemplo ej5.c.

```
/* Probar en Haskell: "Hello, World!" !! 20 */
```

Funciones útiles (1)

Aquí se presentan algunas funciones que pueden resultar de utilidad y están documentadas en la sección 3 de las páginas del manual (*man pages*).

[6] Les Luthiers, La payada de la vaca.

- Funciones que operan sobre caracteres: [7] [8]

```
#include <ctype.h>

int isalnum (int c);
int isalpha (int c);
int isascii (int c);
int isblank (int c);
int iscntrl (int c);
int isdigit (int c);
int isgraph (int c);
int islower (int c);
int isprint (int c);
int ispunct (int c);
int isspace (int c);
int isupper (int c);
int isxdigit (int c);
```

- Funciones que operan sobre strings:

```
#include <string.h>

char *strcpy(char *dest, const char *orig);
char *strcat(char *dest, const char *src);
char *strstr(const char *haystack, const char *needle);
size_t strspn(const char *s, const char *acepta);
size_t strcspn(const char *s, const char *rechaza);
```

Referencias

- The C Programming Language - Brian Kernighan, Dennis Ritchie
- `http://bstring.sourceforge.net`

[7] Si operan sobre caracteres, ¿por qué no veo un sólo `char` en los prototipos?
[8] ¿no debieran retornar `bool` o algo similar?