Objetivos de la práctica:
- Declaración y utilización del tipo de dato estructurado array.
- Aprender a utilizar el tipo string y las funciones definidas sobre él.
- Declarar y utilizar el tipo de dato estructurado registro (o *struct*)

Datos estructurados

Los datos de tipo estructurado o tipo compuesto son agrupaciones de otros tipos de datos.
Atendiendo a cómo se almacenan los datos en la memoria, podemos hacer una primera clasificación de los datos de tipo estructurado en:

- Estructuras contiguas - que se almacenan en la memoria de forma consecutiva.
- Estructuras enlazadas - que se almacenan de forma salteada.

Y dependiendo de sí su tamaño permanece fijo o no durante la ejecución del programa, podemos hacer una segunda clasificación en:

- Estructuras estáticas - su tamaño permanece fijo a lo largo de la ejecución del programa.
- Estructuras dinámicas - su tamaño puede alterarse durante la ejecución del programa.

Normalmente las estructuras enlazadas son siempre dinámicas.

Estructuras contiguas en C/C++: arrays, strings y registros.

Los vectores y matrices (*arrays*), las cadenas de caracteres (*strings*) y los registros son estructuras formadas por una colección de datos, englobados bajo un mismo identificador (nombre) y que se almacenan en memoria en posiciones adyacentes. Además de ser un nuevo tipo de datos, son a su vez una colección de datos más simples, a los cuales se puede acceder de forma independiente.
Podemos hacer una clasificación de las estructuras contiguas en dos grupos dependiendo de sí la colección de datos que la forma son del mismo tipo (strings y arrays) o son de tipo diferente (registros).

Vectores y Matrices

Vectores

Sirven para agrupar variables de un mismo tipo con un único nombre.
Estas variables se almacenan en memoria de forma consecutiva y cada una de ellas constituye un elemento del vector.
Las operaciones permitidas sobre los elementos del vector (como p.e.: la asignación, la suma, etc.) son las mismas que las permitidas sobre los datos del tipo base.
Para identificar al vector le asignamos como siempre un nombre; y para identificar a cada uno de los elementos se le asigna, además, uno o varios índices entre corchetes. El número de índices define el número de dimensiones del vector.
Por ser estructuras estáticas, el tamaño del vector, así como su dimensión, queda fijada en la declaración y no se puede alterar durante la ejecución.

Sintaxis: `tipo nombre [ tamaño ];`
donde:
- `tipo`: es el tipo base del vector, es decir el tipo de cada uno de los elementos.
- `nombre`: es el identificador que asignamos al vector.
- `tamaño`: es el número de elementos del vector. Se fija en la declaración, para que el compilador reserve espacio suficiente en memoria para todos ellos. Este tamaño no se puede alterar durante la ejecución.

Cuando declaramos un vector, estamos declarando al mismo tiempo un conjunto de variables del tipo base del vector, que forman los elementos del vector.
El rango de los índices comienza siempre en el cero y acaba uno antes del tamaño del vector. Así, si el tamaño del vector es `N`, el primer elemento es el de índice `0` y el último el de índice `N-1`.

Cada uno de los elementos del vector es una variable independiente sobre la que se pueden hacer las mismas operaciones que sobre una variable simple del mismo tipo que el tipo base. La peculiaridad de las variables que representan los elementos del vector es que todas ellas se almacenan en posiciones de memoria adyacentes y que todas ellas se llaman igual, y lo único que las diferencia es un índice.
El nombre del vector es también una variable que contiene la dirección de comienzo del vector (la referencia a la variable).

- *Ejemplo:*

Si queremos declarar 10 variables de tipo entero, la única forma de hacerlo hasta ahora sería declararlos como variables indiciduales:

int a0, a1, a2, a3, a4, a5, a6, a7, a8, a9;

Y si quisiésemos inicializarlas a 0, habría que escribir 10 asignaciones.

Ahora podemos declararlas utilizando un vector, es decir:

int a[10];

IMPORTANTE: No se pueden hacer asignaciones entre vectores, es decir, la siguiente asignación no se debe realizar NUNCA:

```
int a[10] , b[10];
a = b ;    //ESTO ESTA MAL!!!!
```

Declaración de vectores con iniciación

Igual que con el resto de tipos simples, los vectores y en general todos los arrays, se pueden iniciar con valores al tiempo que son declarados. Para ello el conjunto de valores se encierran entre llaves separados por comas, siguiendo la siguiente sintaxis:

Sintaxis: `tipo nombre[tamaño] = { valor, valor, ...};`

- *Ejemplo:*

```
float b[4] = {1.3, -3.78, 1.3e10, 0.0025};
```

Cuando se inicia a la vez que se declara un vector, no es necesario fijar el tamaño. En ese caso, el compilador cuenta el número de elementos y reserva en memoria el espacio justo para todos ellos.

Matrices

Llamamos matrices a los arrays de dos dimensiones, es decir, a los arrays cuyos elementos vienen definidos por dos índices. Podemos imaginarnos su organización como una tabla o matriz, donde cada elemento tiene asignado 2 índices: El primero representa la fila y el segundo la columna.

Sintaxis: `tipo nombre_matriz[filas][columnas];`

donde:

`tipo`: es el tipo base de los elementos de la matriz.
`nombre_matriz`: es el identificador de la matriz.
`filas`: es el número de filas de la matriz.
`columnas`: es el número de columnas de la matriz.

Tanto el número de filas como el de columnas se fijan en la declaración y no se puede alterar en tiempo de ejecución.
El rango de las filas y el de las columnas empieza en el cero y acaba en uno menos el tamaño de las filas y en uno menos de las columnas, respectivamente.
Todos los elementos de una matriz se guardan en posiciones de memoria consecutivas, donde cada una de las filas se guardan una a continuación de la otra. Dentro de una misma fila todos sus elementos se guardan también consecutivamente.

Ejemplo: Visualización de los elementos de una matriz de enteros de tamaño TAM_X por TAM_Y

```
int i, j;
int matriz[TAM_X][TAM_Y];

for(i=0 ; i<TAM_X ; i++)
   for(j=0 ; j<TAM_Y ; j++)
       cout << matriz[i][j];
```

Declaración de matrices con iniciación.

Al igual que los vectores, las matrices pueden iniciarse al declararse colocando todas las filas entre llaves y separadas por comas, y dentro de cada fila todos sus elementos entre llaves y separados por comas.

- *Ejemplo:*

```
float g[3][2] = { { -1.2, 2.3 }, { -7, 8.23e13 }, { 89, -45.78 } };
```

Arrays multidimensionales

De igual manera, puedo declarar arrays de más dimensiones donde cada elemento tiene asignado varios índices.

Sintaxis: `tipo nombre [tam1][tam2]...[tamN];`

Como siempre, cada uno de los índices comienza con el valor `0` y por tanto acaban en el índice uno menos el tamaño de la dimensión.

El paso de arrays *como parámetros a las funciones*

Por motivos de eficiencia y de comodidad, cuando en C/C++ se pasa un vector o un array en general, a una función, sólo se le pasa la dirección de comienzo del vector y no una copia del vector entero, es decir, el paso de un vector como parámetro de una función es siempre un paso por referencia. Como todo paso por referencia, los vectores serán parámetros de entrada pero al mismo tiempo serán parámetros de salida.

En la llamada a la función, pasamos como argumento el nombre del vector.

El paso por referencia de vectores no hace uso del operador "**&**". Es más, aparentemente la declaración de los parámetros se hacen como si fuera un paso por valor.

Por último, una función no puede devolver un tipo array, se debe pasar como argumento.

- *Ejemplo: Función para leer los elementos de un vector por teclado*

```
void LeerVector( int vect[TAM])
{
   int i;
   for(i=0 ; i<TAM ; i++)
      cin >> vect[i];
}
```

Cadenas de caracteres (Strings)

En C++, la forma de usar variables que contengan una secuencia de caracteres es usando el tipo de datos *string*.

Sintaxis: `string nombre_de_la_variable;`

Al igual que otros tipos de datos simples, las variables de tipo string se pueden inicializar a la vez que se declaran.

- *Ejemplo: Declaración y asignación de un string*

```
string s;
string s2 = "Hola";
```

Funciones para el manejo de strings

Las cadenas de caracteres (strings) permiten la manipulación de textos. Generalmente, se considera que un string no es más que un array de caracteres. Por esa razón, se suele relacionar el concepto de string con el de array. En C++, existe el tipo (clase, en la nomenclatura de los lenguajes orientados a objetos) *string*. Para su uso es preciso utilizar **`#include <string>`**, no **`<string.h>`**.

Veamos las operaciones básicas definidas para *string*:

- Creación de variables:

```
string palabra, frase;
```

- Asignación: (Al contrario que con los vectores, si se pueden realizar asignaciones entre strings)

```
frase = palabra;
frase = "hola";
```

- Acceso a los caracteres (como arrays):

```
palabra[0]
```

- Comparación lexicográfica (==, !=, <, >):

```
frase == palabra
frase > palabra
```

- Lectura/escritura:

```
cin >> palabra;
getline (cin, frase); //lee frase de cin hasta encontrar el fin de línea
cout << frase << endl;
```

- Manipulación de textos (suponer **`unsigned int i;`**):

```
// nº de caracteres de palabra
i = palabra.length();

// inserta palabra en la posición 3 de frase
frase.insert(3, palabra);

// concatena (une) palabra y "hola" y almacena el  resultado en frase
frase = palabra + "hola";

// concatena (añade al final) palabra a frase
frase += palabra;

// borra 7 caracteres de frase desde la posición 3
frase.erase (3,7);

// sustituye (reemplaza) 6 caracteres de frase, empezando en la posición 1,
// por la cadena palabra
frase.replace (1, 6, palabra);

//busca palabra como una subcadena dentro de frase, devuelve la posición
//donde la encuentra
i = frase.find(palabra);

//devuelve la subcadena formado por 3 caracteres desde la posición 3 de
//frase
palabra = frase.substr(5,3);
```

El paso de strings como parámetros a las funciones

Se hará por referencia o por valor dependiendo de que queramos modificar o no la variable.

Registros (Estructuras)

Hasta el momento sólo hemos trabajado con estructuras de datos homogéneas, donde todos los datos simples eran del mismo tipo (arrays y strings). Pero dentro del C/C++ se puede definir también estructuras de datos heterogéneas, formadas por un conjunto de datos de tipos diferentes.
Los registros, llamados también estructuras, son un conjunto de datos de tipos diferentes, que se citan y manejan bajo un mismo nombre y que se guardan de forma consecutiva en memoria. Cada uno de los elementos que forman un registro se llama campo y, al igual que los elementos de un array, pueden utilizarse de forma individual. Por tanto, entendemos los registros o estructuras como una agrupación de variables diferentes, relacionadas de alguna forma lógica.

Antes de usar una estructura, hay que definirla

Una de las características de las estructuras es que hay que definirlas antes de ser usardas en la declaración de variables.

Sintaxis:

```
struct NombreEstructura
{
   Tipo1 NombreVariable1;
   Tipo2 NombreVariable2;
   ...
   TipoN NombreVariableN;
};
```

(Obsérvese que la definición acaba con un punto y coma. Esto se debe a que la definición de una estructura es en sí una sentencia).

En la definición no estamos declarando ni reservando memoria para ninguna variable, tan sólo estamos creando un nuevo tipo adaptado a nuestras necesidades. La declaración de variables que sean de este nuevo tipo vendrá a posteriori de la definición.
Sin embargo, a la vez que se define la estructura se admite la declaración de variables del tipo estructura definida, pero lo normal es declararlas fuera de la definición.
Normalmente una estructura se crea (se define) para que pueda utilizarse dentro de varias funciones, por lo que la definición de éstas se hace normalmente fuera de cualquier función, al principio del programa, antes de ser utilizada por ninguna de ellas. Sin embargo, nada impide definir una estructura dentro de una función; en este caso el ámbito de la estructura se limita exclusivamente a la función.

Ejemplo:

```
// información de interes sobre un empleado de una empresa
struct empleado
{
   string nombre;
   long int salario;    // salario anual en pesetas
   string num_telefono;
};
```

Sintaxis de la declaración de variables

Una vez definida la nueva estructura, ya puedo declarar variables de ese tipo de la siguiente manera:

Sintaxis: `struct NombreEstructura NomVarEstruct1, ..., NomVarEstructN;`

Utilización de los elementos individuales (los campos) de la estructura

Una vez declarada una variable como estructura, se puede acceder a cada uno de sus campos especificando el nombre de la variable y el identificador del campo separados ambos por un punto.

- *Ejemplo:*

```
cout << "Nombre " << pepe.nombre << endl;
juan.saldo = 2000;
```

Como hemos visto arriba, puedo tratar cada uno de los campos de una estructura como variables diferentes, es decir, puedo asignar valores a uno de los campos sin tocar el resto campos (`juan.saldo = 2000;`) o puedo acceder a uno de los campos sin necesidad de acceder al resto de los valores de la estructura (`cout << "Nombre " << pepe.nombre << endl;`)

A diferencia de los array, está permitida la operación de asignación entre estructuras, con lo que puedo hacer una copia de dos variables del mismo tipo de estructura, sin tener que realizarla campo a campo.

Iniciación de variables de tipo estructuras

Como el resto de tipos, las variables de tipo estructura se pueden iniciar al tiempo que se declaran, y la forma de hacerlo es parecida a la de los vectores; entre llaves y separados por comas se ponen todos los valores iniciales de la estructura siguiendo el orden en el que se definieron los campos.

Estructuras dentro de estructuras

Una estructura puede anidar otras estructuras previamente definidas.
A la hora de mencionar los elementos de la estructura, se indican ordenadamente, el nombre de la estructura principal, el nombre de la estructura anidada y el nombre del elemento de la estructura anidada, separados por puntos.

Arrays dentro de estructuras

No sólo las cadenas son los únicos tipos de arrays que pueden formar parte de una estructura. En general, cualquier array puede ser un campo de una estructura.

Arrays de estructuras

De igual manera que con las variables, se puede formar un vector, una matriz, o un array de más dimensiones, de elementos del tipo estructura. Para ello, previamente tiene que estar definida la estructura.

Operaciones básicas

- Creación de una variable estructurada.

```
struct empleado jose, antonio;
```

- Asignación de estructuras.

```
jose = antonio;
```

- Acceso a los datos contenidos en la estructura (operador 'punto'.)

```
jose.nombre = "Jose";
```

- La lectura y escritura de la información de la estructura se debe hacer campo a campo.

```
cin >> jose.nombre;
cin >> jose.salario;
cin >> jose.telefono;
cout << antonio.nombre << antonio.telefono << endl;
```

Una estructura puede tener como campos otras estructuras:

```
struct departamento
{
   string nombre;
   int numero_empleados;
   struct empleado jefe;
};
```

También es posible definir *arrays* de estructuras:

```
struct empleado plantilla [30];
```

Una función puede devolver una estructura:

```
struct empleado PedirDatos (void);
```

Las estructuras dentro de las funciones

Las estructuras se manejan dentro de una función igual que los tipos simples de datos. Se puede definir funciones de tipo estructura, es decir, devolver una estructura dentro de una sentencia return, y puedo utilizar las estructuras como parámetros por valor y como parámetros por referencia.