TEMA 11: Estructuras de datos

En este tema haremos una profundización del trato de datos tipo vector y tipo tupla, concretamente:

* Vectores abiertos, matrices o formaciones de varias dimensiones.
* Tipos unión, conjuntos y estructuras de datos complejas que combinan varias estructuras sencillas.

1. **Argumentos de tipo vector abierto.**

Si un subprograma debe operar con un vector recibido como argumento, necesita conocer el tipo y número de sus elementos. En el caso de que queramos tratar con el mismo subprograma vectores de distinto tamaño, no tiene sentido hacer diferentes subprogramas para diferentes tamaños de vectores. Por ello, tiene más sentido tratar un vector de cierto tipo, y pasar también como argumento el número de elementos que tiene el vector pasado. Para ello, el vector pasado como argumentos no tiene especificado su dimensión. Para realizar esto, basta con fijarse en la metodología usada en el siguiente ejemplo:

**void** EscribirVectorAbierto (**const int v[], int** numElementos){

**for** (**int** i = 0; i < numelementos; i++){

printf(“%10d”, v[i]);

}

}

Nota: Siempre tendremos que pasar como argumento la dimensión del vector, a no ser que dentro del propio subprograma se utilice la función strlen(v) de la librería <string.h>.

1. **Formaciones anidadas. Matrices.**

En esta sección se generaliza el concepto de vector para disponer de formaciones de más de una dimensión. Las matrices son estructuras de tipo **formación(array)** de dos o más dimensiones. Una manera sencilla de entenderlo es que cada elemento de un vector es otro vector. Se puede declarar una matriz de números enteros de la siguiente manera:

**Const int** NumFilas = 10;

Const int NumColumnas = 15;

typedef int TipoElemento;

typedef TipoElemento TipoFila[NumColumnas];

typedef TipoFila TipoMatriz[NumFilas];

**otra manera:**

typedef TipoElemento TipoMatriz[NumFilas][NumColumnas];

TipoMatriz matriz;

para inicializar cualquier elemento de la matriz

matriz [1][2] = 5;

# CONCEPTO GRÁFICO DE UN MATRIZ

**Vector Simple:**

**typedef char** String[7]; **🡪 Declaración.**

String = “Hola.Ana”; 🡪 Inicialización.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Valor de dato | ‘H’ | ‘o’ | ‘l’ | ‘a’ | ‘ . ’ | ‘A’ | ‘n’ | ‘a’ | ‘\0’ |
| Posición | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

**Matriz (2 dimensiones):** Si la posición fila más la posición columna dan un número par el valor de dicha posición es ‘a’, en caso contrario, el valor será ‘b’.

**tydepef char** Matriz[8]**[9];**

**Matriz** MyMatriz;

**for (int i = 0; i < 8; i++{**

**for (int j = 0; j < 9; j++) {**

**if**((i+j)%2 == 0){

MyMatriz[i][j] = ‘a’;

}else{

My Matriz[i][j] = ‘b’;

**}**

**}**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ |
| ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ |
| ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ |
| ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ |
| ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ |
| ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ |
| ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ |
| ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ | ‘a’ | ‘b’ |

# OPERACIONES CON MATRICES

La operación con elementos individuales de una matriz se puede realizar de forma análoga a la operación con variables simples de ese tipo. Por ejemplo, para imprimir matrices del tipo declarado anteriormente:

**void** EscribirMatriz ( **const** TipoMatriz m) {

**for (int** i = 0; i < NumFilas; i++**){**

**for (int** j; j < NumColumnas; j++**){**

**printf**(“%10d”, m[i][j]);

**}**

**}**

}

Otras operaciones:

typedef int matrizEnteros[10][10];

int main(){

MatrizEnteros MyMatriz1;

MatrizEnteros MyMatriz2;

MatrizEnteros MatrizResultado;

}

SUMA 🡪 MatrizResultado[0] = MyMatriz1[3]+Mymatriz2[9]+10;

MULTIPLICACIÓN 🡪 MatrizResultado[1] =(MyMatriz1[1] \* 9)+5;

Nota: Siempre se tiene que tartar elementos del mismo tipo o se realiza un conversion previa.

No podemos enviar como argumento a un procedimiento o una función una matriz abierta, es decir, un matriz cuyas dimensiones no están definidas.

1. **Esquema Unión.**

Hay aplicaciones en la que es conveniente que el tipo de un dato variase según las circunstancias. Si las posibilidades de variación son un conjunto finito de tipos, se puede decir que el tipo de dato corresponde a un esquema que es la unión de los tipos particulares posibles. Situaciones en las que se utiliza un esquema de tipo unión:

* Datos que pueden representarse de diferentes maneras.
  + Coordenadas polares y cartesianas de un mismo punto. En el caso de las coordenadas polares hace falta un ángulo y un punto, y en el caso de las coordenadas cartesianas son dos puntos.
* Programa que operan indistintamente con varias clases de datos.
  + Programa que opere con números enteros, fraccionarios o decimales indistintamente.
* Datos estructurados con elementos opcionales.
  + Imaginemos una colección de figuras geométricas (punto, circulo, cuadrado, rectángulos, rombo, triangulo…etc.). Dependiendo de la figura geométrica que esté tratando el programa hará falta una colección de datos determinada. Por ejemplo:

|  |  |
| --- | --- |
| FIGURA | PARÁMETROS NECESARIOS |
| Triangulo | Ángulos de vértices, Longitud de lados, altura…etc. |
| Cuadrado | Longitud de lados. |
| Circulo | Perímetro, Radio, Diámetro…etc. |

EL TIPO **unión**

Un tipo **unión** puede utilizarse en programas en C+- y se definen como una colección de campos alternativos, de tal manera que cada dato particular sólo usará uno de esos campos en un momento dado, dependiendo de la alternativa aplicable. La declaración se realiza de manera similar a la de un **struct** pero usando la palabra clave **unión.** Veamos un ejemplo con un programa que trata con números enteros, reales o fracciones indistintamente:

**typedef struct** TipoFraccion {

**int** numerador;

**int** Denominados;

};

**typedef** **union** TipoNumero {

**int** ValorEntero;

**float** valorReal;

tipoFraccion valorRacional;

};

La referencia a los elementos componentes de un tipo de dato **union** se realiza de manera semejante a los datos tipo struct.

TipoNumero numero, otro, fraccion1, fraccion2;

numero.valorEntero = 33; 🡪 **Estamos seleccionando** **un tipo de dato y descartando las demás posibilidades. De aquí en adelante la variable “numero” solo puede ser de tipo entero.**

otro.valorReal = float (numero.valorEntero);

fraccion2.valorRacional = (fraccion1.valorRacional);

Si además, una de las componentes del tipo de dato **union** es a su vez otra tupla(struct) o union habrá que usar varias designaciones.

fraccion1.valorRacional.numerador = 33;

fraccion1.valorRacional.denominador = 44;

printf(“Numerador: %d”,fraccion1.valorRacional.numerador);

Para poder controlar cuál de las componentes de un tipo de dato unión esta activa o cual queremos activar. Tenemos que crear un código que se encargue de realizar esta tarea de manera clara.

**typedef enum** ClaseNumero {Entero , Real, Fraccion};

woid EscribirNumero( TipoNumero n, ClaseNumero c ) {

**switch** (c){

case Entero:

printf("%d", n. valorEntero );

break;

case Real:

printf( "" f". n. valor Real ) ;

break;

case Fraccion:

printf( "%d /~ ", n.valorRacional.numerador, n.valorRacional .denominador );

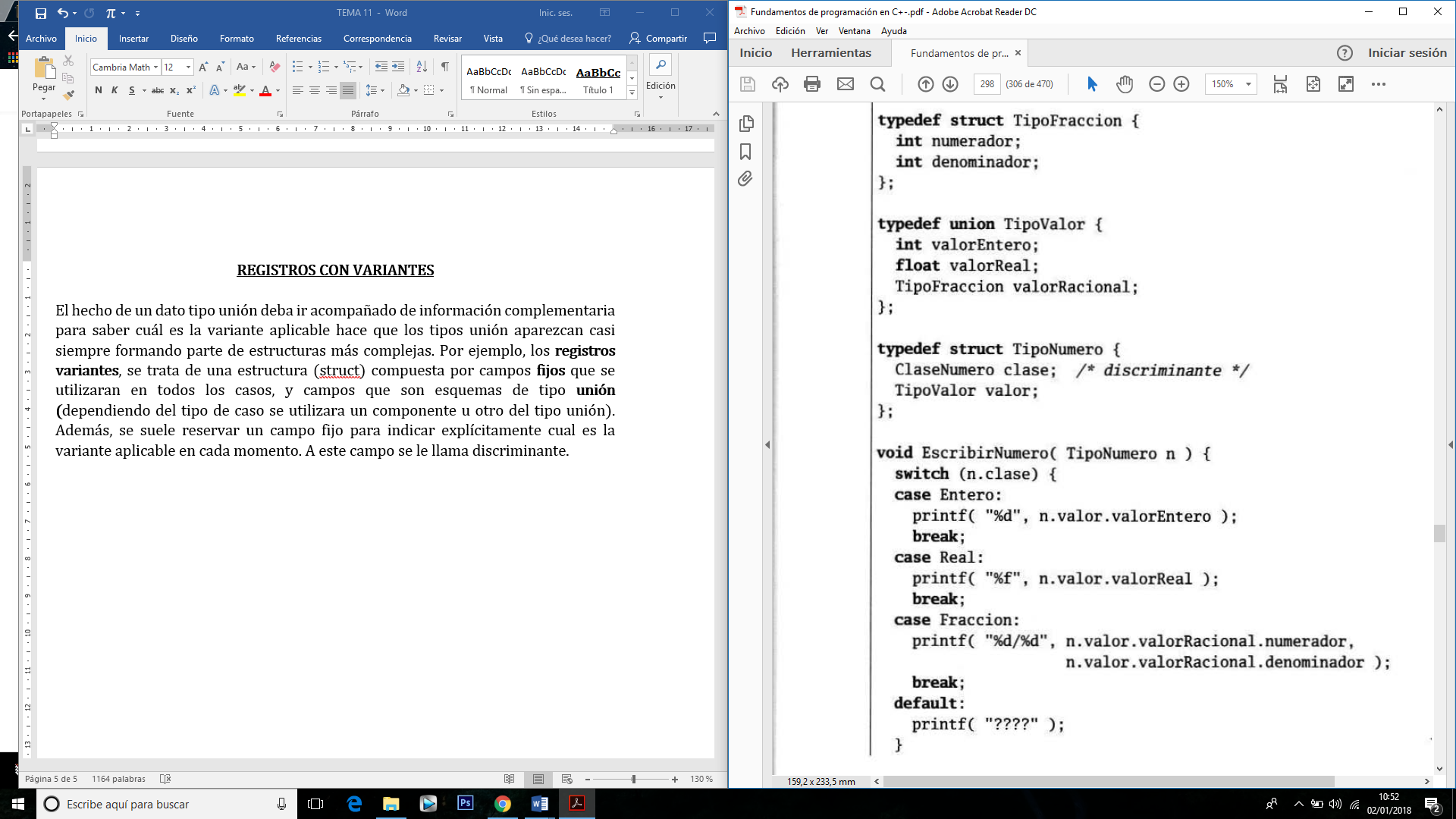
break;

default:

printf( "????" ) ;

**REGISTROS CON VARIANTES**

El hecho de un dato tipo unión deba ir acompañado de información complementaria para saber cuál es la variante aplicable hace que los tipos unión aparezcan casi siempre formando parte de estructuras más complejas. Por ejemplo, los **registros variantes**, se trata de una estructura (struct) compuesta por campos **fijos** que se utilizaran en todos los casos, y campos que son esquemas de tipo **unión (**dependiendo del tipo de caso se utilizara un componente u otro del tipo unión). Además, se suele reservar un campo fijo para indicar explícitamente cual es la variante aplicable en cada momento. A este campo se le llama discriminante.



Si el tratamiento de los registros con variantes se hace solo mediante subprogramas que comprueban siempre el discriminante antes de operar, el código será mucho más seguro.

1. **Esquemas de datos y esquemas de acciones.**

Una vez presentadas las estructuras de datos de tipo formación(vectores) y registro, incluidos los registros con variantes, es interesante conocer que esquemas de acciones (secuencia, selección o iteración) es adecuado usar para cada esquema de datos (vector, registro o registro con variante).

* En el caso de usar una tupla(registro(**struct**)) el esquema de acciones más adecuado para usar este tipo de dato es el de secuencia, siendo esta, un conjunto de instrucciones ejecutadas en el orden que aparecen. Por ejemplo:

/\*ESQUEMAS DE DATOS\*/

/\*Esquema Tupla\*/

**typedef** **struct** TipoTupla {

**char** uno, dos, tres;

};

TipoTupla agregado;

/\*ESQUEMAS DE ACCIONES\*/

/\*Secuecia\*/

printf(“%c”, agregado.uno);

printf(“%c”, agregado.dos);

printf(“%c”, agregado.tres);

* En el caso de usar un registro variante (esquema de tipo **union**), el esquema de acciones más adecuado para manejar este esquema de datos es la Selección (IF-ELSE; SWITCH). Por ejemplo:

/\*ESQUEMAS DE DATOS\*/

/\*Esquema Registro Variane o Union\*/

**typedef** **union** TipoUnion {

**int** alfa;

**int** beta;

};

TipoUnion variante;

/\*ESQUEMAS DE ACCIONES\*/

/\*Selección\*/

**if**( numeroEntero) {

printf(“%d”, variante.alfa);

} **else** {

printf(“%d”, variante.beta);

}

* En el caso de usar un tipo de dato formación (vector, string) la secuencia de acciones más adecuada es la de iteración, es decir, la repetición de una misma acción un número determinado de veces. Por ejemplo, en el caso de que queramos recorrer un vector.

1. **Estructuras combinadas.**

Tal y como hemos visto en temas anteriores con los esquemas de acciones (selección, iteración u secuencia) podemos anidar uno dentro de otros con total libertad creando estructuras de acción más complejas. Lo mismo ocurre con los esquemas de datos. Se pueden definir sin límite de complejidad estructuras que a su vez contienen otras estructuras.

# FORMAS DE COMBINACIÓN

La manera de combinas las estructuras de datos es hacer que los elementos de una estructura sean a su vez otras estructuras. Por ejemplo:

**typedef** struct TipoPunto {

**float** x,y; /\*Cada vértice tiene una componen x e y\*/

}

**typedef** TipoPunto TipoTriangulo[3]; /\* Un triángulo tiene 3 vértices, este vector contiene un registro dentro\*/

**typedef** TipoPunto TipoPuntos[100]; /\*Se crea vector con un máximo de 100 puntos (x,y), además, este vector tiene dentro otro registro\*/

**typedef struct** TipoListaPuntos {

**int** numeroPuntos;

TipoPuntos puntos; 🡪 Este registro contiene dentro un vector.

};

# TABLAS

El esquema de tabla, que puede plantearse como una formación (vector) simple de registros(struct), en otros contextos se le denomina también diccionario o relación. Los esquemas de tabla son el fundamento de las bases de datos relacionales, aunque su implementación es muy diferente. Por ejemplo:

Definimos una tabla destinada a contener la identificación de las provincias.

**typedef** **enum** TipoProvincia {

SinProvincia, Alava, Albacete, …. Zaragoza

};

**const** **int** MaxProvincias = int(Zaragoza);

**typedef** **char** TipoSiglas[2];

ESQUEMA DE TABLA: ES UN VECTOR, CUYOS ELEMENTOS SON REGISTROS(STRUCT)

**typedef** **char** TipoNombre[30];

**typedef struct** TipoDatosProvincia {

TipoSiglas siglas;

TipoNombre nombre;

**int** código;

};

**typedef** TipoDatosProvincia TipoTablaProvincias[MaxProvincias+1];