5 Arrays

Contenido

5.1.- Introducción.

En los temas anteriores se han estudiado los diferentes tipos de datos simples de C++, usados para representar valores simples como enteros, reales o caracteres. Sin embargo, en muchas situaciones se precisa procesar una colección de valores que están relacionados entre sí, como por ejemplo una lista de calificaciones, una tabla de temperaturas, etc. El procesamiento de tales conjuntos de datos utilizando tipos simples puede ser extremadamente complejo y tedioso, y por ello la mayoría de los lenguajes de programación incluyen mecanismos sintácticos para manipular agrupaciones de datos. Son las llamadas "*estructuras de datos*".

Definición: Una *estructura de datos* es una colección de datos que pueden ser caracterizados por su organización y por las operaciones que se definan en ella.

En una estructura de datos podremos hacer referencia tanto a la colección de elementos completa como a cada uno de sus componentes de forma individual.

La estructura de datos más básica soportada por los lenguajes es el "***array***".

5.2.- Utilidad de los arrays.

Veamos mediante un ejemplo la necesidad y utilidad de los arrays para posteriormente desarrollar en detalle todas las características de los arrays en C++.

Consideremos el siguiente caso: Una casa comercial tiene en plantilla 20 agentes de ventas (identificados por números del 1 al 20) que cobran comisión sobre la parte de sus operaciones comerciales que excede los 2/3 del promedio de ventas del grupo.

Se necesita un programa que lea el valor de las operaciones comerciales de cada agente e imprima el número de identificación de aquellos que deban percibir comisión así como el valor correspondiente a sus ventas.

Este problema tiene 2 aspectos que juntos lo hacen difícil de programar utilizando únicamente tipos de datos simples:

- Existe un procesamiento similar sobre los datos de cada agente:
    1. leer ventas
    2. calcular promedio de ventas
    3. comparar niveles
    4. decidir si debe o no recibir comisión
- Se necesita almacenar durante la ejecución del programa los valores de las ventas de cada agente para el cálculo del promedio y para la comparación de niveles.

Esto implica que necesitaríamos 20 variables para retener los valores de las ventas, una por cada agente: ventas1, ventas2, ..., ventas20.

Un posible programa sería:

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const double PORCION = 2.0 / 3.0;

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      double ventas1, ventas2,..., ventas20;
      double suma, umbral;

      // Zona de instrucciones
      suma = 0.0;
      cout << "Introduzca las ventas del agente 1: ";
      cin >> ventas1;
      suma += ventas1;
      cout << "Introduzca las ventas del agente 2: ";
      cin >> ventas2;
      suma += ventas2;

      ......

      cout << "Introduzca las ventas del agente 20: ";
```

```
      cin >> ventas20;
      suma += ventas20;

      umbral = PORCION * (suma / 20.0);   // Calcula el umbral de comisión
```

```
      cout << "Agentes que superan el umbral y sus ventas" << endl;
      if (ventas1 > umbral)
            cout << 1 << ": " << ventas1 << endl;

      if (ventas2 > umbral)
            cout << 2 << ": " << ventas2 << endl;

      .......

      if (ventas20 > umbral)
            cout << 20 << ": " << ventas20 << endl;

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}
```

La escritura de este programa es, como se ve, bastante tediosa.

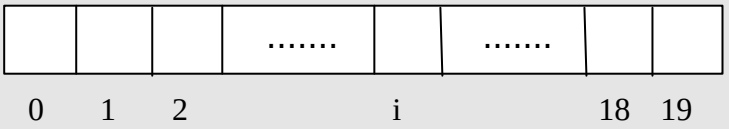
El problema se agravaría más si en lugar de 20 hubiera 200 agentes de ventas. Necesitaríamos 200 variables, 200 lecturas de datos y 200 comprobaciones del umbral.

Una mejor solución consiste en agrupar estas variables de ventas en una estructura de datos "*array*".

Un array se puede visualizar como una colección de cajas que representan variables de un mismo tipo de datos, tal y como se ilustra en la siguiente figura. A este tipo de array se le suele denominar **array unidimensional o vector**.

Así:

```
double ventas [20];
```



Para hacer referencia a una caja o componente en particular, usaremos el nombre o identificador del array seguido por un número (índice) entre corchetes, que indicará su posición dentro del array. En este caso dicho índice sería función del número de identificación del correspondiente agente de ventas.

`ventas [i]` almacenará las ventas realizadas por el agente número (i + 1).

Realmente lo que va entre corchetes, al referirse a la posición de un componente de un array, puede ser cualquier expresión que al evaluarse dé como resultado un valor numérico entero positivo.

De esta forma podremos tratar a los agentes de ventas de forma repetitiva, y nuestro programa podría escribirse de la siguiente manera:

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const double PORCION = 2.0 / 3.0;

// Programa Principal
```

```
int main()
{
      // Zona de Declaración de Variables del Programa principal
      double ventas [20];
      double suma, umbral;
      unsigned int i;

      // Zona de instrucciones
      suma = 0.0;
      for (i = 0; i < 20; i ++)     // Para recoger ventas de agentes
      {
            cout << "Introduzca las ventas del agente " << i + 1 << ": ";
            cin >> ventas [i];
            suma += ventas [i];
      }

      umbral = PORCION * (suma / 20.0);   // Calcula el umbral de comisión

      cout << "Agentes que superan el umbral y sus ventas" << endl;
      for (i = 0; i < 20; i ++)     // Para determinar cobro comisiones
      {
            if (ventas [i] > umbral)
                  cout << i + 1 << ": " << ventas [i] << endl;
      }

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}
```

Si es posible que el número de agentes vaya a variar en algún momento de la vida de la empresa, es preferible definir una **constante global** (`const unsigned int NAGENTES = 20`), y ésta se pone en los lugares del algoritmo en los que antes hemos puesto 20. Con esto, al cambiar el número de agentes, sólo hay que reflejarlo en la declaración de la constante y no en todos los sitios en los que aparezca la cantidad 20, como ocurrirá en el programa anterior.

Siempre lo haremos así a partir de ahora.

La versión final de nuestro algoritmo quedaría entonces como:

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int NAGENTES = 20;
const double PORCION = 2.0 / 3.0;

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      double ventas [NAGENTES];
      double suma, umbral;
      unsigned int i;

      // Zona de instrucciones
      suma = 0.0;
```

```
for (i = 0; i < NAGENTES; i ++)     // Para recoger ventas de agentes
{
      cout << "Introduzca las ventas del agente " << i + 1 << ": ";
      cin >> ventas [i];
      suma += ventas [i];
}
```

```
      umbral = PORCION * (suma / double (NAGENTES));   // Umbral de comisión

      cout << "Agentes que superan el umbral y sus ventas" << endl;
      for (i = 0; i < NAGENTES; i ++)     // Para determinar cobro comisiones
      {
            if (ventas [i] > umbral)
                  cout << i + 1 << ": " << ventas [i] << endl;
      }

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}
```

Nótese como en el cálculo del umbral utilizamos una promoción explícita (casting) del tipo de la constante `NAGENTES` para convertirla de **`unsigned int`** a **`double`**, aunque realmente esta conversión de tipos explícita no es necesaria ya que C++ promociona automáticamente a **`double`** todos los datos de la expresión aritmética anterior.

5.3.- Arrays en C++.

Un *array* es una estructura de datos homogéneos, ordenados, finita y de tamaño fijo.

- ***Homogéneos*** significa que todos los elementos son del mismo tipo (a partir de ahora a este tipo se le denominará ***tipo base***).
- ***Ordenados*** significa que hay un primer elemento, un segundo elemento, y así sucesivamente. Además cada uno de los componentes o elementos de la estructura son igualmente accesibles, y pueden seleccionarse de forma directa, indicando la posición que ocupa la componente dentro de la estructura. De esta forma, el tipo de datos *array* tiene asociado un ***tipo índice***, que permite seleccionar todos y cada uno de los elementos que componen la estructura.
- ***Finita*** significa que hay también un último elemento.
- ***Tamaño fijo*** significa que el tamaño del array debe ser conocido en tiempo de compilación; pero no significa que todos los elementos del array tengan significado. Este tamaño fijo del array se conoce con el nombre de ***dimensión*** del array.

Una variable de array se puede ver como un conjunto de variables del mismo tipo a las que denominamos con un nombre común.

Algunas características importantes de los arrays en C++ son:

- Los elementos del array se almacenan consecutivamente en memoria.
- El nombre del array especifica la dirección en memoria del primer elemento del mismo.
- El nombre del array es una dirección constante, es decir no puede modificarse su valor.
- Como tipo estructurado, no esta permitida la lectura y escritura directa sobre dispositivos de E/S estándares. Este tipo de operaciones debe realizarse elemento a elemento.
- Todas las operaciones disponibles sobre los tipos asociados a las componentes pueden usarse sobre ellos.

Un ***tipo array*** estará formado por todos los posibles arrays con componentes de tipo base T y con un determinado tipo de índice I (todos los arrays del mismo tipo poseen la misma cantidad de elementos) que se puedan formar.

5.3.1.- Definición de arrays.

Un rasgo importante de los arrays en C++ es que el tipo índice es siempre un **subrango de naturales empezando por 0**, por lo que a la hora de definir un array solo será necesario indicar su dimensión:

Así definimos una variable de un tipo array como:

tipo nombre_array [dimensión];

Donde "*tipo*" es el tipo base de los componentes del array y "*dimensión*" es un número natural mayor de cero que representa la longitud o dimensión del array.

En caso de definir varias variables de array con el mismo tipo base se hará de la forma:

tipo nombre$_1$ [dimensión$_1$], nombre$_2$ [dimension$_2$], ..., nombre$_n$ [dimensión$_n$];

Ejemplos:

```
      int vector [10];                    // Array de 10 enteros
      char frase [80];                    // Array de 80 caracteres
      bool flags [5], logicos [10];       // Un array de 5 lógicos y
                                          //  otro de 10 lógicos
```

Aunque lo más adecuado, tal como se ha comentado en el programa de ejemplo del apartado anterior, es usar **constantes globales** para definir la dimensión de los arrays.

```
const unsigned int MAXVECTOR = 10;
const unsigned int MAXFRASE = 80;
const unsigned int MAXFLAGS = 5;
const unsigned int MAXLOGICOS = 10;

int vector [MAXVECTOR];                         // Array de 10 enteros
char frase [MAXFRASE];                          // Array de 80 caracteres
bool flags [MAXFLAGS], logicos [MAXLOGICOS];    // Un array de 5 lógicos y
                                                //  otro de 10 lógicos.
```

En BNF:

```
    <array> ::= [<cualificador>] [<almacenamiento>] <nombre_tipo>
         <identificador> "["<dimension>"]" {,<identificador>
                         "["<dimension>"]"};
```

Para acceder a una determinada componente de un array escribiremos:

nombre_array [i]

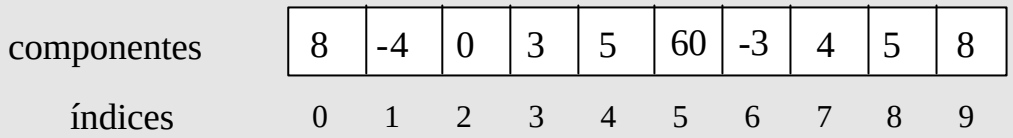
donde `i` es un número natural, denominado "**índice**", que representa el número de orden del componente que queremos acceder. El primer componente lo denotaremos con el valor índice `0`, el segundo con el valor `1`, el i-ésimo con el valor `i-1`, etc. El último elemento del array se podrá seleccionar mediante "`nombre_array [dimensión - 1]`".

Ejemplo:

```
      const unsigned int MAXA = 10;
```

```
int a [MAXA];
```

En el ejemplo "**a**" será un array que contiene 10 elementos de tipo entero, cada uno de ellos con un índice asociado (0 a 9). Gráficamente, por ejemplo:



Así:

- `a [4]` almacena el valor `5`
- `a [7]` almacena el valor `4`
- `a [1]` almacena el valor `-4`

Hay que señalar que el *array* "a" comprende toda la estructura, mientras que cada uno de los elementos pertenece al tipo ***int*** (en este caso el tipo base).

En resumen, *se puede seleccionar cada uno de los elementos del array* indicando la posición que ocupa dentro del mismo.

Siguiendo con el ejemplo de la figura, para seleccionar la variable entera que es el tercer componente del array (el de valor 0), basta con que al nombre del *array* le siga la posición que ocupa dicho componente dentro del *array* entre corchetes empezando por 0, es decir en este caso, "a `[2]`".

5.4.- Definición de tipos. Sentencia `typedef`.

Aunque como hemos visto se pueden definir variables de array al mismo tiempo que se define su tipo base y su dimensión (definición implícita de tipos array), lo correcto es definir separadamente los tipos de datos, asignándoles un nombre, y posteriormente declarar las variables como pertenecientes a los tipos anteriormente definidos.

C++ incorpora un mecanismo para definir nuevos tipos de datos, la sentencia ***typedef***.

Sintaxis:

```
typedef tipo_conocido nombre_nuevo_tipo;
```

En BNF:

```
<definición_tipo> ::= typedef <nombre_tipo> <identificador>;
```

Ejemplo: Declarar variables de tipo Tcontador:

```
typedef unsigned int Tcontador;

Tcontador i, j;
Tcontador K = 1000;
```

El objetivo fundamental de esta sentencia es mejorar la claridad del código fuente asociando nombres simbólicos a tipos de datos definidos por el programador.

Los pasos a seguir para definir un nuevo *nombre de tipo* son los siguientes:

1. Escribir la sentencia de definición como si una variable del tipo deseado se estuviera declarando.
2. En el lugar donde aparece normalmente el nombre de la variable declarada, poner el nuevo nombre de tipo.
3. Delante de todo esto colocar la palabra ***typedef***.
4. Posteriormente utilizar el nombre definido como el de un tipo de datos para definir nuevas variables.

Ejemplo: Definir la variable "a" como perteneciente a un tipo array de 10 enteros:

```
1.  int a [10];
2.  int Tarray [10];
3.  typedef int Tarray [10];
4.  Tarray a;
```

Según esto, en el caso de definición de tipos array la sintaxis concreta sería entonces:

```
typedef tipo_base nombre_nuevo_tipo [dimensión];
```

En BNF:

```
<definición_tipo_array> ::= typedef <nombre_tipo_base>
          <identificador> "["<dimension>"]";
```

Ejemplos:

```
typedef int Tvector [4];
typedef char Ttexto [10];
typedef unsigned int Tlista [7];
```

Posteriormente se pueden definir variables de estos tipos:

```
Tvector v;
Ttexto      palabra;
Tlista      mi_lista;
```

Nota: Sólo se puede definir un único tipo en cada sentencia ***typedef***.

Importante: Por cuestiones de ámbito es recomendable definir los tipos mediante la sentencia ***typedef*** de forma global fuera de los subprogramas, al principio del texto del programa. De esta forma serán visibles tanto para definir los datos involucrados en parámetros reales como los correspondientes parámetros formales de las funciones.

5.5.- Operaciones con arrays.

Las operaciones que se suelen realizar con arrays durante el proceso de resolución de un problema son:

- Acceso a los elementos.
- Asignación.

- Igualdad.
- Lectura/escritura.
- Recorrido (acceso secuencial).

En general, las operaciones con arrays implican el procesamiento o tratamiento de los elementos individuales del vector o array.

5.5.1.- Acceso a elementos.

Ya hemos visto anteriormente como se realiza el acceso a los elementos individuales de un array. Se puede acceder a cada componente de una variable de array usando su nombre seguido de lo que denominamos **índice** entre corchetes. El índice de un elemento de un array es un valor natural que indica el orden del elemento dentro del array.

```
nombre_array [indice]
```

En C++ los ***índices comienzan siempre por 0***, es decir el primer elemento de un array tiene índice 0.

Es importante comprender que el valor que representa el acceso a una de las componentes del *array*, pertenece al tipo base del *array*, pudiéndose utilizar dicha referencia (o de cualquier otro componente del array) en cualquier situación en que lo pueda ser una variable del mismo tipo base. Como por ejemplo a la izquierda de una sentencia de asignación, dentro de una expresión, como parámetro de un subprograma, etc.

Todas las operaciones disponibles sobre los tipos de los componentes pueden usarse sobre ellos sin ninguna restricción.

Ejemplo:

```
...
typedef unsigned int Tarray20 [20];   // Tipo array de 20 naturales
...
void Subprograma (unsigned int a);    // Prototipo de procedimiento
...

...
Tarray20 nuevoArray;                  // Variable de array
unsigned int i;

nuevoArray [19] = 5;           // Asigna el valor 5 a la última
                               //  componente del vector nuevoArray.

i = nuevoArray [0] + 2;        // Asigna el valor del primer
                               //  componente del array más dos a la
                               //  variable i

Subprograma (nuevoArray [5]); // Pasa como parámetro al procedimiento
                               //  Subprograma el valor contenido en el
                               //  elemento sexto del array, el que
                               //  tiene como valor índice 5
...
```

Otro aspecto importante del índice asociado a las componentes de un *array* es que indica un orden dentro de la estructura, así podemos acceder a la primera componente, a la segunda componente y así sucesivamente. Esto implica que **el índice es calculable**, facilitando el tratamiento de las componentes del *array*.

Es decir, los valores de índice pueden ser cualquier expresión que de cómo resultado un número natural inferior a la dimensión del array.

Ejemplo:

```
      typedef unsigned int Tarray20 [20];     // Tipo array de 20 naturales

      ...
      Tarray20 nuevoArray;                    // Variable de array
      insigned int i;                         // Variable natural

      i = 6;
      nuevoArray [i + 1] = 2;  // Asigna el valor 2 a la octava componente
                               //  del array, aquella con índice 7
```

C++ no produce ningún error en **indexaciones del array fuera de rango**, es decir valores de índice que excedan su dimensión., por lo que hay que extremar la precaución a la hora de seleccionar elementos de un array.

Ejemplo:

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXARRAY = 5;

// Zona de Declaración de Tipos
typedef unsigned int Tarray [MAXARRAY];

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      Tarray array;
      unsigned int indice;

      // Zona de instrucciones
      for (indice = 0; indice < MAXARRAY; indice ++)
            array [indice] = indice * 10;

      for (indice = 0; indice <= MAXARRAY; indice ++)
            cout << indice << " : " << array [indice] << endl;

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}
```

Nótese como existe un error en el segundo bucle, ya que la variable de control llega a tomar el valor `MAXARRAY` (5 en este caso) con lo que se intenta seleccionar un elemento que no existe en el array.

Sin embargo el compilador no detectará ningún error, y al ejecutar el programa el resultado será el siguiente:

```
0 : 0
1 : 10
2 : 20
3 : 30
4 : 40
```

```
5 : 2013281329
Presione una tecla para continuar . . .
```

Obsérvese como al no existir en el array un elemento con índice 5, lo que aparece en pantalla es cualquier cosa que haya en memoria a continuación del array. De hecho, en diferentes máquinas e incluso en distintas ejecuciones del programa sobre la misma máquina, el valor para el elemento con índice 5 podrá ser distinto.

5.5.2.- Asignación.

Como se ha visto anteriormente, se puede asignar valor a un array, asignando valores a sus componentes individuales.

Vamos a intentar asignar el valor de una variable array a otro array de forma directa como en el siguiente ejemplo.

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXARRAY = 3;

// Zona de Declaración de Tipos
typedef int Tarray [MAXARRAY];

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      Tarray a, b;

      // Zona de instrucciones
      b [0] = 3;              // Da valor al array b
      b [1] = 2;
      b [2] = 1;

      a = b;                  // Asignación completa de arrays. INCORRECTO.

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}
```

El compilador detectará el siguiente error: "**`warning: ANSI C++ forbids assignment of arrays`**". Nos está avisando de que no se permite la asignación completa de arrays.

Por tanto tendremos que asignar los valores elemento a elemento, ayudándonos de una estructura iterativa, tal como se presenta a continuación:

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXARRAY = 3;

// Zona de Declaración de Tipos
typedef int Tarray [MAXARRAY];

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
```

```
      Tarray a, b;
      unsigned int i;         // Para seleccionar elementos de los arrays

      // Zona de instrucciones
      b [0] = 3;              // Da valor al array b
      b [1] = 2;
      b [2] = 1;


      for (i = 0; i < MAXARRAY; i ++)
            a [i] = b [i];  // Asignación completa por componentes. CORRECTO.

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}
```

Por otro lado, en C++ se puede inicializar el valor de todo un array en la declaración del mismo de la siguiente forma:

```
tipo_array nombre_array = {valor_1, valor_2, ...,
```

De esta forma es posible definir **arrays constantes**, añadiendo el modificador "***const***" de la siguiente forma:

```
const tipo_array nombre_array = {valor_1, valor_2, ... , valor_N};
```

Ejemplo:

```
      const int MAXCAD = 10;
      typedef char Tcadena [MAXCAD];
      const Tcadena vocales = {'A','a','E','e','I','i','O','o','U','u'};
```

En este caso el array `vocales` no podrá ser modificado.

Importante: La inicialización de arrays de esta forma sólo es posible en su declaración, no se puede utilizar como parte de una asignación en el código.

Veamos otro ejemplo sobre un sencillo programa completo:

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXMESES = 12;

// Zona de Declaración de Tipos
typedef unsigned int Tdias [MAXMESES];

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      const Tdias dias = {31,28,31,30,31,30,31,31,30,31,30,31};
```

```
      unsigned int indice;

      // Zona de instrucciones
      for (indice = 0; indice < MAXMESES; indice ++)
            cout << "El mes " << indice + 1 << " tiene " << dias [indice]
                 << " dias. " << endl;

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}
```

El resultado de la ejecución de este programa será:

```
El mes 1 tiene 31 dias.
El mes 2 tiene 28 dias.
El mes 3 tiene 31 dias.
El mes 4 tiene 30 dias.
El mes 5 tiene 31 dias.
El mes 6 tiene 30 dias.
El mes 7 tiene 31 dias.
El mes 8 tiene 31 dias.
El mes 9 tiene 30 dias.
El mes 10 tiene 31 dias.
El mes 11 tiene 30 dias.
El mes 12 tiene 31 dias.
Presione una tecla para continuar . . .
```

De todas formas, hay que tener en cuenta que este programa no es correcto, fallará cada cuatro años, ya que no contempla los años bisiestos.

Si en el programa intentamos cambiar el valor de algún componente del array, el compilador nos dará el siguiente error: "**`assignment of read-only location`**", ya que se trata de un array constante que no puede cambiar de valor.

Por otro lado, ¿que ocurriría si el tamaño de la lista de valores para inicializar el array es distinta de la dimensión?

```
      ...
      const int MAXMESES = 12;
      typedef unsigned int Tdias [MAXMESES];
      ...
      const Tdias dias = {31,28,31,30,31,30,31,31,30,31,30,31,99};
      int indice;
      ...
```

En este caso, la lista de inicialización tiene más valores que la dimensión del array y el compilador nos dará el siguiente error: "**`excess elements in aggregate initializer`**"

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXMESES = 12;

// Zona de Declaración de Tipos
typedef unsigned int Tdias [MAXMESES];
```

```
// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      const Tdias dias = {31,28,31,30,31,30,31,31,30,31};
      unsigned int indice;

      // Zona de instrucciones
      for (indice = 0; indice < MAXMESES; indice ++)
            cout << "El mes " << indice + 1 << " tiene " << dias [indice]
                 << " dias." << endl;

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}
```

En este otro caso, la lista de valores es menor que la dimensión del array, sin embargo el compilador no nos dará ningún error y el programa se ejecutará, dando el siguiente resultado:

```
El mes 1 tiene 31 dias.
El mes 2 tiene 28 dias.
El mes 3 tiene 31 dias.
El mes 4 tiene 30 dias.
El mes 5 tiene 31 dias.
El mes 6 tiene 30 dias.
El mes 7 tiene 31 dias.
El mes 8 tiene 31 dias.
El mes 9 tiene 30 dias.
El mes 10 tiene 31 dias.
El mes 11 tiene 0 dias.
El mes 12 tiene 0 dias.
Presione una tecla para continuar . . .
```

Observamos que C++ inicializa por defecto a cero los valores restantes del array.

Veamos como lo hace para distintos tipos de datos, y en este caso sin usar el modificador "***const***":

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXARRAY = 5;

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      unsigned int i;

      int pru [MAXARRAY] = {1,2};
      double pru1 [MAXARRAY] = {1.0,2.0};
      bool pru2 [MAXARRAY] = {true, false};
      char pru3 [MAXARRAY] = {'a', 'b'};

      // Zona de instrucciones
      for (i = 0; i < MAXARRAY; i ++)
```

```
            cout << i << ".- pru " << pru [i] << " pru1 " << pru1 [i]
                 << " pru2 " << pru2 [i] << " pru3 " << pru3 [i] << endl;

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}
```

La salida en pantalla es la siguiente:

```
0.- pru 1 pru1 1 pru2 1 pru3 a
1.- pru 2 pru1 2 pru2 0 pru3 b
2.- pru 0 pru1 0 pru2 0 pru3
3.- pru 0 pru1 0 pru2 0 pru3
4.- pru 0 pru1 0 pru2 0 pru3
Presione una tecla para continuar . . .
```

En conclusión, en C++, con el operador de inicialización, las componentes de un array sin valor se inicializan por defecto.

Por último, insistir en que al no verificar C++ los índices fuera de rango hay que extremar las precauciones a la hora de manipularlos y especialmente cuando se realizan asignaciones.

Ejemplo:

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXARRAY = 3;

// Zona de Declaración de Tipos
typedef unsigned int Tarray [MAXARRAY];

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      Tarray a1, a2;
      unsigned int i;

      // Zona de instrucciones
      for (i = 0; i <= MAXARRAY + 1; i ++)      // Error: i < MAXRRAY
      {
            a1 [i] = i * 10;
            a2 [i] = i * 10;
      }
      for (i = 0; i < MAXARRAY; i ++)
            cout << i << " : " << a1 [i] << " " << a2 [i] << endl;

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}
```

Obsérvese el error cometido en la condición de control del bucle "***for***", en el que se llega a dar los valores `3` y `4` a la variable de condición del bucle `i`, y se ejecuta el cuerpo del bucle asignando valores a los componentes inexistentes `a1[3]` , `a2[3]` , `a1[4]` y `a2[4]`.

El compilador no detectará ningún error y ejecutará el programa. El resultado de la ejecución del mismo usando el compilador Dev C++ es el siguiente:

```
0 : 40 0
1 : 10 10
2 : 20 20
Presione una tecla para continuar . . .
```

Obsérvese como "`a1 [0]`" que se inicializó con el valor 0 tiene el valor 40, ello se debe a que dicho valor ha sido "*machacado*" por alguna de las asignaciones "`a1 [4] = 4 * 10`" o "`a2 [4] = 4 * 10`", produciendo además un valor extraño.

En otra máquina, otro compilador u otro entorno el efecto producido puede ser diferente.

5.5.3.- Igualdad.

Vamos a intentar verificar la igualdad de dos arrays completos de forma directa mediante el siguiente ejemplo.

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXARRAY = 3;

// Zona de Declaración de Tipos
typedef int Tarray [MAXARRAY];

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      Tarray a, b;

      // Zona de instrucciones
      a [0] = 3;              // Da valor al array a
      a [1] = 2;
      a [2] = 1;

      b [0] = 3;              // Da el mismo valor al array b
      b [1] = 2;
      b [2] = 1;

      if (a == b)             // Verifica la igualdad
            cout << "Iguales" << endl;
      else cout << "Distintos" << endl;

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
```

```
}
```

El programa se compila correctamente y el resultado de la ejecución de este programa es el siguiente:

```
Distintos
Presione una tecla para continuar . . .
```

Como vemos, aunque ambos arrays son del mismo tipo, tamaño y tienen el mismo contenido el resultado de la comparación aparentemente es erróneo.

Ello es debido a que en el lenguaje C++ hay una relación muy estrecha entre arrays y direcciones de memoria, que podríamos resumir diciendo que el nombre de una variable de array denota la dirección de memoria de su primer elemento.

```
      int a [10];  =>  a == &a [0]
```

Donde **&** es un operador unario que devuelve la dirección en memoria en la que se encuentra la variable que es su operando. En temas posteriores se verá en detalle.

Y claro, las direcciones de memoria en donde están situados los dos arrays del ejemplo son distintas. Por eso la verificación de igualdad no se cumple, porque en C++ se comparan las direcciones de memoria en las que se encuentran los arrays, no sus contenidos.

La única forma para poder entonces verificar la igualdad de dos arrays es mediante una estructura iterativa en la que se comparen todos y cada uno de los elementos de ambos arrays que tengan el mismo valor de índice. Y lo mejor es implementar dicha estructura iterativa en una función lógica.

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXARRAY = 3;

// Zona de Declaración de Tipos
typedef int Tarray [MAXARRAY];

// Zona de Cabeceras de Procedimientos y Funciones
bool iguales (Tarray a, Tarray b);

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      Tarray a, b;

      // Zona de instrucciones
      a [0] = 3;              // Da valor al array a
      a [1] = 2;
      a [2] = 1;

      b [0] = 3;              // Da el mismo valor al array b
      b [1] = 2;
      b [2] = 1;
```

```
	if (iguales (a, b))	// Verifica la igualdad
		cout << "Iguales" << endl;
	else cout << "Distintos" << endl;

	system("pause");	// Espera que se pulse una tecla para continuar

	a [0] = 1;		// Cambia el valor del array a
	a [1] = 2;
	a [2] = 3;

	if (iguales (a, b))	// Verifica la igualdad
		cout << "Iguales" << endl;
	else cout << "Distintos" << endl;

	system ("Pause");	// Hacer una pausa
	return 0;		// Valor de retorno al S.O.
}

// Implementación de Procedimientos y Funciones

bool iguales (Tarray a, Tarray b)
{
	// Zona de Declaración de VARIABLES
	unsigned int i;		// Para seleccionar los elementos de los arrays
	bool resultado;		// Testigo, resultado de la verificación

	// Zona de instrucciones
	resultado = true;	// Comprueba si los dos arrays son iguales
	i = 0;
	while ((i < MAXARRAY) && resultado)
	{
		resultado = a [i] == b [i];
		i ++;
	}
	return resultado;
}
```

El resultado de la ejecución del programa anterior como se esperaba es:

```
Iguales
Presione una tecla para continuar . . .
Distintos
Presione una tecla para continuar . . .
```

5.5.4.- Lectura y escritura.

Como ocurre en la mayoría de los lenguajes de programación respecto de los tipos estructurados, en C++ no está permitida la lectura y escritura directa de arrays sobre los dispositivos de E/S estándares (flujos `cin` y `cout`). Este tipo de operaciones debe realizarse elemento a elemento y ayudándose de estructuras repetitivas.

Dadas las siguientes definiciones:

```
const int MAXIMO = 100;
typedef int Tvector [MAXIMO];

Tvector x;
```

```
int i;
```

Leer un array.

```
      for (i = 0; i < MAXIMO; i ++)
            cin >> x [i];
```

Escribir un array:

```
      for (i = 0; i < MAXIMO; i ++)
            cout << x [i] << endl;
```

5.5.5.- Recorrido de arrays.

Se puede acceder a los elementos de un vector para introducir datos (leer) en él, para visualizar su contenido (escribir), para verificar su igualdad o para realizar un tratamiento sobre sus componentes. A la operación de efectuar una acción general sobre todos los elementos de un vector se la denomina recorrido del vector. Estas operaciones se realizan utilizando estructuras repetitivas, cuyas variables de control (por ejemplo, `posicion`) se utilizan como índices del array (por ejemplo, `nuevoArray [posicion]`). El incremento del contador del bucle producirá el tratamiento sucesivo de los elementos del vector.

Ejemplo: Procesar el array `puntos` de 30 enteros, realizando las siguientes operaciones:

a) Lectura del array.
b) Cálculo de la suma de los valores del array.
c) Cálculo de la media de los valores.

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXPUNTOS = 30;

// Zona de Declaración de Tipos
typedef int Tpuntos [MAXPUNTOS];

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      Tpuntos puntos;
      unsigned int posicion;
      int suma, media;

      // Zona de instrucciones
      suma = 0;
      for (posicion = 0; posicion < MAXPUNTOS; posicion ++)
      {
            cin >> puntos [posicion];     // Lectura del array
            suma += puntos [posicion];    // Acumula en suma
      }
      media = suma / int (MAXPUNTOS);     // Calcula la media
      cout << "La media es = " << media << endl;
```

```
      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}
```

5.6.- Arrays y subprogramas.

Ya hemos comentado que los elementos individuales de un array pueden usarse en cualquier lugar en donde se pudiera usar una variable del mismo tipo base.

Por ello podremos usar un elemento de un array como parámetro real en la llamada a un subprograma, tanto si el parámetro es pasado por valor como por referencia. Aunque lógicamente **no podremos usar un componente de un array como parámetro formal**. De igual forma podremos usar un elemento de un array para recoger el resultado de una función, siempre que el tipo base del array coincida o sea compatible con el de la función.

Veamos a continuación el paso completo de arrays con subprogramas.

5.6.1.- Arrays como parámetros.

En C++, para poder usar un parámetro de tipo array en una llamada a un subprograma debe haber un identificador que designe al tipo array, y el parámetro real debe ser del mismo tipo array que el parámetro formal.

Por ejemplo, si la función "`imprimir`" ha de recibir un array de 10 enteros, el aspecto del programa que la contiene sería así:

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXARRAY = 10;

// Zona de Declaración de Tipos
typedef int Tarray [MAXARRAY];

// Zona de Cabeceras de Procedimientos y Funciones
void imprimir (Tarray m);

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      Tarray a;
      unsigned int i;

      // Zona de instrucciones
      for (i = 0; i < MAXARRAY + 1; i ++)    // Inicializa el array
            a [i] = i * 10 + i;
      imprimir (a);           // Llamada a subprograma

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
```

```
}

// Implementación de Procedimientos y Funciones

void imprimir (Tarray m)      // Imprime el contenido del array
{
      // Zona de Declaración de VARIABLES
      unsigned int j;

      // Zona de instrucciones
      for (j = 0; j < MAXARRAY; j ++)
            cout << "El elemento " << j << " tiene el valor : "
                  << m [j] <<  endl;
}
```

El resultado de la ejecución de este programa sería:

```
El elemento 0 tiene el valor : 0
El elemento 1 tiene el valor : 11
El elemento 2 tiene el valor : 22
El elemento 3 tiene el valor : 33
El elemento 4 tiene el valor : 44
El elemento 5 tiene el valor : 55
El elemento 6 tiene el valor : 66
El elemento 7 tiene el valor : 77
El elemento 8 tiene el valor : 88
El elemento 9 tiene el valor : 99
Presione una tecla para continuar . . .
```

Sin embargo, aunque en el ejemplo anterior no usemos el cualificador "**&**" de paso de parámetros por referencia en la definición de los parámetros formales, éste es el mecanismo que sigue C++ a la hora de pasar arrays como parámetros. Veámoslo en el siguiente ejemplo:

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXARRAY = 3;

// Zona de Declaración de Tipos
typedef int Tarray [MAXARRAY];

// Zona de Cabeceras de Procedimientos y Funciones
void cambiar (Tarray x);
void imprimir (Tarray m);

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      Tarray a = {1, 2, 3};

      // Zona de instrucciones
      cout << "Antes de llamar a cambiar" << endl;
      imprimir (a);
      cambiar (a);
      cout << "Despues de llamar a cambiar" << endl;
```

```
      imprimir (a);

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}

// Implementación de Procedimientos y Funciones

void cambiar (Tarray x)
{
      // Zona de Declaración de VARIABLES
      unsigned int j;

      // Zona de instrucciones
      for (j = 0; j < MAXARRAY; j ++)
            x [j] *= 10;
}

void imprimir (Tarray m)      // Imprime el contenido del array
{
      // Zona de Declaración de VARIABLES
      unsigned int j;

      // Zona de instrucciones
      for (j = 0; j < MAXARRAY; j ++)
            cout << "El elemento " << j << " tiene el valor : "
                  << m [j] <<  endl;
}
```

El resultado de la ejecución del programa anterior es:

```
Antes de llamar a cambiar
El elemento 0 tiene el valor : 1
El elemento 1 tiene el valor : 2
El elemento 2 tiene el valor : 3
Despues de llamar a cambiar
El elemento 0 tiene el valor : 10
El elemento 1 tiene el valor : 20
El elemento 2 tiene el valor : 30
Presione una tecla para continuar . . .
```

Como vemos, el subprograma ha modificado los parámetros reales que se le han pasado. Ello es debido a la relación existente entre los arrays y las direcciones de memoria que ya se comentó anteriormente.

Este hecho repercute en el paso de parámetros cuando estos son arrays.

En efecto, en C++ no se puede pasar un array por valor, ya que C++ trasmite siempre por referencia las variables de array, y al pasar su nombre no realiza una copia del valor sino lo que hace realmente es pasar su dirección (una referencia al parámetro real).

Así, en la instrucción que invoca a una función, se escribe siempre el nombre de la variable sin más y en la cabecera de la función invocada se indica el tipo y nombre del correspondiente parámetro formal.

Por ello es necesario extremar las precauciones a la hora de trabajar con arrays y subprogramas.

Finalmente podemos generalizar aun más el diseño de subprogramas con arrays si en vez de utilizar la constante global que indica la dimensión del array dentro del subprograma usamos el operador ***sizeof***.

Sintaxis:

sizeof (nombre_tipo)

En BNF:

<operador_sizeof> ::= sizeof (<nombre_tipo>)

En donde `nombre_tipo` es el nombre de un tipo predefinido o un tipo definido mediante la sentencia ***typedef***, sea simple o estructurado.

Este operador devuelve el **número de bytes** que ocupa una variable del tipo que se le indica en un compilador y maquina concreto.

Ejemplo:

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXARRAY = 5;

// Zona de Declaración de Tipos
typedef int Tentero;
typedef int Tarray [MAXARRAY];

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal

      // Zona de instrucciones
      cout << "char         : " << sizeof (char) << endl;
      cout << "bool         : " << sizeof (bool) << endl;
      cout << "int          : " << sizeof (int) << endl;
      cout << "Tentero      : " << sizeof (Tentero) << endl;
      cout << "short int    : " << sizeof (short int) << endl;
      cout << "long int     : " << sizeof (long int) << endl;
      cout << "unsigned int : " << sizeof (long int) << endl;
      cout << "float        : " << sizeof (float) << endl;
      cout << "double       : " << sizeof (double) << endl;
      cout << "Tarray       : " << sizeof (Tarray) << endl;

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}
```

La ejecución de este programa en Dev C++ nos da el siguiente resultado:

```
char         : 1
bool         : 1
int          : 4
```

```
Tentero      : 4
short int    : 2
long int     : 4
unsigned int : 4
float        : 4
double       : 8
Tarray       : 20
Presione cualquier tecla para continuar . . .
```

La utilidad de este operador a la hora de diseñar subprogramas con arrays es que nos evita el tener que usar el número o constante que define la dimensión del array, pudiendo calcularlo dinámicamente; aunque realmente este cálculo es resuelto en tiempo de compilación y por tanto no sobrecarga la ejecución del subprograma. Esto permite construir subprogramas más genéricos.

La forma de usarlo es simplemente sustituir la constante que indica la dimensión del array por la siguiente expresión constante:

sizeof (tipo_array) / sizeof (tipo_base)

Como *sizeof*(tipo_array) nos da como resultado el número de bytes que ocupa una variable del tipo_array, si dividimos ese valor por el número de bytes que ocupa cada componente (los cuales son de tipo_base) obtendremos la dimensión del array.

Veamos el ejemplo anterior de uso de subprogramas aplicando los operadores ***sizeof*** dentro de los mismos.

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXARRAY = 3;

// Zona de Declaración de Tipos
typedef int Tarray [MAXARRAY];

// Zona de Cabeceras de Procedimientos y Funciones
void cambiar (Tarray x);
void imprimir (Tarray m);

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      Tarray a = {1, 2, 3};

      // Zona de instrucciones
      cout << "Antes de llamar a cambiar" << endl;
      imprimir (a);

      cambiar (a);

      cout << "Despues de llamar a cambiar" << endl;
      imprimir (a);

      system ("Pause");   // Hacer una pausa
```

```
      return 0;           // Valor de retorno al S.O.
}

// Implementación de Procedimientos y Funciones

void cambiar (Tarray x)
{
      // Zona de Declaración de VARIABLES
      unsigned int j;

      // Zona de instrucciones
      for (j = 0; j < sizeof (Tarray) / sizeof (int); j ++)
            x [j] *= 10;
}

void imprimir (Tarray m)
{
      // Zona de Declaración de VARIABLES
      unsigned int j;

      // Zona de instrucciones
      for (j = 0; j < sizeof (Tarray) / sizeof (int); j ++)
            cout << "El elemento " << j << " tiene el valor : "
                  << m [j] <<  endl;
}
```

5.6.2.- Funciones y arrays.

Al igual que podemos pasar arrays como parámetros, también podríamos pensar en usar tipos array como resultado de las funciones.

Por ejemplo, el siguiente programa:

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXIMO = 10;

// Zona de Declaración de Tipos
typedef int Tvector [MAXIMO];

// Zona de Cabeceras de Procedimientos y Funciones
Tvector leeVector ();
void escribeVector (Tvector x);

// Programa Principal
int main()
{
      // Zona de Declaración de Variables del Programa principal
      Tvector vector;

      // Zona de instrucciones
      vector = leeVector ();
      escribeVector (vector);
```

```
      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}

// Implementación de Procedimientos y Funciones

Tvector leeVector ()
{
      // Zona de Declaración de VARIABLES
      Tvector x;
      unsigned int i;

      // Zona de instrucciones
      for (i = 0; i < sizeof (Tvector) / sizeof (int); i ++)
      {
            cout << "Introduzca x [" << i << "] = ";
            cin >> x [i];
      }
      return x;
}

void escribeVector (Tvector x)
{
      // Zona de Declaración de VARIABLES
      unsigned int i;

      // Zona de instrucciones
      for (i = 0; i < sizeof (Tvector) / sizeof (int); i ++)
            cout << "x [" << i << "] = " << x [i] << endl;
}
```

Al compilarlo se produce el siguiente error de compilación: "**'leeVector' declared as function returning an arr**ay".

Claramente C++ no permite devolver el valor de un array.

Por tanto **no se pueden usar arrays como resultado de funciones**.

Si necesitamos devolver un array, usaremos entonces un procedimiento con un parámetro pasado por referencia para el array (de hecho como vimos anteriormente los parámetros de array siempre se pasan por referencia), con lo que el programa anterior deberíamos escribirlo de la siguiente manera:

```
// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXIMO = 10;

// Zona de Declaración de Tipos
typedef int Tvector [MAXIMO];

// Zona de Cabeceras de Procedimientos y Funciones
void leeVector (Tvector &x);
void escribeVector (Tvector x);

// Programa Principal
```

```
int main()
{
      // Zona de Declaración de Variables del Programa principal
      Tvector vector;

      // Zona de instrucciones
      leeVector (vector);
      escribeVector (vector);

      system ("Pause");   // Hacer una pausa
      return 0;           // Valor de retorno al S.O.
}

// Implementación de Procedimientos y Funciones

void leeVector (Tvector &x)
{
      // Zona de Declaración de VARIABLES
      unsigned int i;

      // Zona de instrucciones
      for (i = 0; i < sizeof (Tvector) / sizeof (int); i ++)
      {
            cout << "Introduzca x [" << i << "] = ";
            cin >> x [i];
      }
}

void escribeVector (Tvector x)
{
      // Zona de Declaración de VARIABLES
      unsigned int i;

      // Zona de instrucciones
      for (i = 0; i < sizeof (Tvector) / sizeof (int); i ++)
            cout << "x [" << i << "] = " << x [i] << endl;
}
```

5.7.- Ejemplos.

Para terminar veamos algún ejemplo de programas completos sobre el manejo de arrays.

Un palíndromo es una frase que (atendiendo sólo a sus letras e ignorando los espacios, acentos, signos de puntuación y tipo de letra, mayúscula o minúscula) expresa lo mismo leída de izquierda a derecha que de derecha a izquierda. Por ejemplo: "dabale arroz a la zorra el abad"

Se necesita un programa que lea una frase de no más de 100 letras, terminada en un punto, y que determine si es o no un palíndromo. Para ello usaremos un array con tipo base **char**.

```
/*-------------------------------------------------------------------
|   Autor:                                                          |
|   Fecha:                          Versión: 1.0                    |
|-------------------------------------------------------------------|
|                                                                   |
|  Programa ejemplo para la determinación de si una frase es un     |
|  palíndromo.                                                      |
|                                                                   |
| ------------------------------------------------------------------*/

// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
using namespace std;

// Zona de Declaración de Constantes
const char TERMINADOR = '.';
const unsigned int MAXIMO = 100;

// Zona de Declaración de Tipos
typedef char Tfrase [MAXIMO];

// Zona de Cabeceras de Procedimientos y Funciones
char aMayuscula (char c);

// Programa Principal
int main()
{
    // Zona de Declaración de Variables del Programa principal
    Tfrase frase;       // Array de caracteres para contener la frase
    unsigned int i, j;  // Para recorrer de izqda-dcha y de dcha-izqda
    char car;           // Para recoger la frase carácter a carácter

    // Zona de instrucciones
    cout << "Escriba la frase, terminada por " << TERMINADOR << " : ";

    i = 0;
    do
    {
        cin >> car;

        // Se filtran todos los caracteres recogidos que no sean letras
        if (((car>='A') && (car<='Z')) || ((car>='a') && (car<='z')))
        {
            frase [i] = aMayuscula (car);
            i ++;
        }
```

```
    } while ((car != TERMINADOR) && (i < MAXIMO));

    i --;           // Para volver al carácter anterior al terminador
    j = 0;          // Compara caracteres desde ambos extremos del array
    while ((j < i) && (frase [j] == frase [i]))
    {
        j ++;
        i --;
    }

    if (j >= i)         // Si los recorridos se cruzan es palíndromo
        cout << "Es palindromo" << endl;
    else cout << "No es palindromo" << endl;

    system("Pause");    // Hacer una pausa
    return 0;           // Valor de retorno al S.O.
}

// Implementación de Procedimientos y Funciones

//-----------------------------------------------------------------------
//  aMayuscula (c)
//
//  Parámetros: c   ->  char (valor)
//  Resultado:  char
//
//  Devuelve el carácter que se le pasa como parámetro en mayúsculas.
//-----------------------------------------------------------------------
char aMayuscula (char c)
{
    // Zona de Declaración de VARIABLES

    // Zona de instrucciones
    if ((c >= 'a') && (c <= 'z'))       // Sólo si es letra minúscula
        return char (int (c) - int('a') + int('A'));
    else return c;
}
```

Obsérvese la utilización de los operadores de conversión de tipo (casting) para calcular la letra mayúscula correspondiente a una minúscula dentro de la función `aMayuscula`. La expresión "`char (int (c) - int('a') + int('A'))`" es equivalente a la expresión en pseudolenguaje "`CHR (ORD (c) - ORD ('a') + ORD ('A'))`".

Veamos otro ejemplo.

Realizar un programa en C++ que procese mediante subprogramas un array de hasta 100 números reales realizando las siguientes tareas:

a) Lectura del array.
b) Impresión en pantalla del array introducido.
c) Determinación del número menor del array.
d) Cálculo de la suma de los elementos del array.
e) Cálculo de la media de los valores del array.
f) Cálculo de la varianza de los valores del array.
g) Cálculo de la desviación típica de los valores del array.

```
/*-----------------------------------------------------------------
```

```
|   Autor:                                                          |
|   Fecha:                          Versión: 1.0                    |
|-------------------------------------------------------------------|
|   Programa ejemplo de procesamiento de un vector de reales        |
| ------------------------------------------------------------------*/

// Incluir E/S y Librerías Standard
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;

// Zona de Declaración de Constantes
const unsigned int MAXIMO = 100;

// Zona de Declaración de Tipos
typedef double Tvector [MAXIMO];

// Zona de Cabeceras de Procedimientos y Funciones
void leeVector (Tvector &x, unsigned int &tam);
void escribeVector (Tvector x, unsigned int tam);
double menorVector (Tvector x, unsigned int tam);
double sumaVector (Tvector x, unsigned int tam);
double mediaVector (Tvector x, unsigned int tam);
double varianzaVector (Tvector x, unsigned int tam);
double desviacionVector (Tvector x, unsigned int tam);

// Programa Principal
int main()
{
    // Zona de Declaración de Variables del Programa principal
    Tvector vector;
    unsigned int tamano;      // Número de elementos válidos dentro del array

    // Zona de instrucciones
    cout << "Introduzca las componentes del vector" << endl;
    leeVector (vector, tamano);

    cout << "Vector introducido" << endl;
    escribeVector (vector, tamano);

    cout << "Valor menor = " << menorVector (vector, tamano) << endl;
    cout << "Sumatorio = " << sumaVector (vector, tamano) << endl;
    cout << "Media = " << mediaVector (vector, tamano) << endl;
    cout << "Varianza = " << varianzaVector (vector, tamano) << endl;
    cout << "Desviacion = " << desviacionVector (vector, tamano) << endl;

    system("Pause");    // Hacer una pausa
    return 0;           // Valor de retorno al S.O.
}

// Implementación de Procedimientos y Funciones

//--------------------------------------------------------------------
//  leevector (A, t)
//
//  Parámetros: A   ->  Tvector (referencia)
//              t   ->  unsigned int (referencia)
//  Resultado:  no tiene
```

```
//
//  Lee de teclado el tamaño y los componentes de un array del tipo
//  Tvector y lo devuelve como resultado a través de un parámetro
//  por referencia. También devuelve el tamaño leído en el parámetro t.
//-----------------------------------------------------------------------
void leeVector (Tvector &x, unsigned int &tam)
{
    // Zona de Declaración de VARIABLES
    unsigned int i;

    // Zona de instrucciones
    cout << "Introduzca el tamano del vector a procesar (maximo "
         << sizeof (Tvector) / sizeof (double) << "): ";
    cin >> tam;

    if (tam < 0)                // Comprueba y en su caso ajusta el tamaño
        tam = 0;
    else if (tam > sizeof (Tvector) / sizeof (double))
        tam = sizeof (Tvector) / sizeof (double);

    for (i = 0; i < tam; i ++)  // Lee los componentes del vector
    {
        cout << "Introduzca x [" << i << "] = ";
        cin >> x [i];
    }
}

//-----------------------------------------------------------------------
//  escribeVector (A, t)
//
//  Parámetros: A   ->  Tvector (valor)
//              t   ->  unsigned int (valor)
//  Resultado:  no tiene
//
//  Imprime en pantalla el contenido del array A con tamaño t
//  elemento a elemento. Es decir imprime hasta el componente t-1.
//-----------------------------------------------------------------------
void escribeVector (Tvector x, unsigned int tam)
{
    // Zona de Declaración de VARIABLES
    unsigned int i;

    // Zona de instrucciones
    for (i = 0; i < tam; i ++)
        cout << "x [" << i << "] = " << x [i] << endl;
}

//-----------------------------------------------------------------------
//  menorVector (A, t)
//
//  Parámetros: A   ->  Tvector (valor)
//              t   ->  unsigned int (valor)
//  Resultado:  double
//
//  Devuelve el valor menor de los componentes del array A con tamaño t.
//-----------------------------------------------------------------------
double menorVector (Tvector x, unsigned int tam)
{
    // Zona de Declaración de VARIABLES
```

```
    double menor;
    unsigned int i;

    // Zona de instrucciones
    menor = x [0];
    for (i = 1; i < tam; i ++)
        menor = (menor > x [i]) ? x [i] : menor;

    return menor;
}

//---------------------------------------------------------------------
//  sumaVector (A, t)
//
//  Parámetros: A   ->  Tvector (valor)
//              t   ->  unsigned int (valor)
//  Resultado:  double
//
//  Devuelve la suma de los valores de las componentes del array A
//  con tamaño t.
//---------------------------------------------------------------------
double sumaVector (Tvector x, unsigned int tam)
{
    // Zona de Declaración de VARIABLES
    double suma;
    unsigned int i;

    // Zona de instrucciones
    suma = 0.0;
    for (i = 0; i < tam; i ++)
        suma += x [i];

    return suma;
}

//---------------------------------------------------------------------
//  mediaVector (A, t)
//
//  Parámetros: A   ->  Tvector (valor)
//              t   ->  unsigned int (valor)
//  Resultado:  double
//
//  Calcula y devuelve la media de los valores de las componentes del
//  array A con tamaño t.
//---------------------------------------------------------------------
double mediaVector (Tvector x, unsigned int tam)
{
    // Zona de instrucciones
    return sumaVector (x, tam) / double (tam);
}

//---------------------------------------------------------------------
//  varianzaVector (A, t)
//
//  Parámetros: A   ->  Tvector (valor)
//              t   ->  unsigned int (valor)
//  Resultado:  double
//
//  Calcula y devuelve la varianza de los valores de las componentes
```

```
//  del array A con tamaño t.
//-----------------------------------------------------------------------
double varianzaVector (Tvector x, unsigned int tam)
{
    // Zona de Declaración de VARIABLES
    double suma, media;
    unsigned int i;

    // Zona de instrucciones
    suma = 0.0;
    media = mediaVector (x, tam);
    for (i = 0; i < tam; i ++)
        suma += pow (x [i] - media, 2.0);

    return suma / tam;
}

//-----------------------------------------------------------------------
//  desviacionVector (A, t)
//
//  Parámetros: A   ->  Tvector (valor)
//              t   ->  unsigned int (valor)
//  Resultado:  double
//
//  Calcula y devuelve la desviación típica de los valores de las
//  componentes del array A con tamaño t.
//-----------------------------------------------------------------------
double desviacionVector (Tvector x, unsigned int tam)
{
    // Zona de instrucciones
    return sqrt ((varianzaVector (x,tam) * double (tam)) / double (tam-1));
}
```

Ejercicios.

1.- Diseña un programa C++ que solicite las componentes de dos vectores geométricos por teclado y calcule su producto escalar y su producto vectorial.

2.- Escribe un programa C++ que lea una sucesión de 10 números naturales, encuentre el valor máximo y lo imprima junto con el número de veces que aparece, y las posiciones en que esto ocurre. El proceso se repite con el resto de la sucesión hasta que no quede ningún elemento por tratar.

Ejemplo de entrada: `7 10 143 10 52 143 72 10 143 7`
Salida generada:
```
143 aparece 3 veces, en posiciones 3 6 9
...
7 aparece 2 veces, en posiciones 1 10
```

3.- Diseña un programa C++ para realizar la conversión de números naturales en base decimal entre 0 y 65535 a base hexadecimal. Para ello el algoritmo tendrá como:

- Datos de entrada: Un número entero positivo entre 0 y 65535 cualquiera dado por el usuario. El programa deberá verificar que el número entrado cumple esas condiciones.
- Datos de salida: Impresión en pantalla de un array de caracteres que contenga el equivalente en base hexadecimal del número entrado. Este array deberá estar formado por un máximo de cuatro elementos (dígitos) que pueden ser cifras entre 0 y 9 y letras entre A y F.

4.- Diseña un programa C++ que lea por teclado las coordenadas en el plano de dos vectores geométricos y calcule el ángulo en grados que forman esos dos vectores.

5.- Dadas las siguientes definiciones de tipo:

```
typedef char Tvector [N];
typedef char Tvectorg [2 * N];
```

a) Diseña una función que, dados dos vectores ordenados del tipo `Tvector` como parámetros, devuelva otro vector ordenado de tipo `Tvectorg` que sea la mezcla de ambos.

b) Diseña una función lógica que dados dos vectores del tipo `Tvector`, devuelva **`true`** si son iguales y **`false`** en otro caso. Para este caso supondremos que dos vectores son iguales si contienen los mismos elementos y en el mismo orden relativo, suponiendo que el primer elemento sigue al último. Por ejemplo, si la entrada fuera:

```
['A', 'C', 'D', 'F', 'E']
['D', 'F', 'E', 'A', 'C']
```

la función devolvería **`true`**.
Supón, además, que cada carácter aparece a lo sumo una vez.

c) La moda de un array de caracteres es el carácter del array que se repite más frecuentemente. Si varios caracteres se repiten con la misma frecuencia máxima, entonces

no hay moda. Escribe un procedimiento que acepte un vector de tipo `Tvector` y devuelva la moda, o una indicación de que la moda no existe.

6.- Para operar con números naturales de tamaño grande se puede utilizar un array cuyas componentes sean dígitos decimales (entre 0 y 9, ambos inclusive). Elaborar un programa en C++ para sumar números naturales de hasta 40 cifras por el procedimiento anterior. Ejemplo:

```
Número 1:  394774123
Número 2:  46957
Resultado: 394821080
```

Los números a sumar no contienen decimales.

7.- Dado un polinomio de grado n, $p(x) = p_n x^n + p_{n-1} x^{n-1} + ... + p_1 x + p_0$, podemos representarlo mediante un array de sus coeficientes (p_n, p_{n-1}, ..., p_1, p_0).

Diseña un programa C++ que lea dos polinomios, los almacene en sendos arrays y calcule la suma y el producto de ambos.

8.- Dado un array `f` de enteros con dimensión N, diremos que un elemento es redundante si su valor es igual al del elemento anterior del array.

a) Diseña un programa C++ que escriba la secuencia de elementos de `f` eliminando los redundantes, sin emplear arrays auxiliares.
b) Diseña otro programa C++ que haga lo mismo que el anterior, pero dejando los elementos redundantes sobre el array `f`.

9.- En una entidad bancaria el número de cuenta de un cliente se va a formar añadiendo a una determinada cifra un dígito de autoverificación. Dicho dígito de autoverificación se calculará de la siguiente forma:

a) Multiplicar la posición de las unidades y cada posición alternada por dos.
b) Sumar los dígitos no multiplicados y los resultados de los productos obtenidos en el apartado a).
c) Restar el número obtenido en el apartado b) del número más próximo y superior a éste, que termine en cero.

El resultado será el dígito de autoverificación.

Codifica un programa que vaya aceptando números de cuenta y compruebe mediante el dígito de autoverificación si el número introducido es correcto o no. El proceso se repetirá hasta que se introduzca un cero como número de cuenta.