Punteros, Arreglos y Cadenas en C

Edgardo Hames

Revisado y corregido por Natalia B. Bidart, 13-08-2006 y después por Daniel F. Moisset, 24-08-2007

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). Un puntero es una variable que contiene la dirección de una variable. El uso de punteros es de más bajo nivel, y ofrece más expresividad. A veces conduce a un código más compacto y en algunos casos más eficiente; la desventaja es que sin cuidado, es fácil cometer errores.

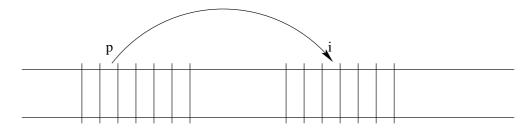


Figura 1: Organización de la memoria

Operadores:

■ El operador unario & retorna la dirección de memoria de una variable, 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 arreglos¹. Ver Fig. 1.

■ El operador unario * es el operador de *indirección* o de *dereferencia-ción*; cuando se aplica a un puntero, accede al contenido de la memoria apuntada por ese puntero. Ver ejemplo ej1.c.

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

■ El lenguaje C nos permite comprobar los tipos de los objetos apuntados por punteros. Esto se logra definiendo para cada tipo T (por ejemplo int), un tipo para los puntores asociados (por ejemplo, puntero a int). Para definir una variable que apunte a celdas con valores de tipo T, se declara esta como si fuera de tipo T, pero anteponiendo el caracter * al nombre de la variable. Por ejemplo:

```
int *p;
```

Esta notación intenta ser mnemónica, nos dice que la expresión *p es un int. Aunque cada puntero apunta a un tipo de dato específico, tambien hay un tipo llamado void * (sería un "puntero a void" que puede contener cualquier tipo de punteros.

■ Como cualquier otra variable, es conveniente inicializar un puntero en su declaración. Si el valor no se conocerá hasta más adelante en el código, una buena convención es usar NULL. NULL es un valor especial de puntero que no referencia a ninguna celda particular.

```
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, o sea, copia la dirección a la que apunta q en p, haciendo que p apunte a lo mismo que apunta q.

 $^{^1\}mathrm{es}$ decir, a cualquier cosa que indique una celda de memoria. Comparese con otros operadores como que pueden aplicarse a cualquier cosa que tenga un valor

■ Los operadores & y * asocian con mayor precedencia que los operadores aritméticos. Por lo tanto, cualquiera de las siguientes instrucciones incrementa en 1 el contenido de memoria apuntada por p.

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

¡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 a la celda siguiente. Esto suele ser útil al manipular arrays, ya que C nos garantiza que las celdas de un array son contiguas. Ver ejemplo ej2.c. No es útil en otros casos, ya que (excepto por el caso del array) una implementación de C puede elegir asignar cual celda corresponde a cual variable en forma bastante arbitraria.
- En C el paso de parámetros a funciones es por valor, cuando deseamos que una función pueda modificar una variable a elección del del llamador debemos usar punteros. Ver Fig. 2. Ver ejemplo ej3.c

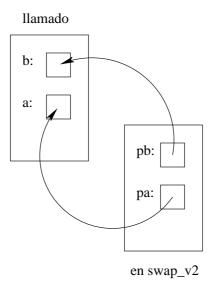


Figura 2: Paso de referencias a funciones.

Punteros y Arreglos

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

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

define un arreglo de tamaño 10, o sea un bloque de 10 enteros consecutivos que se acceden a través de a[0], a[1], ..., a[9]. Ver Fig. 3.

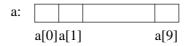


Figura 3: Arreglo de 10 caracteres.

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

int *pa;

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 (con lo cual podemos escribir el ejemplo anterior como pa = a).

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 adelante 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.

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 las operaciones sobre *strings* en realidad trabajan sobre arreglos de caracteres que siguen algunas convenciones. Esencialmente van los caracteres en el orden en el que se leen, el string no puede contener el caracter especial '\0', y al final de la cadena se agrega un '\0' (que no es parte de la cadena, pero que muchas funciones utilizan para saber donde termina el string).
- Generalmente, una variable de tipo *string* se declara como un puntero a char (en realidad, un puntero al primer elemento de un array de char, que ya vimos que en C es casi lo mismo). Ejemplos:

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

Hay una diferencia importante entre las dos declaraciones mostradas. En la primera, saludo_arr es un array, de tamaño 13+1 (el 1 suma por el caracter terminador '\0'). Los caracteres del array pueden ser cambiados, pero saludo_arr siempre va a apuntar a la misma zona de memoria.

En cambio, el puntero saludo_ptr puede ser eventualmente re-apuntado a otra zona de memoria, pero si se trata de modificar el contenido del string se obtiene un resultado indefinido².

Para acceder al string usamos la variable saludo_{arr,ptr}. Ejemplo usando printf:

```
printf ("El contenido de saludo_ptr es: %s", saludo_ptr);
```

 $^{^2\}mathrm{Cuando}$ el estándar de C dice "esto produce un resultado indefinido" quiere decir que su programa puede fallar intermitentemente, de formas distintas, y en condiciones poco esperadas.

■ Por convención, el caracter nulo \0 marca el fin del string.

```
char *mensaje = {'H', 'o', 'l', 'a', ', ', ', 'm', 'u', 'n', 'd', 'o', '\0'}; char *mensaje = "Hola, mundo";
```

■ Aunque a veces al caracter '\0' se le dice "caracter nulo" o inlcuso NUL (con una sola 'L'!), no tiene nada que ver con NULL. Son valores de distintos tipos. Además '\0' es distinto de '0' (el caracter del simbolo "cero" usado 2 veces en la cadena "007").

```
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 $\setminus 0$ 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).

- Se impone una interpretación para 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 del caracter \n.
- No hay administración de memoria y las operaciones provistas (strcpy, strcat, sprintf, etc.) son lugares comunes para el desbordamiento de buffers³.
- Pasar NULL a la librería de strings de C provoca un acceso a puntero nulo

Hay un canción que hace referencia a los strings en C, se llama "Lo que ves es lo que hay"⁴.

La destreza y la memoria son buenas si van en yunta

- Arrays de caracteres declarados como const no pueden ser modificados.
 Por lo tanto, es de muy buen estilo declarar strings que no deben ser modificados con tipo const char *.
- La memoria asignada para un array de caracteres puede extenderse más allá del caracter nulo. A veces eso es útil (por ejemplo si sabemos que vamos a querer agregar caracteres)

```
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! Estos *bugs* son una de las causas que pueden generar que la ejecución de un programa termine anormalmente con un segmentation fault.

³Un desbordamiento de buffer es cuando un programa escribe fuera de los limites de un array. Muchas veces pueden explotarse para tomar el control del programa y hacer que haga cosas que el programador no quería, como mandar SPAM desde nuestra cuenta de correo

⁴Charly García, Álbum "El aguante" :-)

- 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

Estas funciones están documentadas en la sección 3 de las man pages.

■ Funciones que operan sobre caracteres: ^{5 6}

```
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 isprint (int c);
int ispace (int c);
int isupper (int c);
int isupper (int c);
```

#include <ctype.h>

• Funciones que operan sobre strings:

⁵Si operan sobre caracteres, ¿por qué no hay char en los prototipos?

⁶¿no deberían retornar bool o algo similar?

```
#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://www.harpercollege.edu/bus-ss/cis/166/mmckenzi/contents. htm
- http://www.harpercollege.edu/bus-ss/cis/166/mmckenzi/lect12/ 112.htm
- http://bstring.sourceforge.net
- http://www.cs.princeton.edu/courses/archive/spring02/cs217/ asgts/ish/ish.html
- http://www.cs.princeton.edu/courses/archive/spring02/cs217/ asgts/ish/ishhints.html