Los punteros en C

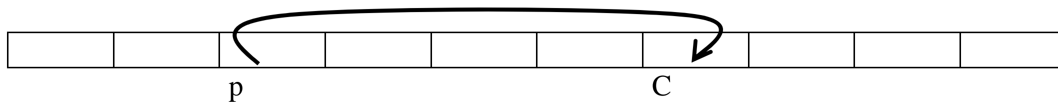
1 Introducción

¿Cómo se organiza la memoria asociada a un programa?
Como una colección de posiciones de memoria consecutivas. En ellas se almacenan los distintos tipos de datos, que ocupan, por ejemplo:

1 char = 1 byte
1 int = 2 bytes
1 float = 4 bytes

Un puntero no es más que una variable estática cuyo contenido es una dirección de memoria.

Los punteros, por lo tanto, guardan en dos o cuatro posiciones de memoria, la dirección de un conjunto de celdas.



donde:

pc es un puntero a carácter `char *pc;`
c es una variable de tipo carácter `char c;`

Inicialmente un puntero no apunta a ningún sitio. En C el valor NULL se reserva para indicar que el puntero está vacío (equivalente al nil de la teoría).

- **Operadores asociados a punteros**

&: me da la <u>dirección</u> de un objeto en la memoria.
Sólo sirve para posiciones de memoria (puede apuntar a variables o a vectores, pero no a constantes o expresiones). Ejemplo:

/* **prog1.c***/
```
pc = &c;
printf ("\nNo tiene lo mismo %c que %d", c, pc); /* Ojo, %d para pc*/
```

***:** me da el <u>contenido</u> de una posición de memoria (generalmente almacenada en un puntero). Se le llama también operador <u>indirección</u>. Por lo tanto es equivalente:

```
printf ("\nTiene lo mismo %d que %d", &c, pc); /* Direcciones */
printf ("\nTiene lo mismo %c que %c", c, *pc);  /* Caracteres */
```

Un puntero siempre está asociado a objetos de un tipo → sólo puede apuntar a objetos (variables o vectores) de ese tipo.

```
int *ip;     /* Sólo puede apuntar a variables enteras */
char *c;     /* Sólo puede apuntar a variables carácter */
double *dp, /* dp sólo puede apuntar a variables reales */
      atof (char *); /* atof es una función que devuelve un real
                        dada una cadena que se le pasa como
                        puntero a carácter*/
```

Los operadores * y & son unarios y tienen más prioridad a la hora de evaluarlos que los operadores binarios.

/* **prog2.c */**

```
int y, *ip;
y = 12;
printf ("\nValor de y %d, de ip %d", y, ip); /*sin asignar ip */
ip = &y;
*ip = *ip + 10;
printf ("\nValor de y %d, de *ip %d y de ip %d", y, *ip, ip);
y = *ip + 10;
printf ("\nValor de y %d, de *ip %d", y, *ip);
*ip += 1;
printf ("\nValor de y %d, de *ip %d", y, *ip);
```

Es necesario utilizar paréntesis cuando aparecen en la misma expresión que otros operadores unarios como ++ o --, ya que en ese caso se evaluarían de izquierda a derecha.

```
++*ip;
(*ip)++;
```

Dado que los punteros son variables, también pueden usarse como asignaciones entre direcciones. Así:

int *ip, *iq;
iq = ip; /* Indica que iq apunta a donde apunte el puntero ip. */

2 Los punteros y los argumentos a funciones

Recordemos la equivalencia:

Código Algorítmico	**Equivalente C**	**Ejemplo en C**
parámetro dato	paso por valor	int scanf (**"%d"**, &entero)
parámetro resultado	paso por referencia/ valor devuelto por una función	**int** scanf ("%d", &entero)
parámetro dato-resultado	paso por referencia	int scanf ("%d", **&entero**)

En C, por defecto, todos los parámetros a las funciones se pasan por valor (la función recibe una copia del parámetro, que no se ve modificado por los cambios que la copia sufra dentro de la función). Ejemplo: Intercambio de dos valores

{VERSION ERRONEA}

```
intercambia (int a, int b) {
int tmp;

tmp = a;
a = b;
b = tmp
}
```

{VERSION CORRECTA}

```
intercambia (int *a, int *b) {
int tmp;

tmp = *a;
*a = *b;
*b = tmp
}
```

Para que un parámetro de una función pueda ser modificado, ha de pasarse por referencia, y en C eso sólo es posible pasando la dirección de la variable en lugar de la propia variable.
Si se pasa la dirección de una variable, la función puede modificar el contenido de esa posición (no así la propia dirección, que es una copia).

3 Punteros y vectores

En C los punteros y los vectores están fuertemente relacionados, hasta el punto de que el nombre de un vector es en sí mismo un puntero a la primera (0-ésima) posición del vector. Todas las operaciones que utilizan vectores e índices pueden realizarse mediante punteros.

int v[10];

1	3	5	7	9	11	13	15	17	19
v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]	v[7]	v[8]	v[9]

v: designa 10 posiciones consecutivas de memoria donde se pueden almacenar enteros.

int *ip;

Designa un puntero a entero, por lo tanto puede acceder a una posición de memoria. Sin embargo, como se ha dicho antes `v` también puede considerarse como un puntero a entero, de tal forma que las siguientes expresiones son equivalentes:

ip = &v[0]	ip = v
x = *ip;	x = v[0];
*(v + 1)	v[1]
v + I	&v[i]

4 Aritmética de punteros

El compilador C es capaz de "adivinar" cuál es el tamaño de una variable de tipo puntero y realiza los incrementos/decrementos de los punteros de la forma adecuada. Así:

int v[10], *p;
p = v;

/* p Apunta a la posición inicial del vector*/
/* p + 0 Apunta a la posición inicial del vector*/
/* p + 1 Apunta a la segunda posición del vector*/
/* p + i Apunta a la posición i+1 del vector*/

p = &v[9]; /* p apunta ahora a la última posición (décima) del vector */

/* p - 1 Apunta a la novena posición del vector*/
/* p - i Se refiere a la posición 9 - i en v*/

Ejemplo de recorrido de un vector utilizando índices y punteros.

```
/* prog3.c*/
main(){
int v[10];
int i, *p;

for (i=0; i < 10; i++) v[i] = i;

for (i=0; i < 10; i++) printf ("\n%d", v[i]);

p = v;
for (i=0; i < 10; i++) printf ("\n%d", *p++);
/* Tras cada p++ el puntero señala a la siguiente posición en v */
}
```

C realiza las mismas operaciones con independencia del tipo de la variable.
De esta forma, si tenemos dos variables que son punteros:

int *ip, vi[10]; /* Necesita dos posiciones para representar un entero */
char *cp, vc[10]; /* Necesita una posición para representar un carácter */

ip = &vi[0];
cp = &vc[0];

Las expresiones:

ip + 1, ó ip++ Apunta al siguiente elemento (dos posiciones): vi[1]
cp + 1, ó cp++ Apunta al siguiente elemento (una posición): vc[1]

<u>Diferencia entre puntero y nombre de vector</u>: Mientras que un puntero es una variable y puede intervenir en asignaciones o incrementos, el nombre del vector no es una variable, por lo tanto, no pueden realizarse operaciones del tipo:

int v[10], *a;
v = a;
v++;

Dado que el nombre del vector es un puntero, puede utilizarse para pasar un vector como parámetro de una función (lo que permite modificar el contenido de esas posiciones de memoria adyacentes, o lo que es lo mismo, los elementos del vector).

char palabra[10];

scanf ("%s", palabra); /* Modifica las posiciones de memoria */
sort (palabra, 0, strlen(palabra));

/* sort va a modificar el contenido del vector, por lo tanto se pasará la dirección del primer elemento = palabra = &palabra[0].
strlen es una función de la librería estándar, cuyo macro es: **(int strlen (char *))**
La función nos devuelve la longitud de un vector de caracteres */

Recordatorio: Los vectores de caracteres terminan en el carácter '\0'.

Ejemplo 1: Realizar una función que sea equivalente a strlen. `int longitud (char *)`

char cadena[20];

gets(cadena);
printf ("\nLa longitud de la cadena %s es %d", cadena, longitud(cadena));

Ejemplo 2: Realizar una función equivalente que permita realizar a := b, cuando ambas son cadenas. `copia (char *, char *)`

char cadena[20], *copiada;

printf ("\nDame la primera cadena");
gets (cadena);
copia (cadena, copiada);
printf ("\nLa segunda cadena es %s", copiada);

Solución a 1):

```
main(){
char cadena[20];
int longitud();

printf ("\nDame una cadena (maximo 20): ");
gets(cadena);
printf ("\nLa cadena %s mide %d ", cadena, longitud(cadena));
}

int longitud(char *s){
int l;

l = 0;
while (*s++ != '\0') l++;

return l;
}
```

Solución a 2):

```
main(){
char uno[20], dos[20];

printf ("\nDame una cadena:");
gets(uno);

copia(uno, dos);

printf ("\nLa copia de %s\n es %s\n", uno, dos);
}

copia (char *s, char *p)
{
 while (*s) *p++ = *s++;
  *p = '\0';
}
```

5 Asignación dinámica de memoria

Hasta el momento sólo se ha visto cómo el lenguaje C define y utiliza los punteros para acceder a las posiciones de memoria asignadas a un programa. Sin embargo, no se ha tratado cómo "conseguir" nuevas posiciones de memoria (ateniéndose al lenguaje de la parte algorítmica: cómo funciona el Módulo de Gestión de la Asignación Dinámica de C).

En la <stdlib.h> están definidas las siguientes funciones:

- `void *calloc(size_t nobj, size_t size)`

  calloc obtiene (reserva) espacio en memoria para alojar un vector (una colección) de nobj objetos, cada uno de ellos de tamaño size. Si no hay memoria disponible se devuelve NULL. El espacio reservado se inicializa a bytes de ceros.
  Obsérvese que calloc devuelve un (void *) y que para asignar la memoria que devuelve a un tipo Tipo_t hay que utilizar un operador de ahormado: (Tipo_T *)

  Ejemplo:
  ```
  char * c;
  c = (char *) calloc (40, sizeof(char));
  ```

- `void *malloc(size_t size)`

  malloc funciona de forma similar a calloc salvo que: a) no inicializa el espacio y b) es necesario saber el tamaño exacto de las posiciones de memoria solicitadas.

  El ejemplo anterior se puede reescribir:
  ```
  char * c;
  c = (char *) malloc (40*sizeof(char));
  ```

- `void *realloc(void *p, size_t size)`

  realloc cambia el tamaño del objeto al que apunta p y lo hace de tamaño size. El contenido de la memoria no cambiará en las posiciones ya ocupadas. Si el nuevo tamaño es mayor que el antiguo, no se inicializan a ningún valor las nuevas posiciones. En el caso en que no hubiese suficiente memoria para "realojar" al nuevo puntero, se devuelve NULL y p no varía.
  El puntero que se pasa como argumento ha de ser NULL o bien un puntero devuelto por malloc(), calloc() o realloc().

```
#define N 10
#include <stdio.h>

main(){
char c, *cambiante;
int i;

i=0;
cambiante = NULL;

printf("\nIntroduce una frase. Terminada en [ENTER]\n");
while ((c=getchar()) != '\n') {
  if (i % N == 0){
   printf("\nLlego a %d posiciones y pido hasta %d", i, i+N);
   cambiante=(char *)realloc((char *)cambiante,(i+N)*sizeof(char));
   if (cambiante == NULL) exit(-1);
  }
  /* Ya existe suficiente memoria para el siguiente carácter*/
  cambiante[i++] = c;
}
```

```
/* Antes de poner el terminador nulo hay que asegurarse de que haya
suficiente memoria */
if ((i % N == 0) && (i != 0)){
      printf("\nLlego a %d posiciones y pido hasta %d", i, i+N);
      cambiante=realloc((char *) cambiante, (i+N)*sizeof(char));
        if (cambiante == NULL) exit(-1);
  }
cambiante[i]=0;

printf ("\nHe leido %s", cambiante);
}
```

- `void free(void *p)`

free() libera el espacio de memoria al que apunta p. Si p es NULL no hace nada. Además p tiene que haber sido "alojado" previamente mediante malloc(), calloc() o realloc().

→ NO MIRAR ESTA PARTE HASTA ENTENDER PERFECTAMENTE LO ANTERIOR←

El siguiente programa también soluciona el anterior problema 2. La diferencia entre ambas soluciones está en que el nuevo vector no tiene un tamaño fijo, sino que se le asigna en función del tamaño del vector original.

```
main(){
char uno[20], *dos;

printf ("\nDame una cadena:");
gets(uno);

copia2(uno, &dos);      /*Inicialmente dos no apunta a ningún sitio*/
                        /*En copia2 se modificará el valor de dos */
printf ("\nLa copia de %s\n es %s\n", uno, dos);
}

/* s es un puntero a char. Su valor (la dirección) no es modificada)
p es un puntero a un puntero a char. Su valor (un puntero a char,
inicialmente vacío) SI que se va a modificar dentro */

copia2 (char *s, char **p)
{
 int i;
 char *r;

 /* *p es el puntero al que se le asignan nuevas posiciones de memoria*/
 *p = (char *) calloc (strlen(s)+1, sizeof(char));

/* Utilizo r para cambiar el contenido de *p */
 r = *p;
 while (*s) *r++ = *s++;
 *r = 0;
}
```

Si no hubiese utilizado el puntero `r`, y hubiese hecho avanzar el puntero `*p` igual que se hizo en el ejemplo de `longitud()`, habría perdido las posiciones anteriores a `*p`. Otras dos formas de realizar el mismo ejercicio serían:

```c
main(){
char uno[20], *dos;
char tres[20], *cuatro;

printf ("\nDame una cadena:");
gets(uno);
copia3(uno, &dos);
printf ("\nLa copia de %s\n es %s\n", uno, dos);

printf ("\nDame otra cadena:");
gets(tres);
copia4(tres, &cuatro);
printf ("\nLa copia de %s\n es %s\n", tres, cuatro);
}

/*Utilizo un índice para recorrer *p y cambiar **p sin modificar *p */
copia3 (char *s, char **p)
{
 int i;

 *p = (char *) calloc (strlen(s)+1, sizeof(char));

 i=0;
 while (*s) (*p)[i++] = *s++;
 (*p)[i] = 0;
}

/*Utilizo el índice, pero accedo mediante *(*p + i) en vez de (*p)[i] */
copia4 (char *s, char **p)
{
 int i;

 *p = (char *) calloc (strlen(s)+1, sizeof(char));
 i=0;
 while(*s) {
 *(*p + i) = *s++;
 i++;
 }
 *(*p + i)= 0;
}
```