Tipos de datos I. Vectores y matrices

Los arrays o matrices surgen de la necesidad de tratar conjuntos de datos del mismo tipo relacionados entre sí, de forma que mediante un nombre común y una variable o valor indicador podamos referirnos bien a todos ellos o a uno en particular. Constituyen una de las estructuras de datos más importantes y serán objeto de estudio en este tema.Vamos a ver cómo utilizar estas estructuras y cómo pueden ser instrumentadas. Estas estructuras son tipos compuestos o estructurados; es decir, están formadas por los tipos de datos simples existentes en los diversos lenguajes.

Concepto de vector (array)

En este tema tratamos los arrays estáticos. Un *array*[1] (vector) es una estructura homogénea de datos, de tamaño constante en la que se accede a cada uno de sus elementos mediante un identificador común y uno o varios índices. Según la definición podemos ver:

- Todos los elementos del array son del mismo tipo.

- El número de ellos no varía durante la ejecución del programa.

- Accedemos a un elemento de la estructura mediante un identificador: el nombre del array, y el valor que toman uno o varios índices. Al número de índices necesarios para designar un elemento del array se le denomina dimensión del array.

- El número máximo de valores posibles que puede tomar cada índice se denomina rango de esa dimensión o índice. Los valores han de ser consecutivos, por lo que el índice ha de ser de un tipo ordinal.

Podemos concluir, por tanto, que un array es una estructura de datos homogénea, estática y de acceso directo a sus componentes por posición. La posición viene determinada por los valores de uno o varios índices. El tamaño ocupado por cada uno de sus componentes sería el mismo que el que ocuparía si se declarara como variable independiente.

Por ejemplo, un array puede contener todo enteros, o todo caracteres, pero no puede contener ambos.

Una línea de un texto puede ser representada como un array de caracteres; un vector puede ser representado como un array de números reales; y como una matriz consta de columnas, cada una de las cuales es un vector, una matriz puede ser representada como un array de vectores.

Ejercicio sin arrays:

[1] El array también es conocido como tabla, matriz, vector, arreglo, disposición, etc.

- ✓ Un grupo de diez amigos quiere organizar una fiesta y necesita almacenar la cantidad de dinero que tiene cada uno en el bolsillo, así como la cantidad que tienen en total.

Una posible solución sería:

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

int main() {
  float amigo1, amigo2, amigo3, amigo4, amigo5, amigo6, amigo7,
amigo8, amigo9, amigo10, total;

  cout << "Cuánto dinero tiene amigo1?: ";
   cin >> amigo1;
  cout << "Cuánto dinero tiene amigo2?: ";
   cin >> amigo2;
  cout << "Cuánto dinero tiene amigo3?: ";
   cin >> amigo3;
  cout << "Cuánto dinero tiene amigo4?: ";
   cin >> amigo4;
  cout << "Cuánto dinero tiene amigo5?: ";
   cin >> amigo5;
  cout << "Cuánto dinero tiene amigo6?: ";
   cin >> amigo6;
  cout << "Cuánto dinero tiene amigo7?: ";
   cin >> amigo7;
  cout << "Cuánto dinero tiene amigo8?: ";
   cin >> amigo8;
  cout << "Cuánto dinero tiene amigo9?: ";
   cin >> amigo9;
  cout << "Cuánto dinero tiene amigo10?: ";
   cin >> amigo10;

  total = amigo1 + amigo2 + amigo3 + amigo4 + amigo5 + amigo6 +
amigo7 + amigo8 + amigo9 + amigo10;
  cout << "En total disponemos de " << total << " euros." << endl;
}
```

Imagina qué ocurriría si el número de amigo se incrementa. En ese caso es mejor reducir las amistades o encontrar otra solución para almacenar tantos datos.

arrays en C y en C++

El vector, en términos de un lenguaje de programación, se corresponde con un array unidimensional. En C se define de la siguiente forma:

```
tipoBase   nombreArray[expresión];
```

tamaño el array

Donde *expresión* es de tipo entero y positiva, e indica el tamaño del array. Se trata pues del número de elementos que se pueden almacenar en el vector, a los que se accede mediante el índice expresado entre corchetes. Los *índices*, que sirven para identificar unívocamente cada elemento del array, son por tanto enteros no negativos. Una vez declarado un vector de *n* elementos, se puede leer y escribir en sus componentes teniendo en cuenta que éstos se numeran del *0* al *n-1*.

Un array multidimensional se define de forma análoga, separando cada uno de los índices por un par de corchetes:

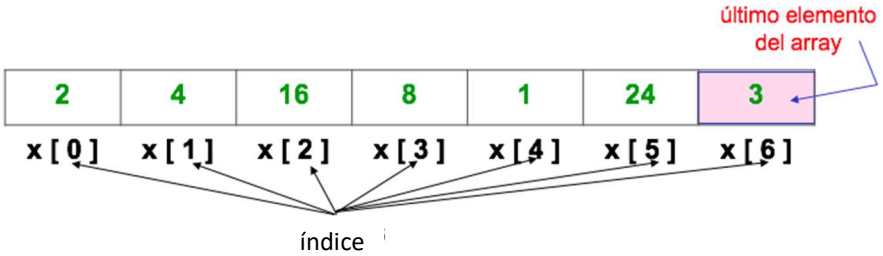
```
tipoBase nombreArray[expresion1][expresion2]...[expresionN];
```

Ejemplo:

Una posible declaración de un vector de 7 elementos de tipo entero en C/C++ sería:

```
int x[7];
```

La definición anterior corresponde a un array de 7 variables de tipo entero, o lo que es lo mismo, disponemos de espacio en memoria central para almacenar 7 valores enteros. En el caso anterior, para referenciar el primer elemento del vector hemos de escribir `x[0]` y para referenciar el último elemento hemos de escribir `x[6]`.



Si se desea declarar un array bidimensional:

```
int lista[10][25];
```

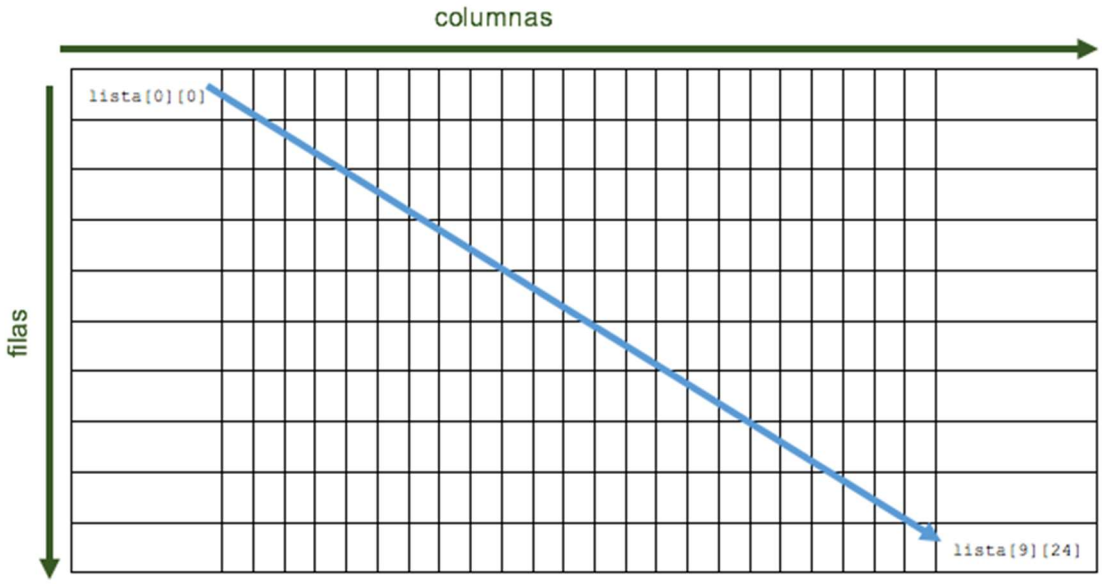
declara un array de 10 por 25 elementos de tipo entero.

Elemento 1,1	Elemento 1,2	Elemento 1,3	...	Elemento 1,25
Elemento 2,1	Elemento 2,2	Elemento 2,3		Elemento 2,25
Elemento 3,1	Elemento 3,2	Elemento 3,3		Elemento 3,25
...				
Elemento 10,1	Elemento 10,2	Elemento 10,3		Elemento 10,25

El acceso a cada elemento del vector bidimensional sería:

`lista[0][0]`	`lista[0][1]`	`lista[0][2]`	`...`	`lista[0][24]`
`lista[1][0]`	`lista[1][1]`	`lista[1][2]`		`lista[1][24]`
`lista[2][0]`	`lista[2][1]`	`lista[2][2]`		`lista[2][24]`
`...`				
`lista[9][0]`	`lista[9][1]`	`lista[9][2]`		`lista[9][24]`

Hay que dejar claro que si tiene 10 filas, se accede desde el elemento que se encuentra en la posición 0 hasta el elemento que se encuentra en la posición 9, y lo mismo con las columnas, si tiene 25 columnas irá por tanto desde la columna 0 hasta la columna 24.



Con los elementos que componen la estructura podemos realizar las mismas operaciones que con el tipo base al que pertenecen.

Acceso a los elementos de un array en C/C++. Declaración e inicialización

Veamos como se representa una matriz (hablando en términos matemáticos). Necesitaremos entonces un array bidimensional, donde el primer índice corresponde a las filas de la matriz y el segundo a las columnas. El siguiente ejemplo muestra la definición de una matriz de 25x25 elementos, de tipo entero, a los que podremos referirnos con el identificador *matriz* y los índices correspondientes.

```
int matriz[25][25];
```

Al igual que a una variable simple, podemos asignarle unos valores iniciales, independientemente de que durante el proceso del programa se modifiquen eventualmente. Sea por ejemplo:

```
int vector1[]= {1,3,15,18,-2,15,44,12,1,4};
int vector2[10] = {1,3,15,18,-2,15,44,12,1,4};
int vector3[10] = {1,3,15};
int vector4[10];
```

En las definiciones anteriores se puede apreciar que si no se indica el número de elementos (ejemplo de la variable vector1) se toma como tal el número de valores asignados. También es posible inicializar parcialmente el array (ejemplo de variable vector3), pero siempre se asignan los valores a los primeros elementos del array, desde la posición cero en adelante. Por tanto, el primer elemento del array *vector3* tendrá valor *1*, el segundo *3*, etc. En esta inicialización, sólo se asignan valores a los tres primeros elementos del array, quedando el resto a 0 por defecto. Notemos que seguimos teniendo 10 variables agrupadas bajo el identificador *vector3*, aunque únicamente le hemos dado valores iniciales a 3 de ellas. En el caso de la variable vector4 hemos reservado 10 posiciones de tipo entero pero sin asignar valores. Al no asignar valores desconocemos el contenido de esas celdas, dependerá de lo que haya en esa posición de memoria en ese momento.

La inicialización tiene otra función bastante interesante, y es la posibilidad de definir el tamaño del array por el número de valores de inicialización:

```
int vector1[]={1,3,15,18,-2,15,44,12,1,4};
```

Entre los corchetes no hemos indicado ningún tamaño, pero el compilador sabe, por la cadena de inicialización, qué cantidad de memoria debe reservar para el array, en este caso, diez enteros. Realmente, la utilidad de esta

característica del lenguaje está más justificada en el caso de **arrays** de caracteres o **string's**, que veremos en los siguientes apartados del tema.

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

int main(){

  int vector1[] = {1,3,15,18,-2,15,44,12,1,4};
  int vector2[10] = {1,3,15,18,-2,15,44,12,1,4};
  int vector3[10] = {1,3,15};
  int vector4[10];
  int i;

  cout << "vector1: ";
  for (i=0; i<10; i++)
    cout << vector1[i] << " " ;
  cout << endl;

  cout << "vector2: ";
  for (i=0; i<10; i++)
    cout << vector2[i] << " " ;
  cout << endl;

  cout << "vector3: ";
  for (i=0; i<10; i++)
    cout << vector3[i] << " " ;
  cout << endl;

  cout << "vector4: ";
  for (i=0; i<10; i++)
    cout << vector4[i] << " " ;
  cout << endl;
}
```

Una ejecución daría como resultado la siguiente salida por pantalla:

```
vector1: 1 3 15 18 -2 15 44 12 1 4
vector2: 1 3 15 18 -2 15 44 12 1 4
vector3: 1 3 15 0 0 0 0 0 0 0
vector4: 0 0 0 0 0 0 0 0 -717304506 -1174390318
```

En el caso del vector4, podríamos obtener distintos valores en cada ejecución, puesto que no ha sido inicializado. En cada ejecución se mostrarán los valores existentes en esas celdas de memoria en ese momento concreto.

No debemos olvidar que al declarar un vector entre corchetes indicamos el tamaño de este y puesto que el índice del primer elemento de un array en C es siempre cero (0), los elementos se encontrarán entre la posición 0 y el tamaño del vector menos 1 (n-1).

La sintaxis sería la misma para el caso de arrays multidimensionales:

```
int matriz[3][2] = {1,2,10,20,100,200};
int matriz[3][2] = {  1,  2,
                     10, 20,
                    100,200};
```

Ambas formas de redacción son válidas y ejecutan la misma acción. El compilador gestiona la memoria de manera que podemos ver un array como una serie de posiciones de memoria adyacentes y en línea, de manera que la distribución de las variables se correspondería a las coordenadas *00, 01, 10, 11, 20, 21*. Esto quiere decir que los índices más a la derecha son los que "corren" más rápidos. En este caso, se van asignando valores primero por filas y, una vez agotados los elementos de esa fila se pasaría a la siguiente. El resultado de la anterior inicialización sería pues:

$matriz_{00}$ = 1	$matriz_{01}$ = 2
$matriz_{10}$ = 10	$matriz_{11}$ = 20
$matriz_{20}$ = 100	$matriz_{21}$ = 200

Es una buena práctica utilizar constantes simbólicas para la definición del tamaño de los arrays, ya que modificaciones de su tamaño sólo precisarían de un cambio en el valor de la constante.

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

const int kcuantos = 10;
const int knotas = 6;

int main(){
   float notas[knotas];
   int matrizA[kcuantos][kcuantos];
   int matrizB[20][kcuantos];

   // resto del programa
   ...
}
```

Acabamos de ver la definición e inicialización. Aparte, puede ser necesario declarar el array, típicamente en la definición de los parámetros formales de una función, o en aquellas circunstancias en que se va a utilizar el array antes de definirlo. La sintaxis del C/C++ nos dice que, de todos los grupos de corchetes que indican el tamaño de cada dimensión, el primer grupo ha de estar vacío, es decir, la primera dimensión. Por ejemplo, si seguimos con el ejemplo de *notas* y *matriz* anteriores, al pasarlos como parámetros pondríamos:

```
int notas[];              // descripción utilizada en la
int matriz[][kcuantos];   // declaración de parámetros formales
```

- ✓ Resuelve el ejercicio de los amigos de forma más eficiente.

```
Otra posible solución utilizando arrays sería:
#include <iostream>
using namespace std;

const int kamigos = 10;

int main() {
   float amigos[kamigos], total;
   int i;

  // almaceno el dinero de cada amigo
  for (i=0; i < kamigos; i++) {
        cout << "¿Cuánto dinero tiene amigo" << i+1 << "? ";
        cin >> amigos[i];
```

```
        }

        // calculo cuánto dinero tenemos entre todos
        total = 0;
        for (i=0; i < kamigos; i++)
              total = total + amigos[i];

        cout << "En total disponemos de " << total << " euros" << endl;
    }
```

- ✓ En el anterior ejercicio, ¿por qué utilizo la constante `kamigos`?

  El código está más claro.
  Podría poner directamente el 10, pero al hacerlo así si cambiamos el total de amigos, sólo con cambiar el valor que asigno a la constante, ya lo tendría todo actualizado.

- ✓ En el mismo ejercicio, ¿por qué voy desde "`0`" hasta "`<kamigos`" y no hasta "`<=kamigos`"?

  El vector se ha declarado con tamaño 10, eso quiere decir que contiene 10 elementos, que según la sintaxis de los vectores, van desde la posición 0 hasta la 9. Si pusiésemos "<=", iría hasta la posición 10 que no existe.

- ✓ Comparando este ejercicio que utiliza vectores con el que no las utiliza, vemos que el número de líneas es similar, ¿por qué lo utilizo?

  En este caso puede ser similar pero si incrementamos el número de amigos, es una barbaridad.
  Por otro lado, el código está más claro.

Manejo de arrays en módulos

El nombre del array puede ser usado como argumento de una función, permitiendo así que el array completo sea pasado a la misma. Para pasar un array a una función, el nombre del mismo debe aparecer sólo, sin corchetes ni índices, como argumento actual dentro de la llamada a la función.

El correspondiente parámetro formal debe constar en la línea de definición de la función, en la declaración de parámetros, como un array de *tipoBase* adecuado. Cuando el array es unidimensional, no se especifica el tamaño del mismo (corchetes vacíos), sin embargo si el array es multidimensional se debe especificar el tamaño de todos los índices, excepto del primero, que tendrá corchetes vacíos, como se vio en el apartado de acceso a los elementos del array).

La forma de pasar un array a una función difiere mucho de la de una variable simple, pues al pasar un array no se hace una copia para la función de los valores de los elementos del array. En vez de esto, lo que se hace es pasar a la función la posición del array, puesto que su identificador representa la dirección en memoria central donde está almacenado el primer elemento del mismo. Con esto, C/C++ evita el copiar (como hace con cualquier variable simple pasado por valor) la estructura completa del array, lo que implica un coste en cantidad de memoria y tiempo de copia.

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

const int kCUANTOS = 6;

// prototipos de las funciones
void leeVector(int []);              // leer
void escVector(int []);              // escribir
void copyVecs(int [], int []);       // copiar
int compVecs(int [], int []);        // comparar
```

```
void leeVector(int v[]) {
  int i;

  for(i=0; i<kCUANTOS; i++)
      cin >> v[i];
}

void escVector(int v[]) {
  int i;

  for(i=0; i<kCUANTOS; i++)
     cout << v[i];
}

void copyVecs(int origen[],int destino[]) {
  int i;

  for(i=0; i<kCUANTOS; i++)
     destino[i]=origen[i];
}

int compVecs(int v[],int w[]) {
  int i;

  for(i=0; i<kCUANTOS && v[i]==w[i]; i++);
  return(i==kCUANTOS); // devolverá FALSO o CIERTO
                       // CIERTO: los arrays contienen la misma información
}

int main() {
    int vecA[kCUANTOS], vecB[kCUANTOS];

    leeVector(vecA);
    copyVecs(vecA,vecB);
    if (compVecs(vecA,vecB))
        cout << "Los vectores son idénticos" << endl;
    else
        cout << "Los vectores no son iguales" << endl;
    escVector(vecB);
    cout << endl;
    return 0;
}
```

La ejecución sería:

```
1 2 3 4 5 6
Los vectores son idénticos
123456
```

Así pues, **los arrays siempre son pasados por referencia** en C/C++. Cualquier modificación sobre ellos será permanente. La forma de acceder a un determinado elemento del vector, dado que como hemos dicho el identificador representa la dirección de memoria, se realiza calculando el desplazamiento necesario para llegar al índice correspondiente.

Por tanto, si los elementos de un array deben ser preservados debemos copiar éste, elemento a elemento, sobre otro array de las mismas características.

En el ejemplo anterior, en la definición de los parámetros formales de las funciones los corchetes se muestran vacíos pues la definición de los arrays ya contiene el tamaño de cada uno. Se han definido dos arrays de enteros. Ambos definen su tamaño en función de la constante simbólica *kCUANTOS*, con lo que si en un momento dado necesitáramos cambiar este valor, únicamente necesitaríamos hacerlo una vez en la línea de la declaración de constantes.

Al inicio del programa se han escrito los prototipos de las funciones que se van a definir a continuación. Como no es necesario especificar ningún identificador, se declaran el tipo de elementos que va a almacenar el vector y se indica que se trata de un array por los dos corchetes, sin tamaño.

Utilizar arrays como parámetros para las funciones implica el paso de los mismos por referencia, por lo que cualquier alteración de los valores que a ellos pertenecen se mantendrá cuando finalice la ejecución de las mismas. Asimismo, aun siendo globales los arrays definidos, para generalizar las funciones y que puedan ser utilizadas con cualquier vector, se definen en función de arrays como argumentos formales.

El programa lee un vector y lo copia seguidamente. La comparación será, por tanto, positiva. Finalmente muestra en pantalla la copia del vector original.

- ✓ Disponemos de un vector con 5 elementos y calculamos su suma. A continuación te mostramos el código que lo resuelve, pero no es un buen diseño, ¿cómo lo cambiarías?

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

int main() {
  int numero[5]; /* Un array de 5 números enteros */
  int suma; /* Un entero que será la suma */

  numero[0] = 200; /* Les damos valores */
  numero[1] = 150;
  numero[2] = 100;
  numero[3] = -50;
  numero[4] = 300;
  suma = numero[0] + numero[1] + numero[2] + numero[3] + numero[4];
  cout << "Su suma es " << suma << endl;
}

La solución más adecuada sería:
#include <iostream>
using namespace std;

const int ktotal = 5;

int main() {
  int numero[ktotal]={200, 150, 100, -50, 300};
  int suma, i;

  suma = 0;
  for (i=0; i < ktotal ; i++)
    suma = suma + numero[i];

  cout << "Su suma es " << suma << endl;
}
```

- ✓ Realiza un programa que almacene en una tabla el número de días que tiene cada mes (supondremos que es un año no bisiesto), pida al usuario que indique un mes en numérico (1=enero, 2= febrero, ...) y muestre en pantalla el número de días que tiene ese mes.

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

```
const int ktotal = 5;

int main() {
  int meses[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  int mes;

  cout << "Los días del mes? ";
  cin >> mes;

  cout << "El mes " << mes << " tiene " << meses[mes-1] << " días"
<< endl;
}
```

Arrays especiales: Las cadenas de caracteres

Es frecuente emplear vectores para la manipulación de cadenas de caracteres, como nombres de personas, de calles, etc. En estos casos, hemos de declarar la variable con una longitud capaz de alojar el nombre más largo de entre los valores posibles. Así, para almacenar nombres de personas, podemos declarar el vector `char nombre[15]` si sabemos que 15 es una cota para el tamaño de los nombres que aparecerán durante la ejecución. Ahora bien, resulta muy incómodo tener que rellenar con espacios el resto de las letras hasta la *15ª* cada vez que leemos un nombre desde el teclado.

Para facilitar el tratamiento de estos vectores de caracteres con longitud variable, C tiene un tipo predefinido que resulta muy conveniente: el tipo **string** (cadena de caracteres). Su declaración es la normal de un vector de caracteres. Lo que cambia es su lectura y escritura. En el momento de leer una cadena de caracteres, el computador los leerá uno a uno hasta encontrar un cambio de línea. Cuando esto suceda, añadirá al final de la secuencia leída un carácter nulo, NULL, representado mediante la secuencia de escape '\0' (el carácter con código ASCII igual a 0), consiste en una secuencia de escape que se utiliza para indicar el fin de la cadena y no aparece cuando se muestra en pantalla. Esta marca le permitirá saber posteriormente dónde se encuentra el fin de la cadena. Si sabemos que la longitud máxima de las cadenas a tratar es *k*, hace falta declarar el vector de dimensión *k+1* para poder alojar también esta marca. Así, en el ejemplo anterior declararíamos

```
char nombre[16];  // permite almacenar 15 caracteres + '\0'
```

Una constante de este tipo se expresa poniendo la cadena de caracteres entre comillas ("). Nótese la diferencia entre una constante de tipo carácter ('a') y su correspondiente constante de cadena de caracteres ("a"), pues la segunda constante está formada por los char 'a' y '\0', no siendo, por tanto, equivalentes ambas.

Definición e inicialización

Normalmente, cuando se le asigna una cadena constante a un array como parte de la definición, se omite el tamaño del array. El tamaño adecuado será asignado automáticamente, incluyendo la previsión del carácter nulo '\0', que se añade, también automáticamente, al final de la cadena.

```
char cad[] = "hola";
```

La definición anterior corresponde a una cadena de caracteres donde el tamaño y el carácter nulo los asigna directamente el compilador. Es equivalente a realizar las siguientes definiciones:

```
char cad[5] = "hola";
char cad[5] = {'h','o','l','a','\0'};
```

Otro ejemplo de inicialización de cadenas de caracteres, en este caso un array de cadenas, conteniendo cada uno de sus elementos una cadena que representa el nombre del mes del año. Es un array de caracteres de dos dimensiones pero es más habitual referirse a esta estructura como un array de cadenas.

```
char meses[13][11] ={" ", "ENERO", "FEBRERO", "MARZO",
                     "ABRIL", "MAYO", "JUNIO", "JULIO",
                     "AGOSTO", "SEPTIEMBRE","0CTUBRE",
                     "NOVIEMBRE", "DICIEMBRE"};
```

La primera dimensión se refiere al número del mes dentro del año. Así, si nos referimos a `meses[2]` estaremos trabajando con la cadena `"FEBRERO"`. Nótese que `meses[0]` contiene una cadena vacía (`" "`) para adecuar el índice a la numeración real de cada mes. La segunda dimensión sería la longitud máxima de las cadenas (`"SEPTIEMBRE"`), teniendo en cuenta el carácter de fin de cadena, `'\0'`.

Una sentencia del tipo

```
cout << "El mes es " << meses[2] << endl;
```

Mostraría por pantalla:

```
El mes es FEBRERO
```

Mientras que

```
cout << "Tenemos " << meses[2][2] << endl;
```

mostraría por pantalla:

```
B
```

Puesto que estamos señalando a un elemento dentro de la tercera cadena del array de cadenas.

Operaciones con cadenas

C/C++ posee también una gran variedad de funciones para manipular cadenas. Las más corrientes son

```
strcpy(cad_1,cad_2);  // copia el contenido de cad_2 sobre cad_1
strcat(cad_1,cad_2);  // añadir el contenido de cad_2 al final de cad_1
resultado_comparacion = strcmp(cad_1,cad_2); // compara cad_1 y cad_2
longitud = strlen(cad);  // retorna la longitud de cad.
```

Veamos esto con un ejemplo que iremos presentando poco a poco para su comprensión:

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

const int kmax = 10;

int main(){
  char cad1[kmax], cad2[kmax];
  int longitud, resultado_comparacion;

  cout << "Introduce una palabra: ";
  cin >> cad1;
  // Mostramos el contenido de cada cadena
  cout << "cad1 contiene " << cad1 << endl;
  cout << "cad2 contiene " << cad2 << endl;
```

Tras la ejecución de este fragmento, tenemos la siguiente salida por pantalla:

```
Introduce una palabra: hola
cad1 contiene hola
cad2 contiene
```

Añadimos el siguiente código en el que calculamos la longitud de la cadena:

```
   // Mostramos la longitud de cad1
   longitud = strlen(cad1);
   cout << "cad1 contiene " << longitud << " caracteres" << endl;
```

Que produce la salida por pantalla:

```
cad1 contiene 4 caracteres
```

Tras añadir el siguiente código para copiar una cadena en otra:

```
   strcpy(cad2,cad1);
   // Mostramos el contenido de cada cadena
   cout << "cad1 contiene " << cad1 << endl;
   cout << "cad2 contiene " << cad2 << endl;
```

Obtenemos la siguiente salida:

```
cad1 contiene hola
cad2 contiene hola
```

Para concatenar ambas cadenas:

```
   strcat(cad1,cad2);
   // Mostramos el contenido de cada cadena
   cout << "cad1 contiene " << cad1 << endl;
   cout << "cad2 contiene " << cad2 << endl;
```

Obtenemos la siguiente salida:

```
cad1 contiene holahola
cad2 contiene hola
```

Añadimos una parte de código en la que solicitamos de nuevo una palabra y la guardamos en cad1:

```
   cout << "Introduce una palabra: ";
   cin >> cad1;
   // Mostramos el contenido de cada cadena
   cout << "cad1 contiene " << cad1 << endl;
   cout << "cad2 contiene " << cad2 << endl;
   // comparamos el contenido de cad1 con con el contenido de cad2
   resultado_comparacion = strcmp(cad1,cad2);
   // mostrará 0 si son iguales,
   // positivo si cad1 es mayor que cad2
   // y negativo si cad2 es mayor que cad1
   cout << "Las cadenas son : " << resultado_comparacion << endl;
   return 0;
}
```

Veamos los diferentes resultados en función de la nueva cadena introducida:

Cuando cad1 > cad2:

```
Introduce una palabra: pato
cad1 contiene pato
cad2 contiene hola
Las cadenas son : 8
```

Cuando cad1 < cad2:

```
Introduce una palabra: hola
cad1 contiene hola
cad2 contiene pato
Las cadenas son : -8
```

Cuando ambas cadenas son iguales:

```
Introduce una palabra: hola
cad1 contiene hola
cad2 contiene hola
Las cadenas son : 0
```

Vamos a escribir algunas funciones en C que realizan tareas equivalentes a algunas operaciones primitivas sobre cadenas de caracteres. Para todas estas funciones, suponemos las siguientes declaraciones:

```
const int kmax = 80;
 ...
int main{
   char cadena[kmax];
   ....
}
```

La primera función encuentra la longitud actual de una cadena.

```
 int longitud (char cadena[]) {
     int i;

     for (i=0; cadena[i] != '\0'; i++);
     return(i);
 }
```

La segunda función recibe dos cadenas como parámetros. La función retorna un entero que indica la posición inicial de la primera ocurrencia de la segunda cadena de caracteres dentro de la primera cadena. Si la segunda cadena no existe dentro de la primera, se retorna *-1*.

```
int strposicion (char s1[], char s2[]) {

  int i, j1, j2, res;

  res = -1;
  len1 = longitud(s1);
  len2 = longitud(s2);

  for (i=0; i+len2 <= len1 ; i++)
    for (j1=i, j2=0; j2<len2 && s1[j1]==s2[j2]; j1++, j2++){
      if (j2 == len2-1)
```

```
            res = i;
        }

    return(res);
}
```

Otra operación frecuente con cadenas es la concatenación. El resultado de concatenar dos cadenas consiste en obtener en una cadena los caracteres de la primera seguidos por los caracteres de la segunda. La función siguiente establece *s1* para la concatenación de *s1* y *s2*.

```
void strconcat(char s1[], char s2[]) {
    int i, j;

    for (i=0; s1[i]!='\0'; i++);
    for (j=0; s2[j]!='\0'; j++, i++)
       s1[i]=s2[j];
    s1[i]='\0';   // añado la marca de final de cadena
}
```

Otra operación habitual es la operación de subcadenas, `subcadena(s1,i,j,s2)` establece la cadena `s2` para los caracteres que empiezan en la posición `i` hasta la posición `j` de `s1`.

```
void subcadena (char s1[], int i, int j, char s2[]) {
    int k, m;

    for (k=i, m=0; m<j; m++, k++)
       s2[m]=s1[k];
    s2[m]='\0';  // añadimos el final de cadena
}
```

- ✓ Realiza un programa que pida por teclado tu nombre, lo guarde y a continuación lo muestre

  Una solución podría ser:

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

const int ktam = 20;

int main() {
  char nombre[ktam];

  cout << "Dime tu nombre: ";
  cin >> nombre;

  cout << "Tu nombre es "<< nombre << endl;
}
```

- ✓ ¿Has probado utilizando el código anterior a introducir un nombre compuesto? ¿Qué ocurre?

  Que sólo almacena el primer nombre.
  Esto es debido a que la instrucción "cin" lee hasta llegar a un espacio o a un intro.

- ✓ ¿Cómo lo podríamos resolver?

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

const int ktam = 20;

int main() {
  char nombre[ktam];

  cout << "Dime tu nombre: ";
  cin.getline(nombre, ktam);

  cout << "Tu nombre es "<< nombre << endl;
}
```

- ✓ ¿Puedes introducr un nombre de 20 letras?

```
No. Puesto que el tamaño del vector es de 20 elementos, 19 son para
almacenar los caracteres y la restante para almacenar '\0'.
```

- ✓ Sea cual sea el código empleado, ¿cómo mostrarías por pantalla la inicial de tu nombre?

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

const int ktam = 20;

int main() {
  char nombre[ktam];

  cout << "Dime tu nombre: ";
  cin.getline(nombre, ktam);

  cout << "Tu nombre empieza por "<< nombre[0] << endl;
}
```

Funciones de tratamiento de cadenas existentes en C++

Cabe preguntarse ¿qué operaciones para manejar cadenas de caracteres completas se podrían necesitar? Algunos ejemplos podrían ser:

- Hallar la longitud de una cadena de caracteres.
- Combinar dos cadenas de caracteres.
- Comparar dos cadenas de caracteres.
- Buscar una cadena (o subcadena) dentro de una cadena de caracteres.

C/C++ proporciona estas operaciones, y muchas más, a través de las funciones definidas en el archivo o archivos string.h/cstring. Podemos concatenar cadenas, copiarlas, compararlas, etc. A continuación mostramos algunas de ellas:

Función	Significado
`int strlen(char [])`	Longitud de la cadena de caracteres.
`strcat(char [], char [])`	Concatenación de cadenas de caracteres.
`strncat(char [], char [], int)`	Concatenación de n caracteres.
`int strcmp(char [], char [])`	Comparación de cadenas de caracteres.
`int strncmp(char [], char [], int)`	Comparación de n caracteres.

strlen()

La función `strlen()` calcula la longitud de una cadena de caracteres. La longitud de una cadena de caracteres es un entero que indica el número de caracteres que forman parte de la cadena, excluyendo el carácter nulo del final. Por ejemplo, para una cadena definida como

```
char alfabeto[]="abcdefghijklmnopqrstuvwxyz";
```

la instrucción

```
strlen(alfabeto)
```

devolverá un valor de 26.

strcat() y strncat()

Las funciones `strcat()` y `strncat()` se usan para **concatenar** dos cadenas de caracteres. Cuando se concatenan dos cadenas de caracteres, los contenidos de la segunda cadena se copian al final de la primera. En este caso, se usa `strcat(cad1, cad2)` para combinar las dos cadenas. Es importante que la primera cadena tenga definida una longitud suficiente como para almacenar la cadena concatenada. Dicha función necesita dos argumentos: los nombres de las cadenas a concatenar. El primer argumento será el destino de la nueva cadena.

Cuando no se necesite concatenar la segunda cadena completa, se debería usar la función `strncat(cad1, cad2, n)`. Esta función concatena a la primera cadena sólo los n primeros caracteres de la segunda cadena. Como la función necesita el valor de `n`, la función tiene tres argumentos:

```
strncat(cad1, cad2, n)
```

strcmp() y strncmp()

Las funciones `strcmp()` y `strncmp()` se usan para comparar dos cadenas de caracteres. La comparación se hace carácter a carácter. Cada función devuelve un valor que depende del resultado de la comparación. El valor entero devuelto es:

- *0* si las cadenas son idénticas (iguales caracteres y longitud).
- *<0* si la primera cadena precede alfabéticamente a la segunda.
- *>0* si la segunda cadena precede alfabéticamente a la primera.

La función `strncmp()` necesita un tercer argumento, `n`, que especifica el número de caracteres a comparar.

- ✓ Realiza un programa que pida por teclado una frase y almacene en otra variable únicamente las 4 primeras letras. Posteriormente las debe mostrar.

```
#include <iostream>
#include <cstring>
```

```
using namespace std;

const int ktam = 40;
const int kmini = 10;

int main() {
  char texto[ktam], reducido[kmini];

  cout << "Introduce una frase: ";
  cin.getline(texto, ktam);
  strncpy(reducido, texto, 4);
  cout << "Tu frase es: " << texto << endl;
  cout << "Las 4 primeras letras son: " << reducido << endl;
}
```

- ✓ Realiza un programa que te pida el nombre, posteriormente el apellido (guardará cada dato en una cadena distinta) y cree una nueva cadena de teto que contenga los dos, separados por un espacio.

```
#include <iostream>
#include <cstring>

using namespace std;

const int ktam = 40;

int main() {
  char nombre[ktam], apellido[ktam], completo[ktam];

  cout << "Introduce tu nombre: ";
  cin.getline(nombre, ktam);
  cout << "Introduce tu apellido: ";
  cin.getline(apellido, ktam);

  strcpy(completo, nombre);
  strcat(completo, " ");
  strcat(completo, apellido);
  cout << "Tu nombre completo es: " << completo << endl;
}
```

- ✓ Realiza un programa que pida dos palabras y diga si hemos tecleado la misma palabra dos veces.

```
#include <iostream>
#include <cstring>

using namespace std;

const int ktam = 40;

int main() {
  char texto1[ktam], texto2[ktam];

  cout << "Introduce una palabra: ";
  cin.getline(texto1, ktam);
  cout << "Introduce otra palabra: ";
  cin.getline(texto2, ktam);

  if (strcmp(texto1, texto2)==0)
        cout << "Son iguales" << endl;
  else
```

```
            cout << "Son distintas" << endl;
    }
```

- ✓ Modifica el programa anterior para que te indique si son iguales o cual es mayor que la otra (según nuestro sistema alfabético).

```
Debería sustituir la parte del if-else por:

  if (strcmp(texto1, texto2) == 0)
        cout << "Son iguales" << endl;
  else if (strcmp(texto1, texto2) > 0)
          cout << texto1 << " es mayor que " << texto2 << endl;
       else
          cout << texto1 << " es menor que " << texto2 << endl;
```

- ✓ Crea un programa que almacene una lista de 5 cadenas de texto y muestre, por ejemplo, la segunda cadena.

```
#include <iostream>
#include <cstring>

using namespace std;

const int kfilas = 5;
const int kcolumnas = 80;

int main() {
  char mensajeError[kfilas][kcolumnas] = {
                "Fichero no encontrado",
                "El fichero no se puede abrir para escritura",
                "El fichero está vacío",
                "El fichero contiene datos de tipo incorrecto"
                "El fichero está siendo usado"
      };

  cout << "El segundo mensaje de error es: " << mensajeError[1] <<
endl;
}
```

Argumentos del `main()`

En un programa escrito en C, su ejecución empieza por el `main`(). Las posibles formas de definir la función `main()` son:

- `int main()`
- `int main(int argc, char *argv[])`

En la primera forma no se recoge ningún parámetro mientras que la segunda definición recoge dos parámetros. Estos parámetros se usan para reflejar los elementos de la línea de ejecución que se pasan al ejecutar el programa.

El siguiente programa (programa.c) muestra cómo acceder y visualizar en la pantalla los párametros del main:

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

int main(int argc, char *argv[]){
  int i;

  cout << "Se han pasado " << argc << " argumentos" <<endl;
  for (i=0; i<argc; i++)
      cout << "argumento " << i << " " << argv[i] << endl;
  return 0;
}
```

El retorno de la función main() es un valor entero. Su significado suele referirse a la ejecución correcta del programa. Valor 0: el programa terminó correctamente. Valor 1: el programa terminó con error.

Todos los elementos de la línea de ejecución se almacenan en las variables que recibe como parámetros la función `main()`. Estos parámetros son:

- `argc`: parámetro de tipo entero que indica el número de elementos de la línea de ejecución (incluyendo el nombre del programa)
- `argv`: parámetro de tipo array de cadenas de caracteres. Cada elemento del array es un elemento de la línea de ejecución (incluyendo el nombre del programa)

La ejecución del programa anterior pasando 3 valores sería:

```
lineadecomando:~$ programa p1 p2 p3
Se ha pasado 4 argumentos
argumento 0 programa
argumento 1 p1
argumento 2 p2
argumento 3 p3
```

La línea de ejecución del ejemplo anterior quedaría almacenada así:

```
argc->4
argv[0]->"programa"
argv[1]->"p1"
argv[2]->"p2"
argv[3]->"p3"
```

Búsqueda y ordenación

Dos importantes aplicaciones en que intervienen arrays son la búsqueda y la ordenación. La primera implica la búsqueda de un valor específico en un array y el informe de su ubicación (subíndice). Si éste no se encuentra, deberá informarse de este hecho. En la segunda aplicación interviene el ordenamiento de los elementos de un array en sentido creciente o decreciente. Para simplificar las cosas, imaginaremos que los arrays son de enteros. Pese a ello, las técnicas desarrolladas se aplicarán a cualquier tipo de datos para los cuales tengan sentido las comparaciones de orden. Esto incluye específicamente a tipos enteros, reales, cadena y escalares definidos por el usuario.

Métodos de búsqueda

La búsqueda es una operación que tiene por objeto la localización de un elemento dentro de la estructura de datos. El elemento que se busca se caracteriza o distingue por el valor de una clave o la combinación de varias si la clave es compleja. A partir de ahora cuando se haga mención a la "clave'', se entenderá por ella el *valor*, simple o no, que caracteriza al elemento.

Si la búsqueda se realiza en un array ordenado ésta puede realizarse secuencialmente finalizando al encontrar el elemento a detectar en un componente del array con clave mayor que el valor buscado, o bien hacerla de forma binaria o dicotómica.

Búsqueda secuencial o lineal

La búsqueda secuencial es la técnica más simple para buscar un elemento en un array. Consiste en recorrer el array elemento a elemento e ir comparando con la clave o valor buscado. El recorrido termina al encontrar el elemento o bien porque se rebasa la posición del último componente del array. La búsqueda puede realizarse en dos circunstancias diferenciadas: estando el array desordenado o estando ordenado de acuerdo a la clave buscada.

Para ésta y las otras aplicaciones, supondremos que manipulamos un array entero *A* con subíndices que varían de 0 a una constante *kIndiceMax-1*. También suponemos que el array contiene en realidad *N* valores, donde $0 \leq N \leq kIndiceMax$.

En problemas de búsqueda, también se nos da un valor que debemos buscar en el array. En la *búsqueda lineal* simplemente comenzamos con el primer elemento *A[0]* y procedemos a través del array hasta que ocurre una de dos condiciones:

- Llegamos al final de la parte del array que contiene valores.
- Encontramos el valor que buscamos.

Debemos tener cuidado al escribir la condición del ciclo. A continuación se muestra un subprograma que realiza la búsqueda.

En realidad hay dos "respuestas'' en este subprograma. Estas responden las preguntas:

- Está presente el valor?
- Si lo está, ¿en qué parte está?

Sin embargo, estableciendo la ubicación igual a -1 si no se encuentra el valor, podemos responder a ambas preguntas en una sola variable.

```
int  busqueda(int s[], int clave, int lon) {
    int i, busc;
    bool encontrado;

    encontrado=false;
    busc=-1;
    i=0;
    while ((i<lon) && (!encontrado) ){
       if (s[i]==clave)
           encontrado=true;
       else i=i+1;
    }
    if (encontrado)
       busc=i;

    return(busc);
}
```

Búsqueda binaria o dicotómica

La búsqueda binaria es el método más eficiente para encontrar elementos en un array ordenado. El proceso comienza comparando el elemento central del array con el valor buscado. Si ambos coinciden haremos búsqueda binaria en el subarray superior, es decir, el delimitado por los índices *medio+1* y *N-1*, donde *medio = (N) div 2*. Si el elemento es menor haremos búsqueda binaria entre los índices 0 y *medio-1*. El proceso finaliza cuando se encuentra el elemento o cuando el subarray de búsqueda se queda sin elementos. Este método exige que la lista esté ordenada. Un programa completo con el procedimiento de búsqueda binaria es:

```
int busquedabinaria(int s[], int clave, int lon) {
    int inicio, final, medio,  busc;
    bool encontrado;

    encontrado = false;
    inicio = 0;
    busc = -1;
    final = lon-1;
    while ((inicio <= final) && (!encontrado)) {
       medio = (inicio+final) / 2;
       if (s[medio] == clave )
              encontrado = true;
        else  if (s[medio] < clave)
                  inicio = medio + 1;
              else
                  final = medio - 1;
    }
    if (encontrado)
          busc = medio;
    return(busc);
}
```

Como puede verse por el desarrollo del algoritmo, la búsqueda binaria es de naturaleza recursiva, ya que si el elemento no se encuentra en la posición central del array, se procede a seguir buscándolo en el subarray superior o inferior, según el valor de la clave. Esto, evidentemente, conduce a una situación similar a la anterior, excepto que la zona de búsqueda es más pequeña. El proceso continúa o bien hasta encontrar el elemento o cuando el subarray de búsqueda no contiene elementos `(inicio>final)`.

Métodos de ordenación

Dado un array *A* que contiene *N* valores, deseamos reordenar el array de modo que los valores se encuentren en orden creciente.

Una de las tareas más frecuentes durante el procesamiento de los datos es la ordenación o clasificación de un conjunto de valores de acuerdo con el contenido de un campo clave o referencial.

Existen numerosos métodos de ordenación. La principal condición que tienen que cumplir los métodos de ordenación interna es el empleo eficiente de la memoria de la computadora. En otras palabras, la ordenación de los items, o componentes de la estructura, debe realizarse sobre las mismas posiciones de memoria ocupadas por los datos, salvo alguna posición auxiliar.

Existen diversos algoritmos de ordenación, a continuación se verán tres de ellos (intercambio, selección e inserción), distinguiéndose cada una de ellos por el método base que utilizan para colocar los datos en el orden final deseado.

Ordenación por intercambio

La idea subyacente a todos los métodos de intercambio es la mera comparación de elementos e intercambiarlos si su posición actual o inicial es contraria o inversa a la deseada.

El método de ordenación por burbuja, es, sin género de dudas, el más popular, sencillo e intuitivo y, por tanto, el más fácil de entender (y también el menos eficiente). Este método se basa en la comparación de elementos adyacentes e intercambio de los mismos si éstos no guardan el orden deseado.

Si se comparan e intercambian, si ha lugar, elementos consecutivos o adyacentes del array, es evidente que después de realizar un recorrido o pasada por toda la estructura, el elemento mayor se encontrará en la posición más alta del array. En pasadas sucesivas se irán colocando en las posiciones más altas los inmediatamente menores. Por tanto, el número de pasadas necesarias para que el array resulte ordenado es tantas como elementos menos uno (*N-1*), ya que en la última pasada se colocarán los dos elementos más pequeños en sus posiciones finales.

Supongamos que deseamos ordenar un vector v, el procedimiento siguiente denominado procedimiento burbuja muestra cual es el proceso a seguir:

```
void burbuja(int v[], int n) {
   int aux, i, j;

   for (i = 1; i < n; i++)
     for ( j = n-1; j >= i; j--)
        if (v[j-1] > v[j]) {
            aux = v[j-1];
            v[j-1] = v[j];
            v[j] = aux;
        }
}
```

Ordenación por selección

Los métodos de ordenación por selección se basan en dos principios básicos:

- Seleccionar el elemento más pequeño (o más grande) del array.
- Colocarlo en la posición más baja (o más alta) del array.

```
void seleccion(int v[], int n) {
   int aux, i, j, k;

   for (k = 0; k < n-1; k++) {

      i = k;
      j = k+1;
      while (j < n) {
        if (v[j] < v[i])
           i = j;
        j = j+1;
      }
      aux = v[k];
      v[k] = v[i];
      v[i] = aux;
   }
```

Ordenación por inserción

El fundamento de este método consiste en insertar los elementos no ordenados del array en subarrays del mismo que ya estén ordenados. Es evidente que si tenemos un subarray de *N-1* elementos ya ordenado, si introducimos

un elemento en la posición adecuada, obtendremos un array de *N* elementos ordenado. Dependiendo del método elegido para encontrar la posición de inserción tendremos distintas versiones del método de inserción.

Como un array con un único elemento (*A[1]*) es un array ordenado, si el array contiene *N* elementos el número de inserciones necesarias para ordenar completamente el array será *N-1*, y empezará a realizarse la primera a partir de *A[2]* hasta *A[N]*.

```
void insercionDirecta(int s[],int n){
    int i,j,aux;

    for (i = 1; i < n; i++) {
       aux = s[i];
       j = i - 1;
       while (j >= 0 && s[j] > aux) {
           s[j+1] = s[j];
           j--;
       }
       s[j+1] = aux;
    }
}
```

Tipos de datos definidos por el usuario

En muchas ocasiones es útil crear nuevos tipos de datos para mejorar la legibilidad de los programas. Para poder crear un nuevo tipo de datos en C/C++, se utiliza la palabra reservada `typedef` con la siguiente sintaxis:

```
typedef tipo_de_datos nuevo_tipo;
```

Donde `nuevo_tipo` es el nuevo tipo de que se desea crear y `tipo_de_datos`, el tipo ya definido en C, que se utilizará para el nuevo tipo.

Veamos ejemplos sin utilizar nuevos tipos:

```
int main() {
   float notasProg1[50];
   float notasProg2[50]
   char nomAlum1[30];
   char nomAlum2[30];
   int matriz1[3][3];
   int matriz2[3][3];

    ...
}
```

Si utilizáramos una nueva definición de tipos de datos:

```
// Definición de tipos de datos
typedef   int   Tenteros[20];
typedef   float  Tnotas[50];
typedef   char  Tcadena[30];
```

```
typedef   int    Tmatriz[3][3];

int main(){
   Tnotas notasProg1, notasProg2;
   Tcadena nomAlum1, nomAlum2;
   Tmatriz matriz1, matriz2;

   ...
}
```

Para entender el funcionamiento de `typedef` debemos pensar que éste se "anota" la definición de un tipo nuevo que hacemos, para posteriormente, cuando declaremos una variable de ese tipo, llevar a cabo una sustitución de nombre, el de la variable por el del tipo en la línea donde lo definimos con `typedef`.

Volviendo a los ejemplos anteriores, hacer una declaración como ésta:

```
Tcadena nomAlum1, nomAlum2;
```

Es equivalente a

```
char nomAlum1[30], nomAlum2[30];
```

En realidad lo que hace `typedef` es sustituir `Tcadena` por los nombres `nomAlum1` y `nomAlum2` del ejemplo anterior en la línea del `typedef`, obteniendo así unas declaraciones de variables sintácticamente correctas.