TEMA 6: Tipos de datos estructurados

6.1. Introducción

Los tipos de datos estructurados o tipos compuestos son agrupaciones de otros tipos de datos.

Los tipos de datos estructurados más comunes son: vectores y matrices (*array*), cadenas de caracteres (*string*), registros y uniones, aunque estos últimos no los vamos a ver en este curso.

6.2. Vectores y Matrices

6.2.1. Vectores

Sirve para agrupar variables de un mismo tipo con un único nombre.

Supongamos que queremos declarar 10 variables de tipo entero (por ej. contadores). La única forma de hacerlo hasta ahora sería declararlos como variables individuales:

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

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

Otra forma de hacerlo es utilizando un vector (o *array*).

Forma de declarar un vector:

```
Tipo Nombre[NumElementos]
```

Donde *Tipo* indica el tipo de datos de los elementos del vector. El tipo puede ser cualquier tipo de dato, sea simple o estructurado. *NumElementos* es el número de elementos que contiene el vector. Tiene que ser siempre una constante de tipo entero.

Ejemplo:

```
int a[10];
```

De esta forma tenemos los 10 enteros agrupados y se pueden tratar como una única variable.

Podemos acceder a cada uno de los elementos de un vector utilizando un índice. A esta operación se la denomina **indexación**. El índice indica la posición del elemento dentro del vector.

En C++ El primer elemento es siempre el 0.

Ejemplo:

1º entero del vector:	`a[0]`
2º entero del vector:	`a[1]`
3º entero del vector:	`a[2]`
...	
Último entero del vector:	`a[9]`

Hay que tener presente que cuando indexamos un vector, el resultado es un elemento del vector y tendrá por tanto el tipo de este elemento. En nuestro ejemplo, `a[2]` es de tipo `int`, y se podrá utilizar por tanto en cualquier lugar del programa donde se admita una variable de tipo int.

La estructura más adecuada para manejar vectores es un bucle **`for`**, debido a que siempre conocemos el número de elementos del vector.

Ejemplo: *si tenemos un vector de 10 enteros que representan contadores, para inicializar todos los contadores a 0 sería:*

```
int a[10];
int i;

for(i = 0; i < 10; i++)
    a[i] = 0;
```

Para escribir en pantalla todos los elementos:

```
for(i = 0; i < 10; i++)
    cout << a[i];
```

Aunque una variable de tipo vector se puede declarar directamente, lo más recomendable para evitar errores es declarar un tipo de dato nuevo mediante un `typedef`. Además, para facilitar la realización de bucles y evitar también cometer errores, el número de elementos del vector tiene que ser una constante declarada previamente.

Ejemplo:

```
const int MAX = 10;
...
typedef int VectorInt[MAX];
...
VectorInt a;
...
for(i = 0; i < MAX; i++)
    a[i] = 0;
...
```

NO se pueden realizar asignaciones entre vectores. La siguiente asignación no se debe realizar **NUNCA**:

```
VectorInt a, b;
...
a = b;   // MAL
```

6.2.2. Matrices

Al igual que podemos crear vectores de cualquier tipo simple (enteros, reales, ...), podemos crear vectores tomando como tipo base otro vector. De esta manera lo que tendremos será una matriz.

Ejemplo:

```
typedef int Vector[10];
typedef Vector Matriz[20];

Matriz a;

a[19][3] = 0;
```

Esta declaración de la matriz se puede abreviar de la siguiente manera:

```
typedef int Matriz[20][10];
Matriz a;
```

O directamente declarar la variable (aunque es más recomendable declarar siempre un tipo):

```
int a[20][10];
```

Se pueden declarar matrices no sólo de dos dimensiones, sino de cualquier número de dimensiones.

La estructura más adecuada para manejar matrices serán bucles **`for`** anidados.

Ejemplo: *Visualizar una matriz.*

```
const int MAX_X, MAX_Y;
typedef int Matriz[MAX_X][MAX_Y];

Matriz a;
int i, j;

for(i = 0; i < MAX_X; i++)
    for(j = 0; j < MAX_Y; j++)
        cout << a[i][j];
```

6.2.3. Paso de *arrays* como parámetros

Los *arrays*, como cualquier otro tipo de datos, se pueden pasar como parámetros a una función. Sin embargo, en C++ no es posible pasarlos por valor. Siempre que se pasa un *array* a una función, se pasa por referencia, aunque no pongamos nada especial en el parámetro.

Ejemplo: *Una función para leer un vector por teclado se haría de la siguiente manera:*

```
const int MAX = 10;
typedef int Vector[MAX];

void LeerVector(Vector v)
{
    int i;

    for(i = 0; i < MAX; i++)
        cin >> v[i];
    return;
}
```

El vector v en este ejemplo está pasado por referencia, no es necesario escribir el &.

Al pasarse exclusivamente por referencia, el paso de *arrays* como parámetros es más eficiente, puesto que no hay que copiar cada vez toda la matriz (que normalmente son bastante grandes). El inconveniente es que dentro de la función se puede modificar el *array* aunque no se quisiera. Para evitar esto, en C++ existen los *parámetros constantes*. Son parámetros que se declaran como constantes, de manera que no pueden ser modificados.

Ejemplo: *Mostrar un vector.*

```
void MostrarVector(const Vector v)
{
    int i;

    for(i = 0; i < MAX; i++)
        cout << v[i];
    return;
}
```

En la función anterior, si intentamos realizar una asignación del tipo:
`a[0] = 1;`

nos dará un error de compilación.

Por último, una función no debe devolver valores de tipo *array*, es decir, no hay que declarar funciones de tipo *array*. Si se quiere devolver un *array*, se debe hacer utilizando un parámetro como en la función `LeerVector`.

6.2.4. Representación en memoria de un *array*

Los elementos de un vector se almacenan en memoria de forma contigua, es decir, uno al lado del otro.

Ejemplo:

```
int Vector[10]; //Si empieza en la dirección de memoria 10:
```

10	14	...	42	46
2	1		4	2

Por lo tanto, para calcular la posición en memoria de un elemento se utiliza la fórmula:

$d = d_o + Ind$ * `sizeof` (*Elemento*)

donde d_o es la dirección de memoria donde empieza el vector
Ind es el índice del elemento del vector del que queremos saber su dirección, y
Elemento es el tipo de elementos que contiene el vector

`sizeof` devuelve el tamaño en bytes de una variable o tipo de dato. `sizeof(char)`, por ejemplo devolverá `1`.

Puesto que una matriz no es más que un vector de vectores, el almacenamiento en memoria es también contiguo:

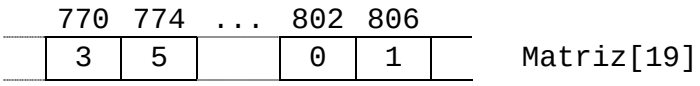
Ejemplo:

```
int Matriz[20][10];   //Si empieza en 10:
```

10	14	...	42	46	
2	1		4	2	Matriz[0]

50	54	...	82	86	
1	4		1	8	Matriz[1]

```
.......
```



Para calcular la posición del elemento Matriz[19][1], lo que hay que hacer es primero resolver el primer índice (Matriz[19]), que tiene como elementos vectores enteros de 10 elementos, y después el segundo tomando como dirección inicial la que resulta del calculo anterior. Así, si el vector empieza por ejemplo en la posición 10, la fórmula será:

```
d = 10 + 19 * sizeof(int [10]) + 1 * sizeof(int)
  = 10 + 760 + 4 = 774
```

6.2.5. Uso de *arrays* con número de elementos variable

Como ya se ha comentado, el tamaño de un *array* ha de ser una constante, por lo tanto no es posible modificar el tamaño de un *array* después de haber compilado el programa. Para solucionar esto existen dos posibilidades: la primera es utilizar memoria dinámica, que veremos en el Tema 8, la segunda es declarar un *array* lo suficientemente grande y utilizar sólo una parte de este *array*.

Por ejemplo, si vamos a realizar un programa para sumar vectores matemáticos y sabemos que en ningún caso vamos a tener vectores de un tamaño mayor de 10, podemos declarar todos los vectores de tamaño 10 y después utilizar sólo una parte del vector. Para ello necesitaremos utilizar también una variable entera extra para saber el tamaño actualmente utilizado por el vector, puesto que todas las operaciones se realizarán considerando el tamaño actual del vector y no el total.

Los inconvenientes de este método son que estamos desperdiciando memoria y además que no siempre hay un límite conocido al tamaño del *array*.

Ejemplos de suma de vectores con tamaño fijo:

```
/*********************************************/
/* Programa para la suma de vectores de enteros */
/*********************************************/

#include<iostream.h>

const int MAX = 3; // Tamanyo del vector
typedef int Vector[MAX];

void MostrarVector(const Vector v);
void LeerVector(Vector v);
```

```
void SumarVector(const Vector v1, const Vector v2, Vector vr);

int main()
{
    Vector v1, v2, vr;

    cout << "Introduce el primer vector:" << endl;
    LeerVector(v1);
    cout << "Introduce el segundo vector:" << endl;
    LeerVector(v2);

    SumarVector(v1, v2, vr);

    cout << "El vector suma es:" << endl;
    MostrarVector(vr);

    return 0;
}

void MostrarVector(const Vector v)
{
    int i;

    for(i = 0; i < MAX; i++)
        cout << v[i] << " ";
    cout << endl;
    return;
}

void LeerVector(Vector v)
{
    int i;

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

void SumarVector(const Vector v1, const Vector v2, Vector vr)
{
    int i;

    for(i = 0; i < MAX; i++)
        vr[i] = v1[i] + v2[i];
    return;
}
```

Ejemplos de suma de vectores con tamaño fijo y con tamaño variable (hasta un máximo de 10):

```
/*************************************************************/
/* Programa para la suma de vectores de enteros de           */
/* tamanyo variable                                          */
/*************************************************************/

#include<iostream.h>

const int MAX = 10;     // Tamanyo maximo del vector
typedef int Vector[MAX];

void MostrarVector(const Vector v, int tam);
void LeerVector(Vector v, int tam);
void SumarVector(const Vector v1, const Vector v2, Vector vr, int tam);

int main()
{
    Vector v1, v2, vr;
    int tam;            // Tamanyo real del vector

    cout << "Introduce tamanyo del vector (Max. " << MAX << "):";
    cin >> tam;
    cout << "Introduce el primer vector:" << endl;
    LeerVector(v1, tam);
    cout << "Introduce el segundo vector:" << endl;
    LeerVector(v2, tam);

    SumarVector(v1, v2, vr, tam);

    cout << "El vector suma es:" << endl;
    MostrarVector(vr, tam);
    return 0;
}

void MostrarVector(const Vector v, int tam)
{
    int i;

    for(i = 0; i < tam; i++)
        cout << v[i] << " ";
    cout << endl;
    return;
}

void LeerVector(Vector v, int tam)
{
    int i;

    for(i = 0; i < tam; i++)
        cin >> v[i];
    return;
}
```

```
void SumarVector(const Vector v1, const Vector v2, Vector vr, int tam)
{
    int i;

    for(i = 0; i < tam; i++)
        vr[i] = v1[i] + v2[i];
    return;
}
```

6.2.6. Búsqueda secuencial

Cuando se utilizan *arrays*, surge muy frecuentemente la necesidad de buscar un elemento concreto dentro del *array*. Para ello existen dos algoritmos muy utilizados: la búsqueda secuencial y la búsqueda dicotómica o binaria. En este Tema sólo veremos el primero de ellos.

La búsqueda secuencial es el tipo de búsqueda más sencilla y consiste en ir comparando el elemento buscado con todos los elementos del *array*, desde el primero hasta el último.

```
/***********************************************/
/* Busqueda secuencial en un vector            */
/***********************************************/

#include<iostream.h>

const int MAX = 10;

typedef int TipoBase;
typedef TipoBase Vector[MAX];

bool BusquedaSec(const Vector v, TipoBase buscado, int & pos);
```

```
int main()
{
    int pos;
    TipoBase elem;
    Vector v = {1, 3, 2, 5, 8, 2, 1, 3, 9, 2};

    cout << "Introduce el elemento a buscar:";
    cin >> elem;

    if (BusquedaSec(v,elem, pos))
        cout << "Elemento encontrado en posición " << pos << endl;
    else
        cout << "Elemento no encontrado" << endl;
    return 0;
}

/*********************************************************
* Funcion BusquedaSec
*
* Parametros:
* Nombre    E/S     Descripcion
* ------    ---     ---------------
*  v         E       Vector donde buscar
*  buscado   E       Elemento a buscar
*  pos       S       Posicion del elemento
*
* Valor devuelto:
*  bool    Cierto si encuentra el elemento, falso si no
*********************************************************/
bool BusquedaSec(const Vector v, TipoBase buscado, int & pos)
{
    pos = 0;
    while((pos < MAX - 1) && (v[pos] != buscado))
        pos++;
    return (v[pos] == buscado);
}
```

Si el vector en el que buscamos está ordenado, la busqueda se puede hacer más eficiente parando en cuanto lleguemos a un elemento que sea mayor que el buscado. Para ello, la función anterior se debería modificar de la siguiente manera:

```
bool BusquedaSecOrd(const Vector v, TipoBase buscado, int & pos)
{
    pos = 0;
    while((pos < MAX - 1) && (v[pos] < buscado))
        pos++;
    return (v[pos] == buscado);
}
```

Además, si no encontramos el elemento, esta función nos devolverá en el parámetro *pos* la posición donde debería ir colocado.

6.2.7. Inserción ordenada

En ocasiones interesa tener un vector con los elementos ordenados. Para ello, lo más sencillo es que cada vez que se introduzca un elemento en el vector, se introduzca ya en su posición correcta. Los pasos a seguir serán:

1. Encontrar la posición correcta del elemento utilizando una función de búsqueda (la función *BusquedaSecOrd*).

2. Hacer sitio al elemento moviendo todos los elementos posteriores una posición hacia arriba.

3. Insertar el elemento en su posición.

```
/*********************************************/
/* Inserción ordenada en un vector           */
/*********************************************/

#include<iostream.h>

const int MAX = 10;
typedef int TipoBase;
typedef TipoBase Vector[MAX];

bool BusquedaSecOrd(const Vector v, TipoBase buscado, int & pos,
      int tam);
bool InsercionOrd(Vector v, TipoBase elemento, int pos, int & tam);
```

```
int main()
{
    int pos;
    TipoBase elem;
    Vector v;
    int tam = 0;    // Numero de elementos del vector
    bool encontrado, correcto;

    cout << "Elemento a introducir en el vector (-1 para acabar):";
    cin >> elem;

    while(elem >= 0)
    {
        encontrado = BusquedaSecOrd(v, elem, pos, tam);
        correcto = InsercionOrd(v, elem, pos, tam);
        if(!correcto)
            cout << "Vector lleno !!!" << endl;
        cout << "Elemento a introducir (-1 para acabar):";
        cin >> elem;
    }
    return 0;
}

bool BusquedaSecOrd(const Vector v, TipoBase buscado, int & pos,
                    int tam)
{
    pos = 0;

    /* Permitimos que pos tome el valor de la primera posicion
     * libre
     */
    while((pos < tam) && (v[pos] < buscado))
        pos++;
    return (v[pos] == buscado && pos < tam);
}
```

```
bool InsercionOrd(Vector v, TipoBase elemento, int pos, int & tam)
{
    int i;
    bool insertado = false;

    if(tam < MAX)
    {
        for(i = tam; i > pos; i--)
            v[i] = v[i – 1];
        v[pos] = elemento;
        tam ++;
        insertado = true;
    }
    return insertado;
}
```

6.3. Cadenas de caracteres

Una cadena de caracteres es una secuencia de caracteres que es tratada como un único dato. Por ello la forma más lógica de implementar este tipo es con un vector de caracteres.

```
typedef char cadena[10];
cadena str1, str2;
```

El inconveniente de este vector es que, como ya hemos visto, la única forma de manejar un vector es elemento a elemento y esto hace que manejar la cadena de caracteres sea bastante incómodo.

Por ello, casi todos los lenguajes definen un tipo nuevo para manejar cadenas de caracteres. Este tipo es básicamente un vector de caracteres que lleva asociado un entero y un conjunto de funciones que permiten realizar operaciones de cadenas. El entero representa la longitud actual de la cadena, al igual que sucedía con los vectores de tamaño variable que ya hemos visto.

En C++ este tipo se denomina *string*, y está implementado como una clase de C++.

Declaración:

La forma de declarar y asignar un *string* es:

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

Asignación:

Al contrario que con los vectores, si que se pueden realizar asignaciones sin ningún problema.

```
s = s2;
s = "Adios";
s2 = 'a';
```

Acceso a caracteres:

Se puede acceder a las componentes del *string* mediante indexación, exactamente igual que en cualquier vector.

```
s = "Hola";
s[1] = 'a';
cout << s;        ->   "Hala"
```

Paso de *string* a funciones:

También se pueden pasar a funciones como parámetros y su comportamiento es como el de cualquier tipo simple (por defecto se pasa por valor). También es posible devolverlo como resultado de una función.

```
string saluda()
{
    string s = "Hola";
    return s;
}

...
```

```
cout << saluda();     ->  "Hola"
```

Comparaciones:

Se pueden comparar mediante los operadores de comparación:

```
string s = "Adios";
s < "Hola"   ->  true
s < "Ae"     ->  true
"aa" < s     ->  false
```

Longitud:

La longitud actual del *string* se puede consultar con length:

```
s = "Hola";
cout << s.length();   ->  4
s = "Adios";
cout << s.length();   ->  5
```

Para realizar operaciones sobre el string carácter a carácter, habrá que utilizar bucles for hasta la longitud del string:

```
// Función para cambiar todas las apariciones de un carácter
string cambia(string s, char ch, char nuevo_ch)
{
    int i;

    for(i = 0; i < s.length(); i++)
        if (s[i] == ch)
            s[i] = nuevo_ch;
    return s;
}
```

Lectura por teclado:

La introducción de datos de tipo string se puede realizar mediante el operador >>:

```
cin >> s;
```

Sin embargo tiene el problema de que sólo lee palabras, en cuanto llega a un separador (espacio, salto de línea, etc...) se acaba el string. Si por ejemplo, en la sentencia anterior introducimos "Hola Pepe", el valor que tomará s será "Hola".

Para leer una línea entera se puede usar getline:

```
getline(cin, s);
```

Otras operaciones:

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. Si no lo encuentra devuelve un valor imposible.

```
i = frase.find(palabra);
```

Devuelve la subcadena formado por 3 caracteres desde la posición 5 de frase

```
palabra = frase.substr(5,3);
```

6.3.1. Representación en memoria de cadenas de caracteres

Para representar cadenas de caracteres en memoria existen dos métodos básicos:

1) Guardar la longitud actual de la cadena en una variable de tipo entero asociada al vector de caracteres:

```
s = "HOLA";
    H O L A ..............
    4
```

Este método se utiliza por ejemplo en Pascal o C++.

2) Poner un carácter especial para indicar el final de la cadena:

```
    H O L A \0
```

Este método se utiliza por ejemplo en C, donde se emplea el carácter `\0` como final, o también al leer por teclado, donde el final de la línea se indica mediante el carácter `\n`.

6.4. Estructuras

Es una agrupación de elementos de tipos diferentes. También se denomina en ocasiones *registro*. Cada elemento se denomina campo, y se representa mediante un identificador propio. Cada campo puede ser de cualquier tipo.

```
struct Nombre
{
    Tipo Nombre_campo1;
    Tipo Nombre_campo2;
    ...
}
```

La declaración de estructura ya define un tipo (al igual que pasaba con enum) por lo tanto no es necesario utilizar typedef.

```
struct complejo
{
    float re;
    float im;
}
```

```
complejo c;

c.re = 0;
c.im = 0
```

```
struct Fecha
{
    int dia;
    int mes;
    int anyo;
}

Fecha f = {1, 1, 2001};
// (No funciona con componentes string)
```

Las estructuras también se pueden anidar, es decir, uno de los campos de una estructura puede ser a su vez otra estructura:

```
struct Alumno
{
    string nombre;
    Fecha nacimiento;
    int curso;
}
```

Asignación de estructuras:

Las estructuras sí se pueden asignar, al igual que los strings o cualquier tipo simple. Una asignación de estructuras es equivalente a una asignación de cada uno de los componentes.

```
Complejo c1, c2;
c1 = c2;
```

Paso de parámetros:

Las estructuras se pasan como parámetros exactamente igual que cualquier otro tipo simple. Las funciones también pueden devolver estructuras.

```
// Suma de complejos
Complejo SumaC(Complejo c1, Complejo c2)
{
    Complejo cres;

    cres.re = c1.re + c2.re;
    cres.im = c1.im + c2.im;
    return cres;
}
```

Vectores de estructuras:

Uno de los usos más comunes y útiles de las estructuras es como elementos de un vector.

(Ejemplo de guía de estilo)

6.5. Equivalencia de tipos

Hay ciertos tipos de operaciones que sólo se pueden realizar entre tipos de datos que sean equivalentes. Estas son por ejemplo la asignación o el pase de parámetros. Por ello resulta necesario definir el criterio de equivalencia de tipos de datos que sigue un lenguaje determinado.

Normalmente, los criterios de equivalencia utilizados son dos: equivalencia estructural y equivalencia por nombre.

Equivalencia estructural:

Dos tipos de datos son equivalentes cuando la estructura de la información que contienen es la misma.

Ejemplo:

```
typedef int Vector[10];
typedef int Vector2[10];

Vector v1;
Vector v2;
Vector2 v3;
```

```
int v4[10];
int v5[10];  // Todos equivalentes
```

Este criterio de equivalencia es utilizado sobre todo en el lenguaje C.

Equivalencia por nombre:

Dos variables son equivalentes únicamente cuando se han definido utilizando el mismo nombre de tipo.

Si en el ejemplo anterior utilizásemos equivalencia por nombre, únicamente serían equivalentes entre sí las variables v1 y v2.

Este criterio de equivalencia se utiliza en lenguajes como Pascal y Ada.

En C++ el criterio de equivalencia utilizado depende de los datos. Para vectores y matrices se utiliza criterio de equivalencia estructural. Sin embargo para estructuras se utiliza un criterio de equivalencia por nombre.

Ejemplo:

```
typedef int Vector[10];
typedef int Vector2[10];

struct Complejo
{
    float re;
    float im;
};

struct Complejo2
{
    float re;
    float im;
};

void f1(Vector2 x);
void f2(Complejo2 x);
```

```
int main()
{
    Vector v;
    Complejo c;

    f1(v);   // Correcto
    f2(c);   // Error

    return 0;
}
```