

CAPÍTULO 7

© F.J.Ceballos/RA-MA

PUNTEROS, REFERENCIAS Y GESTIÓN DE LA MEMORIA

Un puntero es una variable que contiene la *dirección* de memoria de un dato o de otra variable que contiene al dato. Quiere esto decir que el puntero apunta al espacio físico donde está el dato o la variable. Un puntero puede apuntar a un objeto de cualquier tipo, como por ejemplo, a una estructura o a una función. Los punteros se pueden utilizar para referenciar y manipular estructuras de datos, para referenciar bloques de memoria asignados dinámicamente y para proveer el paso de argumentos por referencia en las llamadas a funciones.

Cuando se trabaja con punteros son frecuentes los errores por utilizarlos sin haberles asignado una dirección válida; esto es, punteros que por no estar iniciados apuntan no se sabe a dónde, produciéndose accesos a zonas de memoria no permitidas. Por lo tanto, debe ponerse la máxima atención para que esto no ocurra, iniciando adecuadamente cada uno de los punteros que utilicemos.

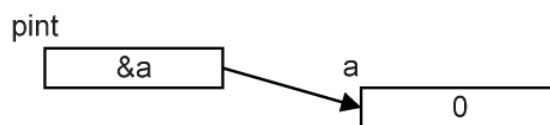
CREACIÓN DE PUNTEROS

Un puntero es una variable que guarda la dirección de memoria de otro objeto. Para declarar una variable que sea un puntero, la sintaxis es la siguiente:

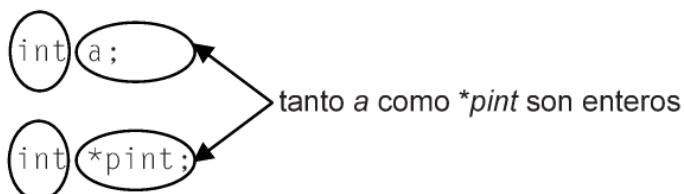
```
tipo *var-puntero;
```

En la declaración se observa que el nombre de la variable puntero, *var-puntero*, va precedido del modificador ***, el cual significa “*puntero a*”; *tipo* especifica el tipo del objeto apuntado, puede ser cualquier tipo primitivo o derivado. Por ejemplo, si una variable *pint* contiene la dirección de otra variable *a*, entonces se dice que *pint* apunta a *a*. Esto mismo expresado en código C++ es así:

```
int a = 0;      // "a" es una variable entera
int *pint;     // pint es un puntero a un entero
pint = &a;     // pint igual a la dirección de a; entonces,
               // pint apunta al entero "a"
```



La declaración de *a* ya nos es familiar. La declaración del puntero *pint*, aunque también la hemos visto en más de una ocasión, quizás no estemos tan familiarizados con ella como con la anterior, pero si nos fijamos, ambas declaraciones tienen mucho en común. Observe la sintaxis:



Observamos que **pint* es un nemotécnico que hace referencia a un objeto de tipo **int**, por lo tanto puede aparecer en los mismos lugares donde puede aparecer un entero. Es decir, si *pint* apunta al entero *a*, entonces **pint* puede aparecer en cualquier lugar donde puede hacerlo *a*. Por ejemplo, en la siguiente tabla ambas columnas son equivalentes:

<pre>#include <iostream> using namespace std; int main() { int a = 0; a = 10; a = a - 3; cout << a << endl; }</pre>	<pre>#include <iostream> using namespace std; int main() { int a = 0, *pint = &a; *pint = 10; *pint = *pint - 3; cout << *pint << endl; }</pre>
---	---

Suponiendo definida la variable *a*, la definición:

```
int *pint = &a;
```

es equivalente a:

```
int *pint;
pint = &a;
```

En conclusión **pint* es un entero que está localizado en la dirección de memoria almacenada en *pint*.

El espacio de memoria requerido para un puntero es el número de bytes necesarios para especificar una dirección máquina, que normalmente son 4 bytes.

Un puntero iniciado correctamente siempre apunta a un objeto de un tipo particular. Un puntero no iniciado no se sabe a dónde apunta.

Operadores

Los ejemplos que hemos visto hasta ahora ponen de manifiesto que en las operaciones con punteros intervienen frecuentemente el operador *dirección de* (&) y el operador de *indirección* (*).

El operador unitario & devuelve como resultado la dirección de su operando y el operador unitario * interpreta su operando como una dirección y nos da como resultado su contenido (para más detalles, vea el capítulo *Elementos del lenguaje C++*). Por ejemplo:

```
#include <iostream>
using namespace std;

int main()
{
    // Las dos líneas siguientes declaran la variable entera a,
    // los punteros p y q a enteros y la variable real b.
    int a = 10, *p, *q;
    double b = 0.0;
    q = &a; // asigna la dirección de a, a la variable q.
           // q apunta a la variable entera a
    b = *q; // asigna a b el valor de la variable a
    *p = 20; // error: asignación no válida
           // ¿a dónde apunta p?
    cout << "En la dirección " << q << " está el dato " << b << endl;
    cout << "En la dirección " << p << " está el dato " << *p << endl;
}
```

En teoría, lo que esperamos al ejecutar este programa es un resultado análogo al siguiente:

```
En la dirección 0x22ff6c está el dato 10
En la dirección 0x42b000 está el dato 20
```

Pero en la práctica, cuando lo ejecutamos ocurre un error por utilizar un puntero *p* sin saber a dónde apunta. Sabemos que *q* apunta a la variable *a*; dicho de otra forma, *q* contiene una dirección válida, pero no podemos decir lo mismo de *p* ya que no ha sido iniciado y, por lo tanto, su valor es desconocido para nosotros (se dice que contiene basura); posiblemente sea una dirección ocupada por el sis-

tema y entonces lo que se intentaría hacer sería sobrescribir el contenido de esa dirección con el valor 20, lo que ocasionaría graves problemas. Quizás lo vea más claro si realiza la definición de las variables así:

```
int a = 10, b = 0, *p = 0, *q = 0;
// ...
*p = 20;
```

El error se producirá igual que antes, pero ahora es claro que *p* almacena la dirección 0, dirección en la que no se puede escribir por estar reservada por el sistema.

Importancia del tipo del objeto al que se apunta

¿Cómo sabe C++ cuántos bytes tiene que asignar a una variable desde una dirección? Por ejemplo, observe el siguiente programa:

```
#include <iostream>
using namespace std;

int main()
{
    int a = 10, *q = 0;
    double b = 0.0;
    q = &a; // q apunta al entero a
    b = *q; // asigna a b el valor de a convertido a double
    cout << "En la dirección " << q << " está el dato " << b << endl;
}
```

Se habrá dado cuenta de que en la sentencia $b = *q$, *b* es de tipo **double** y *q* apunta a un **int**. La pregunta es: ¿cómo sabe C++ cuántos bytes tiene que asignar a *b* desde la dirección *q*? La respuesta es que C++ toma como referencia el tipo del objeto definido para el puntero (**int**) y asigna el número de bytes correspondiente a ese tipo (cuatro en el ejemplo, no ocho).

OPERACIONES CON PUNTEROS

A las variables de tipo puntero, además de los operadores **&**, ***** y el operador de asignación, se les puede aplicar los operadores aritméticos **+** y **-** (sólo con enteros), los operadores unitarios **++** y **--** y los operadores de relación.

Operación de asignación

El lenguaje C++ permite que un puntero pueda ser asignado a otro puntero. Por ejemplo:

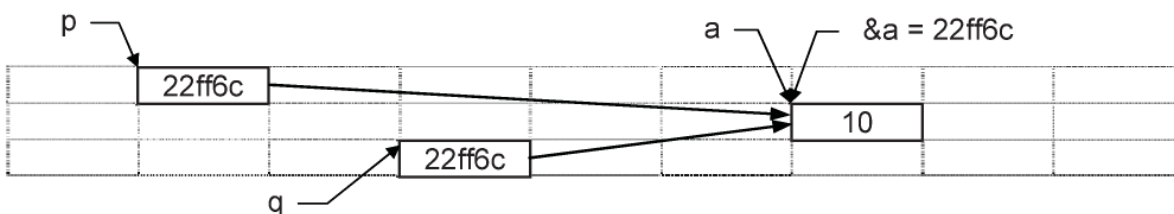
```
#include <iostream>
using namespace std;

int main()
{
    int a = 10, *p, *q;
    p = &a;
    q = p; // la dirección que contiene p se asigna a q
    cout << "En la dirección " << q << " está el valor " << *q << endl;
}
```

Ejecución del programa:

En la dirección 0x22ff6c está el valor 10

Después de ejecutarse la asignación $q = p$, p y q apuntan a la misma localización de memoria, a la variable a . Por lo tanto, a , $*p$ y $*q$ son el mismo dato; es decir, 10. Gráficamente puede imaginarse esto así:



Operaciones aritméticas

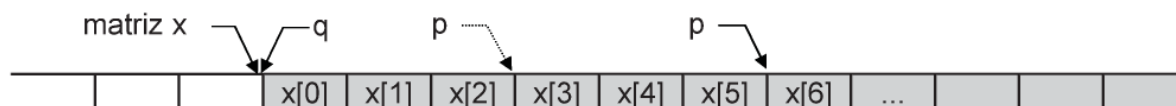
A un puntero se le puede sumar o restar un entero. La aritmética de punteros difiere de la aritmética normal en que aquí la unidad equivale a un objeto del tipo del puntero; esto es, sumar 1 implica que el puntero pasará a apuntar al siguiente objeto, del tipo del puntero, más allá del apuntado actualmente.

Por ejemplo, supongamos que p y q son variables de tipo puntero que apuntan a elementos de una misma matriz x :

```
int x[100];
int *p, *q; // declara p y q como punteros a enteros
p = &x[3]; // p apunta a x[3]
q = &x[0]; // q apunta a x[0]
```


La operación $p + n$, siendo n un entero, avanzará el puntero n enteros más allá del actualmente apuntado. Por ejemplo:

```
p = p + 3; // hace que p avance tres enteros; ahora apunta a x[6]
```



Así mismo, la operación $p - q$, después de la operación $p = p + 3$ anterior, dará como resultado 6 (elementos de tipo `int`).

La operación $p - n$, siendo n un entero, también es válida; partiendo de que p apunta a `x[6]`, el resultado de la siguiente operación será el comentado.

```
p = p - 3; // hace que p retroceda tres enteros; ahora apuntará a x[3]
```

Si p apunta a `x[3]`, $p++$ hace que p apunte a `x[4]`, y partiendo de esta situación, $p--$ hace que p apunte de nuevo a `x[3]`:

```
p++; // hace que p apunte al siguiente entero; a x[4]
p--; // hace que p apunte al entero anterior; a x[3]
```

No se permite sumar, multiplicar, dividir o rotar punteros y tampoco se permite sumarles un real.

Veamos a continuación algunos ejemplos más de operaciones con punteros. Definimos la matriz `x` y dos punteros `pa` y `pb` a datos de tipo entero.

```
int x[100], b, *pa, *pb;
// ...
x[50] = 10;
pa = &x[50]; // a pa se le asigna la dirección de x[50]
pb = &x[10]; // a pb se le asigna la dirección de x[10]

b = *pa + 1; // el resultado de la suma del entero *pa más 1
             // se asigna a b; es decir, b = x[50] + 1

b = *(pa + 1); // el siguiente entero al apuntado por pa,
               // es asignado a b; esto es, b = x[51]

(*pb)--; // x[10] se decrementa en una unidad

x[0] = *pb--; // a x[0] se le asigna el valor de x[10] y pb
              // pasa a apuntar al entero anterior (a x[9])
```

Comparación de punteros

Cuando se comparan dos punteros, en realidad se están comparando dos enteros, puesto que una dirección es un número entero. Esta operación tiene sentido si ambos punteros apuntan a elementos de la misma matriz. Por ejemplo:

```
int n = 10, *p = 0, *q = 0, x[100];
// ...
p = &x[99];
q = &x[0];
// ...
if (q + n <= p)
    q += n;
if (q != 0 && q <= p) // 0 es una constante que identifica
    q++;              // a un puntero nulo
```

La primera sentencia **if** indica que el puntero *q* avanzará *n* elementos si se cumple que la dirección *q+n* es menor o igual que *p*. La segunda sentencia **if** indica que *q* pasará a apuntar al siguiente elemento de la matriz si la dirección por él especificada no es nula y es menor o igual que la especificada por *p*.

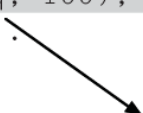
Punteros genéricos

Un puntero a cualquier tipo de objeto puede ser convertido al tipo **void ***. Por eso, un puntero de tipo **void *** recibe el nombre de puntero genérico. Por ejemplo:

```
void fnx(void *, int);
int main()
{
    int x[100], *q = x; // se declara q y se le asigna la dirección x
                        // (el nombre de una matriz es la dirección
                        // de comienzo de la matriz)

    // ...
    fnx(q, 100); // implícitamente se convierte (int *) a (void *)
    // ...
}

void fnx(void *p, int n)
{
    // ...
}
```



En cambio, el compilador C++ no puede asumir nada sobre la memoria apuntada por un **void ***. Se necesita, por lo tanto, una conversión explícita. El operador **dynamic_cast**, como examina el tipo del objeto apuntado, no puede realizar la conversión, por lo que se tiene que utilizar **static_cast**:

```
void fnx(void *p, int n)
{
    int *q = static_cast<int *>(p);
    // ...
}
```

En este ejemplo se observa que para convertir un puntero genérico a un puntero a un **int** se ha realizado una conversión explícita (conversión *cast*).

Puntero nulo

En general, un puntero se puede iniciar como cualquier otra variable, aunque los únicos valores significativos son **0** o la dirección de un objeto previamente definido. El lenguaje C++ garantiza que un puntero que apunte a un objeto válido nunca tendrá un valor cero. El valor cero se utiliza para indicar que ha ocurrido un error; en otras palabras, que una determinada operación no se ha podido realizar.

En general, no tiene sentido asignar enteros a punteros porque quien gestiona la memoria es el sistema operativo, y por lo tanto es él el que sabe en todo momento qué direcciones están libres y cuáles están ocupadas. Por ejemplo:

```
int *px = 103825; // se inicia px con la dirección 103825
```

La asignación anterior no tiene sentido porque, ¿qué sabemos nosotros acerca de la dirección 103825?

Punteros y objetos constantes

Una declaración de un puntero precedida por **const** hace que el objeto apuntado sea una constante, no sucediendo lo mismo con el puntero. Por ejemplo:

```
char a[] = "abcd";
const char *pc = a;
pc[0] = 'z';    // error: modificación del objeto
pc = "efg";     // correcto: modificación del puntero
```

Si lo que se pretende es declarar un puntero como una constante, procedemos así:

```
char a[] = "abcd";
char* const pc = a;
pc[0] = 'z';    // correcto: modificación del objeto
pc = "efg";     // error: modificación del puntero
```


Para hacer que tanto el puntero como el objeto apuntado sean constantes, procederemos como se indica a continuación:

```
char a[] = "abcd";
const char* const pc = a;
pc[0] = 'z'; // error: modificación del objeto
pc = "efg"; // error: modificación del puntero
```

REFERENCIAS

Una referencia es un nombre alternativo (un sinónimo) para un objeto. La forma de declarar una referencia a un objeto en general es:

tipo& referencia = objeto

Para más detalles, véase el apartado *Operador referencia a* en el capítulo *Elementos del lenguaje C++*.

No se debe aplicar la aritmética de punteros a las referencias (por ejemplo, comparar dos referencias) ni tomar la dirección de una referencia. De hecho, estas últimas operaciones no generarán un error, pero porque, como ya sabemos, dichas operaciones se realizan sobre las variables referenciadas. Por ejemplo:

```
int a[5] = {10, 20, 30, 40, 50}; // matriz a
int& rUltimo = a[4]; // referencia al último elemento de a
int& rElemento = a[0]; // referencia al primer elemento de a
int *p;

while ( rElemento <= rUltimo )
{
    p = &rElemento;
    cout << *p << ' ';
    rElemento++;
}
```

En este ejemplo, la expresión *rElemento <= rUltimo* no compara direcciones, compara los valores de *a[0]* y *a[4]*; la expresión *p = &rElemento* siempre toma la dirección de *a[0]*; la sentencia *cout << *p* siempre escribe el valor de *a[0]*; y la expresión *rElemento++* siempre incrementa el valor de *a[0]*. Según lo expuesto, el resultado al ejecutar el código anterior será *10, 11, 12, 13, ..., 50*.

Paso de parámetros por referencia

Pasar parámetros por referencia significa que lo transferido no son los valores, sino las direcciones de las variables que contienen esos valores, con lo que los

parámetros actuales se verán modificados si se modifican los contenidos de sus correspondientes parámetros formales.

Para pasar una variable por referencia, podemos utilizar una de las dos formas siguientes:

1. Pasar la dirección del parámetro actual a su correspondiente parámetro formal, el cual tiene que ser un puntero (vea *Pasando argumentos a las funciones* en el capítulo *Estructura de un programa*).

```
insertar( &ficha1 );
// ...
void insertar( ficha* pf )
{
    // La estructura de datos ficha1, apuntada por pf, puede
    // ser modificada por esta función.
}
```

2. Declarar el parámetro formal como una *referencia* al parámetro actual que se quiere pasar por referencia, para lo cual hay que anteponer el operador **&** al nombre del parámetro formal. Por ejemplo:

```
insertar( ficha1 );
// ...
void insertar( ficha& rf )
{
    // La estructura de datos ficha1, referenciada por rf, puede
    // ser modificada por esta función.
}
```

Una llamada a la función *insertar(ficha&)* como la siguiente daría lugar a un error durante la compilación, porque no se pueden definir referencias a constantes.

```
const ficha f1 = { ... };
insertar( f1 ); // error
```

Una referencia a un objeto constante proporciona la eficiencia de los punteros en el paso de argumentos (no se hace una copia) y seguridad impidiendo que el argumento pasado se modifique, característica implícita cuando el argumento se pasa por valor. El siguiente ejemplo utiliza una referencia a una constante:

```
void insertar( const ficha& rf )
{
    // La estructura de datos ficha, referenciada por rf, no puede
    // ser modificada por esta función, por ser constante.
}
```

PUNTEROS Y MATRICES

En C++ existe una relación entre punteros y matrices tal que cualquier operación que se pueda realizar mediante la indexación de una matriz se puede hacer también con punteros.

Para clarificar lo expuesto, analicemos el siguiente programa que muestra los valores de una matriz, primero utilizando indexación y después con punteros.

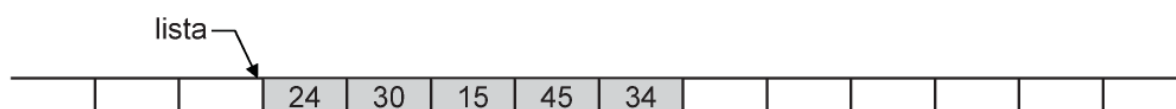
```
// Escribir los valores de una matriz.
// Versión utilizando indexación.
#include <iostream>
using namespace std;

int main()
{
    int lista[] = {24, 30, 15, 45, 34};
    for (int ind = 0; ind < 5; ind++)
        cout << lista[ind] << " ";
    cout << endl;
}
```

Ejecución del programa:

24 30 15 45 34

En este ejemplo se ha utilizado la indexación, expresión `lista[ind]`, para acceder a los elementos de la matriz `lista`. Cuando C++ interpreta esa expresión sabe que a partir de la dirección de comienzo de la matriz, esto es, a partir de `lista`, tiene que avanzar `ind` elementos para acceder al contenido del elemento especificado por ese índice. Dicho de otra forma, con la expresión `lista[ind]` se accede al contenido de la dirección `lista+ind` (ver *Operaciones aritméticas con punteros*).



Veamos ahora la versión con punteros:

```
// Escribir los valores de una matriz.
// Versión con punteros.
#include <iostream>
using namespace std;

int main()
{
    int lista[] = {24, 30, 15, 45, 34};
    int *plista = &lista[0];
```

```

    for (int ind = 0; ind < 5; ind++)
        cout << *(plista+ind) << " "; // equivalente a plista[ind]
    cout << endl;
}

```

Ejecución del programa:

24 30 15 45 34

Esta versión es idéntica a la anterior, excepto que la expresión para acceder a los elementos de la matriz es ahora **(plista+ind)*.

La asignación *plista = &lista[0]* hace que *plista* apunte al primer elemento de *lista*; es decir, *plista* contiene la dirección del primer elemento, que coincide con la dirección de comienzo de la matriz; esto es, con *lista*. Por lo tanto, en lugar de la expresión **(plista+ind)*, podríamos utilizar también la expresión **(lista+ind)*. Según lo expuesto, las siguientes expresiones dan lugar a idénticos resultados:

*lista[ind], *(lista+ind), plista[ind], *(plista+ind)*

El mismo resultado se obtendría con esta otra versión del programa:

```

// Escribir los valores de una matriz.
// Versión con punteros.
#include <iostream>
using namespace std;

int main()
{
    int lista[] = {24, 30, 15, 45, 34};
    int *plista = lista;
    for (int ind = 0; ind < 5; ind++)
        cout << *plista++ << " ";
    cout << endl;
}

```

La asignación *plista = lista* hace que *plista* apunte al comienzo de la matriz *lista*; es decir, al elemento primero de la matriz, y la expresión **plista++* da lugar a las dos operaciones siguientes en el orden descrito: **plista*, que es el valor apuntado por *plista*, y a *plista++*, que hace que *plista* pase a apuntar al siguiente elemento de la matriz. Esto es, el bucle se podría escribir también así:

```

for (int ind = 0; ind < 5; ind++)
{
    cout << *plista << " ";
    plista++;
}

```

Sin embargo, hay una diferencia entre el identificador de una matriz y un puntero. El identificador de una matriz es una constante y un puntero es una variable. Esto quiere decir que el siguiente bucle daría lugar a un error, porque *lista* es constante y no puede cambiar de valor.

```
for (int ind = 0; ind < 5; ind++)
    cout << *lista++ << " ";
```

En cambio, un parámetro de una función que sea una matriz se considera una variable (un puntero):

```
void VisualizarMatriz(int lista[], int n)
{
    int ind;
    for (ind = 0; ind < n; ind++)
        cout << *lista++ << " ";
}
```

Otro detalle a tener en cuenta es el resultado que devuelve el operador **sizeof** aplicado a una matriz o a un puntero que apunta a una matriz:

```
int lista[] = {24, 30, 15, 45, 34};
int *plista = lista;
cout << sizeof(lista) << " " << sizeof(plista) << endl;
```

El operador **sizeof** devuelve el tamaño en bytes de su operando. Por lo tanto, aplicado a la matriz *lista* devuelve el tamaño en bytes de la matriz, en el ejemplo 20, y aplicado al puntero *plista* que apunta a una matriz devuelve el tamaño en bytes del puntero, en el ejemplo 4.

Como aplicación de lo expuesto vamos a realizar una función *CopiarMatriz* que permita copiar una matriz de cualquier tipo de datos y de cualquier número de dimensiones en otra (será una función análoga a la función **memcpy** de la biblioteca de C). Dicha función tendrá el siguiente prototipo:

```
void CopiarMatriz(void *dest, void *orig, int nbytes);
```

El parámetro *dest* es un puntero genérico que almacenará la dirección de la matriz destino de los datos y *orig* es un puntero también genérico que almacenará la dirección de la matriz origen. El tamaño en bytes de ambas matrices es proporcionado a través del parámetro *nbytes*.

Según lo estudiado anteriormente en el apartado de *Punteros genéricos*, la utilización de éstos como parámetros permitirá pasar argumentos que sean matrices de cualquier tipo. Así mismo, según lo estudiado en este apartado, el nombre de

una matriz es siempre su dirección de comienzo, independientemente del tamaño y del número de dimensiones que tenga.

¿Qué tiene que hacer la función *CopiarMatriz*? Sencillamente copiar *nbytes* desde *orig* a *dest*. Piense que una matriz, sin pensar en el tipo de datos que contiene, no es más que un bloque de bytes consecutivos en memoria. Pensando así, *orig* es la dirección de comienzo del bloque de bytes a copiar y *dest* es la dirección del bloque donde se quieren copiar. La copia se realizará byte a byte, lo que garantiza independencia del tipo de datos (el tamaño de cualquier tipo de datos es múltiplo de un byte). Esto exige, según se explicó anteriormente en este mismo capítulo en el apartado *Importancia del tipo del objeto al que se apunta*, convertir *dest* y *orig* a punteros a **char**. Según lo expuesto, la solución puede ser así:

```
void CopiarMatriz( void *dest, void *orig, int nbytes )
{
    char *destino = static_cast<char *>(dest);
    char *origen = static_cast<char *>(orig);
    for (int i = 0; i < nbytes; i++)
    {
        destino[i] = origen[i];
    }
}
```

El siguiente programa utiliza la función *CopiarMatriz* para copiar una matriz *m1* de dos dimensiones de tipo **int** en otra matriz *m2* de iguales características.

```
// Copiar una matriz en otra.
#include <iostream>
using namespace std;

void CopiarMatriz( void *dest, void *orig, int nbytes );
int main()
{
    const int FILAS = 2;
    const int COLS = 3;
    int m1[FILAS][COLS] = {24, 30, 15, 45, 34, 7};
    int m2[FILAS][COLS];

    CopiarMatriz(m2, m1, sizeof(m1));

    for (int f = 0; f < FILAS; f++)
    {
        for (int c = 0; c < COLS; c++)
            cout << m2[f][c] << " ";
        cout << endl;
    }
}
```

```
void CopiarMatriz( void *dest, void *orig, int nbytes )
{
    char *destino = static_cast<char *>(dest);
    char *origen = static_cast<char *>(orig);
    for (int i = 0; i < nbytes; i++)
    {
        destino[i] = origen[i];
    }
}
```

Ejecución del programa:

```
24  30  15
45  34   7
```

Punteros a cadenas de caracteres

Puesto que una cadena de caracteres es una matriz de caracteres, es correcto pensar que la teoría expuesta anteriormente es perfectamente aplicable a cadenas de caracteres. Un puntero a una cadena de caracteres puede definirse de alguna de las dos formas siguientes:

```
char *cadena;
unsigned char *cadena;
```

¿Cómo se identifica el principio y el final de una cadena? La dirección de memoria donde comienza una cadena viene dada por el nombre de la matriz que la contiene y el final, por el carácter `\0` con el que C++ finaliza todas las cadenas. El siguiente ejemplo define e inicia la cadena de caracteres *nombre*.

```
char *nombre = "Francisco Javier";
cout << nombre << endl;
```



En el ejemplo anterior *nombre* es un puntero a una cadena de caracteres. El compilador C++ asigna la dirección de comienzo del literal “Francisco Javier” al puntero *nombre* y finaliza la cadena con el carácter `\0`. Por lo tanto, el operador `<<` sabe que la cadena de caracteres que tiene que visualizar empieza en la dirección *nombre* y que a partir de aquí, tiene que ir accediendo a posiciones sucesivas de memoria hasta encontrar el carácter `\0`.

Es importante tomar nota de que *nombre* no contiene una copia de la cadena asignada, sino la dirección de memoria del lugar donde la cadena está almacenada.

da, que coincide con la dirección del primer carácter; los demás caracteres, por definición de matriz, están almacenados consecutivamente. Según esto, en el ejemplo siguiente, *nombre* apunta inicialmente a la cadena de caracteres “Francisco Javier” y, a continuación, reasigna el puntero para que apunte a una nueva cadena. La cadena anterior se pierde porque el contenido de la variable *nombre* ha sido sobrescrito con una nueva dirección, la de la cadena “Carmen”.

```
char *nombre = "Francisco Javier";
cout << nombre << endl;
nombre = "Carmen";
```

Un literal, por tratarse de una constante de caracteres, no se puede modificar. Por ejemplo, si intenta ejecutar el código siguiente obtendrá un error:

```
char *nombre = "Francisco Javier";
nombre[9] = '-'; // error en ejecución
```

Lógicamente, el error comentado anteriormente no ocurre cuando la cadena de caracteres viene dada por una matriz que no haya sido declarada constante (**const**), según muestra el ejemplo siguiente:

```
char nombre[] = "Francisco Javier";
char *pnombre = nombre;
pnombre[9] = '-'; // se modifica el elemento de índice 9
```

El siguiente ejemplo presenta una función que copia una cadena en otra. Para ello, utiliza una función *copiacad* que tiene dos parámetros: la matriz destino y la matriz origen que contiene la cadena a copiar (recuerde que la biblioteca de C proporciona la función **strcpy** para realizar esta misma operación). Según esto, la llamada a la función podría ser así:

```
copiacad(cadena2, cadena1); // copia la cadena1 en la cadena2
```

Resulta evidente que la función *copiacad* tiene que recibir como parámetros las direcciones de las matrices destino y fuente. Por lo tanto, la solución del problema planteado puede ser así:

```
// Función para copiar una cadena en otra.
#include <iostream>
using namespace std;

void copiacad(char *, char *);

int main()
{
    char cadena1[81], cadena2[81];
```

```

    cout << "Introducir una cadena: ";
    cin.getline(cadena1, 81);
    copiacad(cadena2, cadena1);    // copia la cadena1 en la cadena2
    cout << "La cadena copiada es: " << cadena2 << endl;
}

void copiacad(char *dest, char *orig) // copia orig en dest
{
    *dest = *orig;
    while (*orig != '\0')
    {
        dest++;
        orig++;
        *dest = *orig;
    }
}

```

Ejecución del programa:

Introducir una cadena: hola
La cadena copiada es: hola

Aplicando lo estudiado en operaciones con punteros, podemos escribir una nueva versión de la función *copiacad* así:

```

void copiacad(char *dest, char *orig) // copia orig en dest
{
    while (*dest++ = *orig++);
}

```

MATRICES DE PUNTEROS

En capítulos anteriores, hemos trabajado con matrices multidimensionales, aunque en la práctica lo más habitual es utilizar matrices de punteros por las ventajas que esto reporta, como verá más adelante.

Se puede definir una matriz, para que sus elementos contengan, en lugar de un dato de un tipo primitivo, una dirección o puntero. Por ejemplo:

```

int *p[5];    // matriz de 5 elementos de tipo (int *)
int b = 30;   // variable de tipo int
p[0] = &b;    // p[0] apunta al entero b
cout << *p[0]; // escribe 30

```

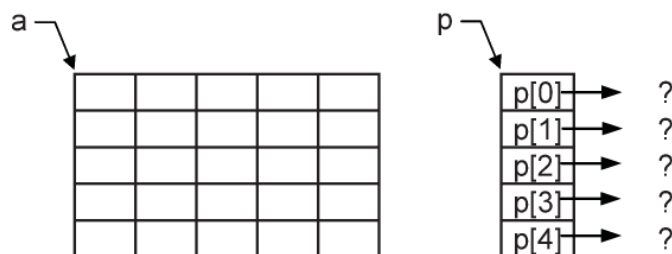
Este ejemplo define una matriz *p* de cinco elementos, cada uno de los cuales es un puntero a un **int**, y una variable entera *b*. A continuación asigna al elemento *p[0]* la dirección de *b* y escribe su contenido. Análogamente podríamos proceder

con el resto de los elementos de la matriz. Así mismo, si un elemento como $p[0]$ puede apuntar a un entero, también puede apuntar a una matriz de enteros; en este caso, el entero apuntado se corresponderá con el primer elemento de dicha matriz.

Según lo expuesto, una matriz de dos dimensiones y una matriz de punteros se pueden utilizar de forma parecida, pero no son lo mismo. Por ejemplo:

```
int a[5][5]; // matriz de dos dimensiones
int *p[5];   // matriz de punteros
```

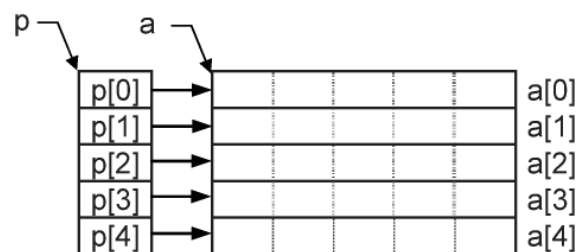
Las declaraciones anteriores dan lugar a que el compilador de C++ reserve memoria para una matriz a de 25 elementos de tipo entero y para una matriz p de cinco elementos declarados como punteros a objetos de tipo entero (**int ***).



Supongamos ahora que cada uno de los objetos apuntados por los elementos de la matriz p es a su vez una matriz de cinco elementos de tipo entero. Por ejemplo, hagamos que los objetos apuntados sean las filas de a :

```
int a[5][5]; // matriz de dos dimensiones
int *p[5];   // matriz de punteros a int

for (int i = 0; i < 5; i++)
    p[i] = a[i]; // asignar a p las filas de a
```



El bucle que asigna a los elementos de la matriz p las filas de a ($p[i] = a[i]$), ¿no podría sustituirse por una sola asignación $p = a$? No, porque los niveles de indirección son diferentes; dicho de otra forma, los tipos de p y a no son iguales ni admiten una conversión entre ellos. Veamos:

Tipo de p	<code>int * [5]</code>	(matriz de cinco elementos de tipo puntero a int)
Tipo de $p[i]$	<code>int *</code>	(puntero a int)

Tipo de a	<code>int (*)[5]</code>	(puntero a una matriz de cinco elementos de tipo int ; compatible con <code>int [][][5]</code> ; los elementos de a , $a[i]$, son matrices unidimensionales)
Tipo de $a[i]$	<code>int *</code>	(puntero a int ; compatible con <code>int []</code>)

A la vista del estudio de tipos anterior, la única asignación posible es la realizada: $p[i] = a[i]$.

El acceso a los elementos de la matriz p puede hacerse utilizando la notación de punteros o utilizando la indexación igual que lo haríamos con a . Por ejemplo, para asignar valores a los enteros referenciados por la matriz p y después visualizarlos, podríamos escribir el siguiente código:

```
for (int i = 0; i < 5; i++)
    for (int j = 0; j < 5; j++)
        cin >> p[i][j];

for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 5; j++)
        cout << setw(7) << p[i][j];
    cout << endl;
}
```

Según lo expuesto, ¿qué diferencias hay entre p y a ? En la matriz p el acceso a un elemento se efectúa mediante una indirección a través de un puntero y en la matriz a , mediante una multiplicación y una suma. Por otra parte, como veremos más adelante, las matrices apuntadas por p pueden ser de longitud diferente, mientras que en una matriz como a todas las filas tienen que ser de la misma longitud.

Supongamos ahora que necesitamos almacenar la dirección p , dirección de comienzo de la matriz de punteros, en otra variable q . ¿Cómo definiríamos esa variable q ? La respuesta la obtendrá si responde a esta otra pregunta: ¿a quién apunta p ? Evidentemente habrá respondido: al primer elemento de la matriz de punteros; esto es, a $p[0]$ que es un puntero a un **int**. Entonces p apunta a un puntero que a su vez apunta a un entero. Por lo tanto, q tiene que ser definida para que pueda almacenar un puntero a un puntero a un **int**.

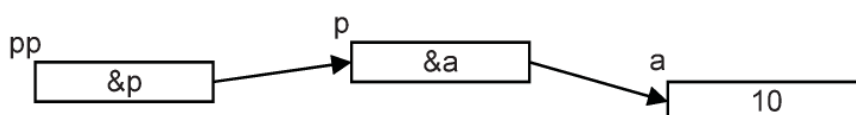
Punteros a punteros

Para especificar que una variable es un puntero a un puntero, la sintaxis utilizada es la siguiente:

```
tipo **varpp;
```

donde *tipo* especifica el tipo del objeto apuntado después de una doble indirección (puede ser cualquier tipo incluyendo tipos derivados) y *varpp* es el identificador de la variable puntero a puntero. Por ejemplo:

```
int a, *p, **pp;
a = 10; // dato
p = &a; // puntero que apunta al dato
pp = &p; // puntero que apunta al puntero que apunta al dato
```



Se dice que *p* es una variable con un nivel de indirección; esto es, a través de *p* no se accede directamente al dato, sino a la dirección que indica dónde está el dato. Haciendo un razonamiento similar diremos que *pp* es una variable con dos niveles de indirección.

El código siguiente resuelve el ejemplo anterior, pero utilizando ahora una variable *q* declarada como un puntero a un puntero. El acceso a los elementos de la matriz *a* utilizando el puntero a puntero *q* puede hacerse utilizando la indexación igual que lo haríamos con *a* o utilizando la notación de punteros. Utilizando la indexación sería así:

```
// Puntero a puntero.
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int i, j;
    int a[5][5]; // matriz de dos dimensiones
    int *p[5];   // matriz de punteros
    int **q;     // puntero a puntero a un entero

    for (i = 0; i < 5; i++)
        p[i] = a[i]; // asignar a p las filas de a
    q = p;

    for (i = 0; i < 5; i++)
        for (j = 0; j < 5; j++)
        {
            cout << "q[" << i << "][" << j << "]: ";
            cin >> q[i][j];
        }
}
```