# Punteros

Los punteros son variables que almacenan direcciones, lo que los diferencia del resto de las variables que almacenan valores de un tipo determinado (números, caracteres o valores booleanos). Más allá de esta importante diferencia, los punteros no encierran ningún "misterio": son recipientes donde se puede escribir o leer datos, lo mismo que se hace con el resto de las variables.

Es probable que la dificultad que se presenta para entender los punteros resida en que solemos escuchar o leer que un puntero "apunta" a algo (una dirección de memoria), lo cual les confiere cualidades "mágicas" (o peligrosas) que no poseen el resto de las variables, que sólo pueden contener un valor; contener algo es menos riesgoso que apuntar a algo.

Para "amigarnos" con los punteros (son usados con frecuencia en los programas en C e imprescindibles para la asignación dinámica de memoria) analicemos los siguientes fragmentos de código:

```
int var, *pint;

cout<<var<<endl;

cout<<pint;
```

¿Qué salida obtendremos en la pantalla?. En ambos casos "basura" porque ni a var ni a pint se les ha asignado nada. Para **var** esa "basura" será cualquier valor entero (ya que var puede almacenar valores enteros); para **pint** la "basura" será cualquier dirección (ya que pint almacena direcciones). Hasta acá entonces las dos variables se comportan de igual manera. Sigamos:

```
int var, *pint;

cout<<&var<<endl;

cout<<&pint;
```

El **símbolo &**, cuando lo ponemos antes del nombre de una variable ya declarada, se convierte en **el operador de dirección:** nos da la dirección de la variable; la salida que

obtendremos entonces serán dos direcciones. Por ejemplo:

```
      0x22ff44

      0x22ff40
```

que son las direcciones físicas que ambas variables ocupan en la memoria. Nuevamente el comportamiento de las variables es el mismo. Sigamos:

```
var=15;

pint=&var;

cout<<var<<endl;

cout<<pint;
```

La salida en este caso será:

```
      15

      0x22ff44
```

15 es el valor que se le asignó a var, esto es, el valor que contiene var.

0x22ff44: es el valor que se le asignó a **pint** (**&var**, la dirección de var), esto es, el valor que contiene pint. Otra vez el comportamiento es idéntico. Sigamos:

```
int var, *pint;

var=15;

pint=&var;

cout<<*pint;
```

La salida por pantalla será 15, que es el valor que contiene var. Veamos el código en detalle:

```
var=15;// se asigna el valor 15 a var

pint=&var; // se asigna a pint la dirección de var

cout<<*pint; // se muestra lo que hay en la dirección que contiene pint
```

Y acá aparece la diferencia: si se escribiera ***var** el compilador daría un error, ya que el operador ***** sólo puede anteponerse a una variable puntero. **Este operador se llama operador de indirección, y permite acceder al contenido de una dirección**. Como **pint** contiene la dirección de **var**, ***pint** nos lleva al contenido de var.

Del mismo modo, en el siguiente fragmento de código se modifica el valor de **var** por medio de **pint**.

```
int var, *pint;

pint=&var;

*pint=15;

cout<<var;
```

Cuando escuchamos o leemos que un puntero “apunta” a una dirección, nos estamos refiriendo a la dirección que ese puntero contiene; en el ejemplo visto como **pint** “apunta” a **var**, por medio de ***pint** podemos leer o escribir en **var**.

Cuando se declara una variable de puntero, debe especificarse además a qué tipo de dato apunta. Por ejemplo:

```
int *pi;

float *pf;

char *pc;
```

**pi** es la declaración de un puntero a entero, y por lo tanto podrá almacenar la dirección física de una variable de tipo entero;

**pf** es la declaración de un puntero a float, y por lo tanto podrá almacenar la dirección física de una variable de tipo float;

**pc** es la declaración de un puntero a char, y por lo tanto podrá almacenar la dirección física de una variable de tipo char.

Independientemente del tipo de datos del puntero, su tamaño será siempre de 4 bytes. En los punteros el tipo de datos no determina el tamaño de la variable, ya que en cualquiera de los casos lo que almacenará será una dirección. Más adelante, en el título **Aritmética de punteros**, se explicará la razón por la cual es necesario definir el tipo de los punteros.

Veamos un ejemplo de uso de punteros

```
#include<iostream>

using namespace std;

int main()

  {

  int entera, *apunto;   // declaración de una variable entera, y un
puntero a entero

  entera=2;                  // se asigna 2 a la variable entera

  apunto=&entera;       //se asigna la dirección de entera a apunto

  cout<<*apunto;// se imprime el contenido de la dirección a la que apunta
el puntero

  cout<<apunto;// se imprime lo que contiene apunto

 cout<<&entera;// se imprime la dirección de entera

  system(“pause”);

  return 0;

  }
```

Supongamos que la variable **entera** ocupa la dirección FFF4. Al anteponerle el operador de dirección & a esa variable, y asignarla a **apunto**, **apunto** almacenará el valor FFF4.

Luego con el operador de indirección ***** antepuesto a **apunto**, podemos imprimir lo que contiene la dirección a la cual apunta, esto es, el valor 2 de **entera**.

También podríamos cambiar el valor de **entera** a través de su dirección. La siguiente línea de código cambiaría el valor de **entera** a 4:

***apunto= 4;**

Finalmente, si imprimimos el valor de **apunto**, y el de la dirección de **entera** (**&entera**) el programa nos informará FFF4.

Trabalenguas aparte, lo importante es lo siguiente:

> Las variables tiene un nombre por medio del cual el programador trabaja con ellas, y una dirección física en la memoria de la máquina. El nombre que le damos nos interesa sólo a nosotros, ya que nos simplifica el trabajo. El programa ejecutable no trabaja con los nombres, sino que traduce todo a posiciones relativas de memoria, es decir a direcciones físicas.
>
> C/C++ nos permite trabajar directamente con las direcciones que las variables ocuparán (en tiempo de ejecución) en la memoria. Podemos entonces escribir y leer una determinada posición de la memoria por medio de los punteros. Recordar que los punteros son variables que almacenan direcciones; si ponemos delante de una variable puntero el operador * podremos leer o escribir en la dirección que está almacenada en el puntero (la dirección a la que apunta el puntero).

Los punteros son particularmente útiles cuando necesitamos que una función modifique una o más variables locales a otra función. En ese caso la función propietaria de las variables que se quieren modificar llama a la función que las modificará enviando como parámetros las direcciones de esas variables. Siempre que se envía un parámetro por dirección, este debe ser recibido por una variable puntero.

```
int main(){

        int var1, var2;

       //se llama a modifica_var() y se envía las direcciones de variables

      modifica_var(&var1, &var2);
      cout<<var1<<”\t”<<var2;//se ven los valores asignados en la función
      return 0;

      }



void modifica_var(int *pvar1, int *pvar2){

      *pvar1=8;

      *pvar2=10;

      }
```

**Nota**: sólo deben enviarse direcciones como parámetros cuando se necesite que la función llamada modifique el valor de una o más variables.

**Punteros y vectores**

Un caso especial de los punteros son los nombres de los vectores. Cuando declaramos un vector, por ejemplo int v[10], reservamos espacio en memoria para 10 variables enteras a las que por conveniencia trataremos conjuntamente, y a las que podremos acceder para leer o escribir mediante el nombre del vector y el subíndice correspondiente (p.e. v[0]=3). No podemos hacer ninguna operación si no decimos específicamente cuál de los elementos del vector queremos mostrar o escribir.

El nombre del vector, sin subíndice, almacena la dirección de inicio de ese vector en la memoria. Como ya sabemos que el único tipo de variable que puede almacenar una dirección es una variable puntero, el nombre del vector es entonces un puntero, pero a diferencia del resto de los punteros es un **puntero constante**: contiene la dirección de memoria donde empieza el vector y no puede modificarse.

Siguiendo el razonamiento anterior, podemos hacer la siguiente relación algebraica:

v=&v[0]

aplicando el operador * a ambos miembros

*v=*(&v[0])

como * y & se anulan entre sí, se simplifican y nos queda:

***v=v[0]**

**Nota:** el espacio de memoria que se reserva al declarar un vector se ubica en posiciones contiguas (una al lado de la otra). Su tamaño será igual a la cantidad de componentes multiplicado por el tamaño del tipo de datos.

**Aritmética de punteros**

C/C++ nos permiten sumar o restar valores enteros a direcciones. Ahora, ¿qué significa incrementar en 1 (o cualquier otro número entero) una dirección?. Lógicamente no será lo mismo que si el incremento se produce en una variable entera, ya que ésta última almacena números y la otra direcciones. Por ejemplo:

```
 int a, v[10], *p;
 a=0;
 a=a+1
 cout<<a;         //se imprimirá un 1
 v[0]=1;
 v[1]=2;
 p=v;
 cout<<*p;        //como p guarda la dirección de v, se imprimirá el
contenido de v[0], o sea 1
 p=p+1;// o p++, se incrementa en 1(decimal) la dirección
 cout<<*p;// como el puntero se incrementó en 1, ahora contiene la
dirección de v[1]. Se //imprimirá 2.
```

¿Por qué pasa esto?. C tiene la capacidad de analizar “contextualmente” sus operadores. Si el incremento es sobre una variable entera, se incrementará el valor que contiene. Si el incremento es sobre una variable puntero se incrementará la dirección que contiene. Al sumar 1 a una dirección, C incrementa esa dirección una cantidad igual a la cantidad de bytes necesarias para almacenar el tipo de dato con el que se está trabajando.

Si el incremento es mayor a 1, la dirección se incrementará una cantidad de bytes equivalente a lo que surja de multiplicar el incremento por la cantidad de bytes del tipo de dato declarado para el puntero.

Por ejemplo:

Supongamos que al vector v se le ha asignado como dirección de inicio la dirección 1000.

Luego al asignar p=v, p también almacenará el valor de dirección 1000.

Al incrementar p en 1 (p++, o su equivalente p=p+1) el resultado no será 1001, sino 1004, ya que lo que sucede es:

p=1000+(1*4), siendo 4 el número de bytes necesarios para almacenar un valor de tipo int.

Lo mismo sucede cuando el incremento o decremento tiene cualquier valor entero distinto de 1.

Volvamos ahora a nuestra relación algebraica inicial:

v=&v[0]
v+1=&v[1]

v+2=&v[2]

generalizando (y siendo n un entero definido entre 0 y 9 para este caso), podemos expresar que

v+n=&v[n]

Si ahora aplicamos * en ambos miembros

*(v+n)=*&v[n]

como * y & se anulan, nos queda:

***(v+n)=v[n]**

Con lo cual se demuestra que es posible trabajar con vectores por medio de subíndices, o a través de la notación de punteros indistintamente.

**Nota**: en la demostración anterior el signo = se usó con el significado que se le da en matemática, la igualdad, y no como lo utiliza C para operador de asignación.

Como se dijo anteriormente, la única forma de enviar como parámetro un vector es por dirección. La función que llama envía la dirección de inicio del vector (que no es otra cosa que el nombre del vector sin subíndice), y la función llamada recibe esa dirección en una variable puntero. Luego se puede usar dentro de la función indistintamente la notación de subíndices o de punteros.

Veamos una función que ya conocemos porque la utilizamos muchas veces en Programación I:

```
//void cargarVector(int v[], int tam) cambiamos a

void cargarVector(int *v, int tam){

            int i;

            for(i=0;i<tam;i++){

                  cin>>v[i];

      }

 }
```

La función recibe la dirección de inicio del vector y su tamaño. La almacena en un puntero (v) y en una variable entera (tam) respectivamente. Luego asigna a cada posición un valor

que se ingresa por teclado. La función podría reescribirse de la siguiente manera, cambiando la notación de subíndice por la de punteros:

```
void cargarVector(int *v, int tam){
            int i;
            for(i=0;i<tam;i++){
                  cin>>*(v+i);
      }
}
```

Veamos ahora el siguiente programa, que usa la función cargar()

```
int main(){
      int v[5],*p, i;
      cargar(v,5);
      p=v;
      for(i=0;i<5;i++) cout<<p[i]<<endl;
      return 0;
}
```

¿Funcionará?¿Podemos usar subíndices en un puntero?. La respuesta a las 2 preguntas es **SI**, ya que **siempre** usamos subíndices con punteros:

- los nombres de los vectores son punteros constantes.

También lo hemos venido usando en todas las funciones de vectores. Analicemos de nuevo la función cargarVector()

```
void cargarVector(int *v, int tam){
            int i;
            for(i=0;i<tam;i++){
                  cin>>v[i];
      }
}
```

En todos los casos, lo que el compilador hace es traducir la notación de subíndices (más sencilla, descriptiva y cómoda para escribir programas) a la de punteros, es decir:

- toma la dirección que contiene el puntero
- a esa dirección le suma el número contenido dentro de los corchetes
- como sabemos por aritmética de punteros, esa suma incrementa la dirección una cantidad equivalente al tamaño del tipo de datos multiplicando por el número contenido entre los corchetes
- en la dirección resultante escribe el número (o lee si fuera el caso) que se le asigna por teclado (o cualquier otra operación que se quiera realizar)

Esa es la razón por la que en todos los casos, el primer elemento del vector se identifica con el 0, y el último con la cantidad de elementos menos 1.

En el siguiente programa, se usan notaciones equivalentes para mostrar por pantalla el contenido de un vector previamente cargado.

```
int main(){
      int v[5],*p, i, *j;
      cargarVector(v,5);
      p=v;
      for(i=0;i<5;i++)        cout<<v[i]<<endl;
      for(i=0;i<5;i++)        cout<<p[i]<<endl;
      for(i=0;i<5;i++)        cout<<*(v+i)<<endl;
      for(i=0;i<5;i++)        cout<<*(p+i)<<endl;
```

```
	for(j=v;j<v+5;j++)	cout<<*j<<endl;


	for(i=0;i<5;i++)	cout<<*p++<<endl;
	for(i=0;i<5;i++)	cout<<*v++<<endl;
	return 0;
}
```

Nota: en la explicación se utilizó un vector de tipo entero. Las consideraciones expuestas son aplicables para vectores de cualquier tipo de dato.