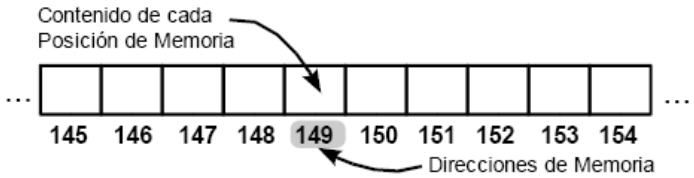
Punteros

¿Qué es un puntero?

En el famoso libro de Kernighan y Ritchie "El lenguaje de programación C", se define un puntero como «*una variable que contiene la dirección de una variable*». Tras esta definición clara y precisa, podemos remarcar que un puntero es un tipo especial de variable que almacena el valor de una dirección de memoria. La memoria de las computadoras está formada por un conjunto de bytes. Simplificando, cada byte tiene un número asociado; una dirección en esa memoria.

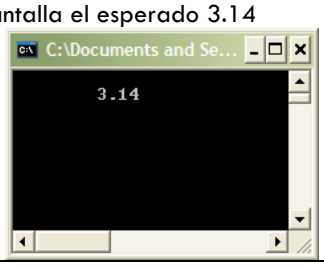
Representación de la Memoria



La representación anterior se utilizará a lo largo del texto para ilustrar los distintos ejemplos. De esta forma, si ejecutamos el siguiente ejemplo:

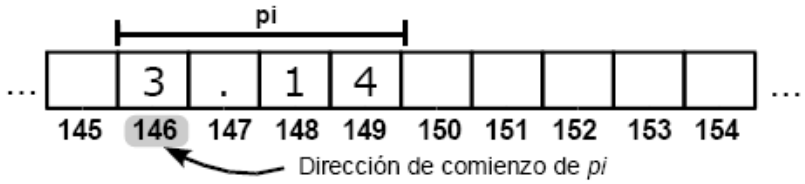
```
01 #include <stdio.h>
02 int main(void)
03 {
04 float pi=3.14;
05 printf ("%.2f\n", pi);
06 return 0;
07 }
```

Obtenemos en pantalla el esperado 3.14



Sin embargo, ¿qué operaciones realizó la computadora hasta mostrar el resultado?. En la línea 4, la instrucción reserva memoria para la variable pi. En este ejemplo, asumimos que un decimal de tipo float ocupa 4 bytes. Dependiendo de la arquitectura del computador, la cantidad de bytes requerida para cada tipo de datos puede variar. La Figura 2 muestra el estado de la memoria después de la declaración de la variable pi. La representación elegida facilita el seguimiento de los ejemplos, sin embargo, la representación real en la memoria de ordenador no tendría esa disposición.

Memoria reservada para una variable

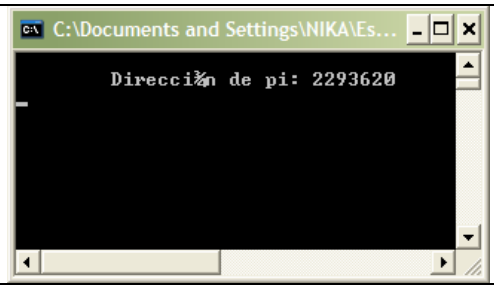


Cuando se utiliza pi en la línea 5, ocurren dos pasos diferenciables. En primer lugar, el programa busca la dirección de memoria reservada para pi (en nuestro ejemplo, sería la dirección 146). Hecho esto, en segundo lugar, el programa recupera el contenido de esa dirección de memoria. Así, distinguimos por un lado la dirección de memoria asignada a una variable, y por otro lado el contenido de la posición de memoria reservada.

Podemos acceder a la dirección de una variable utilizando el operador &.

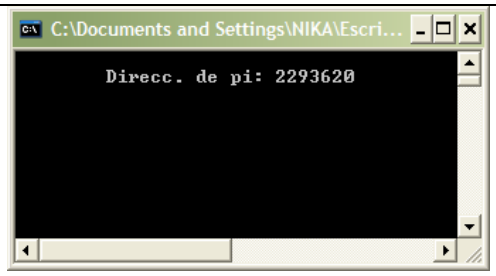
Mediante este operador, obtenemos la dirección donde se almacena la variable (¡un número!). En el ejemplo, se ha utilizado %u y una conversión forzada a entero sin signo para que la salida por pantalla sea más entendible. Se podía haber utilizado el conversor estándar para punteros %p y haber quitado la conversión forzada a entero sin signo.

```
01 #include <stdio.h>
02 int main(void)
03 {
04 float pi=3.14;
05 printf ("Dirección de pi: %u\n", (unsigned int)&pi);
06 return 0;
07 }
```



Ejecutando el programa anterior en la memoria del *ejemplo anterior*, debería mostrarnos la dirección 146. Recordemos que una dirección de memoria es únicamente un número. De hecho, podríamos almacenarla en una variable entera sin signo

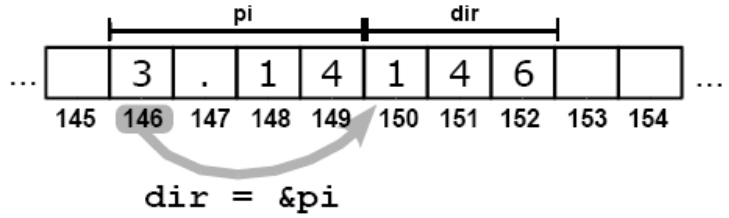
```
01 #include <stdio.h>
02 int main(void)
03 {
04 float pi=3.14;
05 unsigned int dir=(unsigned int)&pi;
06 printf ("Direcc. de pi: %u\n", dir);
07 return 0;
08 }
```



El código anterior demuestra que no hay nada de magia ni efectos especiales en el tema de las direcciones. Simplemente son números que pueden ser almacenados en variables enteras. Esto se puede ver gráficamente en la siguiente figura (suponiendo que las variables de tipo entero sin signo requieran 3 bytes en memoria).

***Nota**: Recordemos que el modificador unsigned simplemente indica que no se permiten números negativos. Las direcciones no pueden ser negativas.*

Uso del operador '&'



Lo que se quiere representar simplemente es que

mediante el operador & accedemos a la dirección de memoria de una variable

Esta dirección es un simple número entero con el que podemos trabajar directamente. Para diferenciar una variable entera que almacene valores de nuestro programa (por ejemplo, índices, contadores, valores...) de las variables que almacenan direcciones, en C se creó un nuevo tipo de variable para almacenar direcciones. Este tipo es, precisamente, el puntero.

Así, podemos tener algunas declaraciones de punteros como las siguientes:
char * nombre;
float * altura;

¿Cómo interpretar las líneas anteriores? Podemos leer que nombre es un entero, pero podemos añadir que es un entero especialmente diseñado para guardar la dirección de un carácter. Por su parte, altura es un entero, además es un entero especialmente diseñado para guardar la dirección de un float. Es lo que decimos resumidamente con "nombre es un puntero a char" o "altura es un puntero a float".

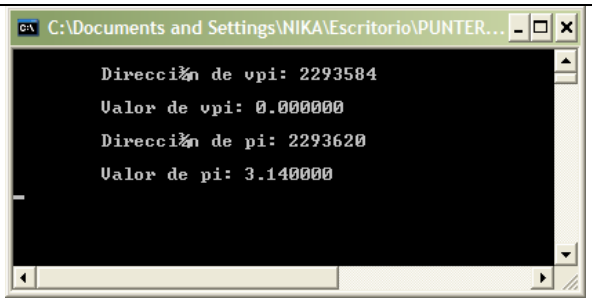
El operador * también se utiliza para recuperar el contenido de una posición de memoria identificada por una dirección.

No debemos confundir su uso en la declaración de un puntero, como hemos visto antes, con los casos en que se emplea para recuperar el contenido de una posición de memoria. Este uso lo veremos a continuación.

¿Para qué usar punteros?

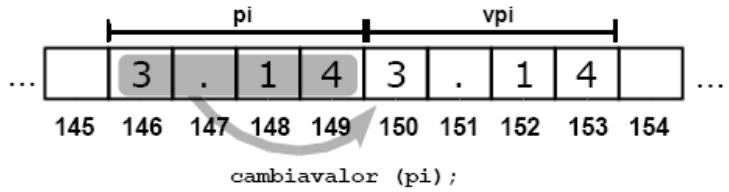
Hagamos un programa que cambie el valor de la variable pi del ejemplo anterior a cero. Para ello, realizaremos el código que se muestra en el Listado 4. En este caso intentamos pasar el parámetro vpi por referencia, para que cambie el contenido de pi dentro de la función. Si ejecutamos el programa, el resultado obtenido no es el esperado. ¿Por qué?

```
01 void cambiavalor(float vpi) {
02 vpi = 0;
03 printf ("Dirección de vpi: %u\n", (unsigned int) &vpi);
04 printf ("Valor de vpi: %f\n", vpi);
05 }
06 int main(void){
07 float pi=3.14;
08 cambiavalor(pi);
09 printf ("Dirección de pi: %u\n", (unsigned int) &pi);
10 printf ("Valor de pi: %f\n", pi);
11 return 0;
14 }
```



La ejecución del programa muestra que el parámetro **pi ha sido pasado por valor** (en lugar de por referencia).
Si hacemos un seguimiento más exhaustivo del programa, vemos que cuando llamamos a la **función cambiavalor** en la línea 8, el programa salta a la línea 1 para ejecutar la función. Allí, se crea una nueva variable llamada **vpi**, **y se copia el valor de pi en ella**. Esto se representa en el esquema siguiente:

Copia del valor de pi en vpi

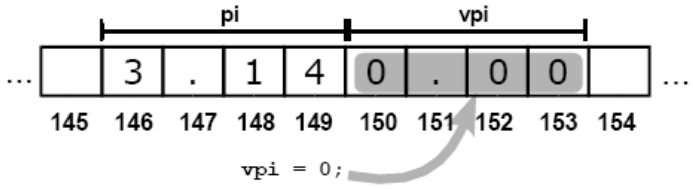


Queda clara la razón por la que **vpi** y **pi** tienen diferente dirección en memoria:

1. se ha realizado una copia del contenido de la memoria que tiene pi,
2. se ha llevado a otra zona de memoria y
3. se le ha puesto de nombre vpi.

De esta forma, cuando se asigna 0 a la variable vpi en la línea 2, se está asignando al contenido de memoria de **vpi**, no al de **pi**.

Cambio de valor de vpi

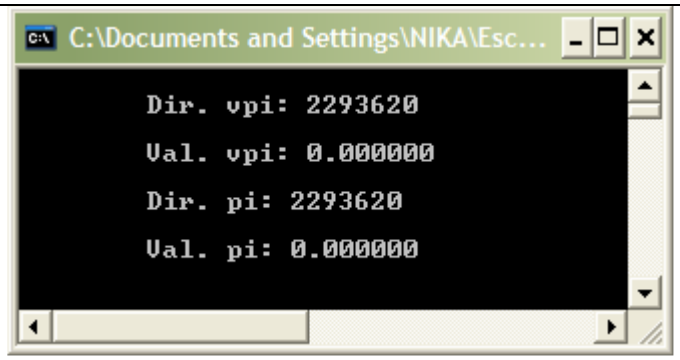


Los argumentos de las funciones en C se pasan siempre por valor. No existe una forma directa para que la función que se invoca altere una variable de la función que la llama.

Es imprescindible el uso de punteros en el caso de querer modificar el valor de una variable pasada como parámetro a una función.

Así, podemos modificar el programa, para que el paso del argumento se realice por referencia. Tendremos que utilizar los operadores * y & que hemos visto anteriormente. El nuevo programa sería:

```
00 #include <stdio.h>
01 void cambiavalor(float *vpi) {
02 *vpi = 0;
03 printf ("Dir. vpi: %u\n", (unsigned int) vpi);
04 printf ("Val. vpi: %f\n", *vpi);
05 }
06 int main(void) {
07 float pi=3.14;
08 cambiavalor(&pi);
09 printf ("Dir. pi: %u\n", (unsigned int)&pi);
10 printf ("Val. pi: %f\n", pi);
11 return 0;
12 }
```



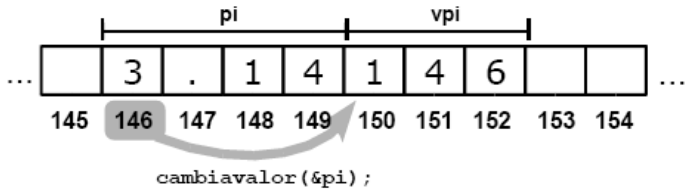
En esta nueva versión del programa, la **función cambiavalor**, recibe el parámetro como un puntero (línea 1). Le estamos diciendo al compilador que pase como argumento a la función una dirección de memoria.

Es decir, queremos que le pase un entero; ese entero será una dirección de memoria de un float. En un seguimiento en detalle del programa vemos que, en la línea 8, cuando llamamos a la **función cambiavalor**, le tenemos que pasar una dirección de memoria.

Para ello, tenemos que hacer uso del operador &, para pasarle la dirección de la variable pi.

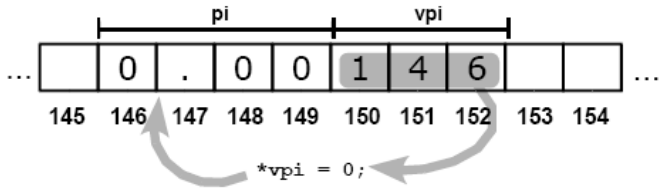
Con esto, cuando llamamos a la función cambiavalor en 1, lo que se hace es una copia de la dirección de memoria de pi en la variable vpi (ver Figura 6). Así, mediante el operador * podremos acceder al contenido de la posición de memoria de pi y cambiar su valor.

Paso por referencia de "pi"



En la línea 2 del programa se cambia el valor de la zona de memoria apuntada por vpi. Podemos leer esa instrucción como "asignamos cero al contenido de la dirección de memoria apuntada por vpi". Recordemos que esa zona de memoria tiene que ser de tipo float. En el ejemplo esta dirección de memoria es la 146. Podemos ver un esquema del resultado de la operación en la siguiente figura.

Asignación de 0 al contenido de memoria apuntado por vpi



Ahora, cuando acabe la **función cambiavalor** y retornemos a la función principal **main**, el valor de **pi** será 0.

Hemos realizado un paso de parámetro por referencia.

Aritmética de Punteros

Como hemos mencionado anteriormente, podemos considerar el tipo de un puntero como el tipo de datos que contiene la zona de memoria a la que éste apunta.

Es decir, si tenemos:
int *ptr;
*ptr = 2;

la primera línea indica que tenemos una variable llamada ptr, cuyo contenido es una dirección de memoria a una zona donde hay un entero. En la segunda línea asignamos al contenido de esa dirección de memoria el valor 2.

Una operación interesante que podemos realizar con punteros es incrementarlos. Obviamente, la operación de incremento en un puntero no es igual que en una variable normal. Un incremento en un puntero hará que apunte al siguiente elemento del mismo tipo de datos que se encuentre en memoria.

Incremento de punteros

En el ejemplo de la figura inicialmente el puntero ptr apunta al tercer entero almacenado en el bloque de memoria representado (suponiendo que cada entero ocupa 3 posiciones de memoria).

Así, la asignación *ptr=168 hará que el contenido de esa posición de memoria sea 168.

Si incrementamos el puntero (¡cuidado!, no hablamos del contenido del puntero, sino del propio puntero), haremos que apunte al siguiente entero que hay en memoria. Este incremento del puntero no implica sumar 1 a la dirección que contiene el puntero.

En este ejemplo, sería necesario sumar 3 (tres posiciones de memoria que ocupa el entero para acceder al siguiente).

Naturalmente esta "suma de posiciones de memoria" no la tendrá que hacer el programador. Será el compilador el que se encargue de realizar las cuentas para ver cuántas posiciones de memoria debe avanzar.

En general, el compilador tendrá que sumar tantas posiciones de memoria como requiera el tipo de datos sobre el que se haya definido el puntero.

En el caso de que incrementemos alegremente el puntero y nos salgamos de la zona de memoria que tengamos reservada, recibiremos la visita de nuestro amigo «segmentation fault».