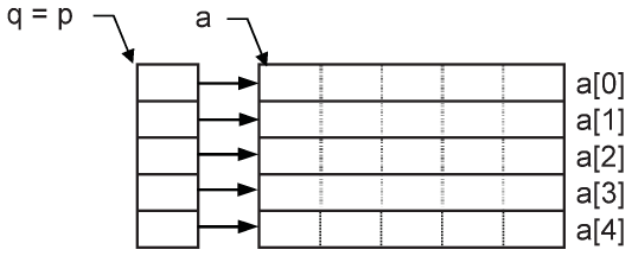
```
  for (i = 0; i < 5; i++)
  {
    for (j = 0; j < 5; j++)
      cout << setw(7) << q[i][j];
    cout << endl;
  }
}
```

Seguramente habrá pensado: ¿por qué no se asigna a *q* directamente *a* (*q=a*)? Pues porque *q* es una variable con dos niveles de indirección y *a* es una variable con un solo nivel de indirección. En otras palabras el tipo de *q* es **int **** y el tipo de *a* es **int (*)[5]**. En cambio, *p* si es una variable con dos niveles de indirección; su tipo es **int *[5]**, que significa matriz de cinco elementos de tipo **int ***; pero como el nombre de la matriz es un puntero a su primer elemento que es de tipo puntero a **int**, estamos en el caso de un puntero a un puntero.

A continuación modifiquemos el ejemplo anterior para realizar el acceso a los elementos de la matriz *p* utilizando la notación de punteros.




En la figura anterior podemos observar que la dirección de comienzo de la matriz es *q* o *p*. Entonces, si la dirección de comienzo de la matriz de punteros es *q*, suponiendo un valor entero *i* entre 0 y 4, ¿cuál es la dirección de su elemento *i*? Evidentemente *q+i*. Y, ¿cuál es el contenido de esta dirección? Esto es, ¿cuál es el valor de **(q+i)* o *q[i]*? Pues la dirección de la fila *a[i]* de la matriz *a*; esto es, *q[i]* y *a[i]* son la misma dirección, la de la fila *i* de la matriz *a*. Si a esta dirección le sumamos un entero *j* entre 0 y 4 (**(q+i)+j* o *q[i]+j*), ¿cuál es el resultado? Pues otra dirección: la que corresponde al elemento *j* de la fila *a[i]*. Y, ¿cuál es el contenido de esta dirección? Esto es, ¿cuál es el valor de **(*(q+i)+j)* o **(q[i]+j)* o *q[i][j]*? Pues el valor del elemento *a[i][j]* de la matriz *a*. De este análisis se deduce que las siguientes expresiones representan todas ellas el mismo valor:

```
q[i][j], *(q[i]+j), *(*(q+i)+j)
```

Según lo expuesto, observe que las direcciones *q+1* y **(q+1)* tienen significados diferentes. Por ejemplo:

*q+1+2* es la dirección del elemento *q[3]* de la matriz de punteros.

*(q+1)+2 es la dirección del elemento *q[1][2]*.
*(*(q+1)+2) es el valor del elemento *q[1][2]*.

De acuerdo con lo expuesto, la versión con punteros del ejemplo anterior presenta solamente la siguiente modificación:

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

for (i = 0; i < 5; i++)
{
  for (j = 0; j < 5; j++)
    cout << setw(7) << (*(*(q+i)+j));
  cout << endl;
}
```

## Matriz de punteros a cadenas de caracteres

Haciendo un estudio análogo al realizado para las matrices de punteros numéricas, diremos que una matriz de punteros a cadenas de caracteres es una matriz unidimensional en la que cada elemento es de tipo **char *** o **unsigned char ***. Por ejemplo:

```
char *p[5];           // matriz de cinco elementos de tipo (char *)
char c = 'z';         // variable c de tipo char
p[0] = &c;            // p[0] apunta al carácter 'z'
cout << *p[0] << endl; // escribe: z
```

Este ejemplo define una matriz *p* de cinco elementos, cada uno de los cuales es un puntero a un carácter (**char ***), y una variable *c* de tipo **char** iniciada con el valor 'z'. A continuación asigna al elemento *p[0]* la dirección de *c* 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 carácter, también puede apuntar a una cadena de caracteres o matriz unidimensional de caracteres; en este caso, el carácter apuntado se corresponderá con el primer elemento de la cadena. Por ejemplo:

```
char *p[5];           // matriz de 5 elementos de tipo (char *)
p[0] = "hola";        // p[0] apunta a la cadena hola
cout << p[0] << endl; // escribe: hola
```

Una asignación de la forma *p[0] = "hola"* asigna la dirección de la cadena especificada al elemento de la matriz indicado, que tiene que ser de tipo **char ***. Por otra parte, sería un error escribir una sentencia como:

```
p[1] = 'a';
```

porque se está intentando asignar un valor **int** (valor ASCII del carácter *a*) a una variable de tipo **char ***, y a un puntero sólo se le puede asignar otro puntero.

Quizás también, haya pensado en escribir el siguiente código para leer todas las cadenas de caracteres:

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

int main()
{
  char *p[5]; // matriz de punteros
  for (int i = 0; i < 5; i++)
    cin >> p[i];
}
```

Escribir el código anterior es un error, porque el operador >> lee una cadena de caracteres de la entrada estándar y la almacena en la cadena de caracteres especificada; en nuestro caso la tiene que colocar a partir de la dirección especificada por *p[i]*, pero, ¿qué dirección es ésta si la matriz no ha sido iniciada? La solución pasa por almacenar en *p[i]* la dirección de un bloque de memoria que pueda ser utilizado por el programa. Por ejemplo, se podría seguir un razonamiento análogo al realizado anteriormente en el apartado *Punteros a punteros*, pero hay soluciones mejores, como trabajar con matrices de objetos **string** (este tema fue expuesto en el capítulo anterior).

Sí sería posible declarar una matriz de punteros a **char** e iniciarla con cadenas de caracteres. Por ejemplo, el siguiente programa muestra una función que recibe como parámetro el entero correspondiente a un mes y nos devuelve como resultado un puntero a la cadena de caracteres que nombra a dicho mes.

```
// Función que devuelve el nombre del mes 1 a 12 dado
#include <iostream>
using namespace std;

char *nombre_mes(unsigned int mm)
{
  // mes es una matriz de punteros a cadenas de caracteres
  static char *mes[] = { "Mes no correcto",
                         "enero", "febrero", "marzo",
                         "abril", "mayo", "junio", "julio",
```

```
                          "agosto", "septiembre", "octubre",
                          "noviembre", "diciembre"                };
  return ((mm > 0 && mm <= 12) ? mes[mm] : mes[0]);
}

int main()
{
  char *m = nombre_mes(8);
  cout << "\nMes: " << m << endl; // escribe Mes: agosto
}
```

En este ejemplo, *mes* es una matriz de 13 elementos (0 a 12) que son punteros a cadenas de caracteres. Cada elemento de la matriz ha sido iniciado con un literal. Como se ve, éstos son de diferente longitud y todos serán finalizados automáticamente por C++ con el carácter nulo. Si en lugar de utilizar una matriz de punteros, hubiéramos utilizado una matriz de dos dimensiones, el número de columnas tendría que ser el del literal más largo, más uno para el carácter nulo, con lo que la ocupación de memoria sería mayor.

Siguiendo con el ejemplo, es fácil comprobar que las declaraciones **char** **mes[]* y **char** ***mes* no son equivalentes. La primera declara una matriz de punteros a cadenas de caracteres y la segunda un puntero a un puntero a una cadena de caracteres. Por lo tanto, sería un error sustituir en el código **mes[]* por ***mes*.

## ASIGNACIÓN DINÁMICA DE MEMORIA

C++ cuenta fundamentalmente con dos métodos para almacenar información en la memoria. El primero utiliza variables globales y locales. En el caso de variables globales, el espacio es fijado para ser utilizado a lo largo de toda la ejecución del programa; y en el caso de variables locales, la asignación se hace a través de la pila del sistema; en este caso, el espacio es fijado temporalmente, mientras la variable existe. El segundo método utiliza los operadores **new** y **delete** de C++. Como es lógico, estos operadores utilizan el área de memoria libre para realizar las asignaciones de memoria solicitadas.

La *asignación dinámica de memoria* consiste en asignar la cantidad de memoria necesaria para almacenar un objeto durante la ejecución del programa, en vez de hacerlo en el momento de la compilación del mismo. Cuando se asigna memoria para un objeto de un tipo cualquiera, se devuelve un puntero a la zona de memoria asignada. Según esto, lo que tiene que hacer el compilador es asignar una cantidad fija de memoria para almacenar la dirección del objeto asignado dinámicamente, en vez de hacer una asignación para el objeto en sí. Esto implica declarar un puntero a un tipo de datos igual al tipo del objeto que se quiere asignar dinámicamente. Por ejemplo, si queremos asignar memoria dinámicamente para