Estructura de datos Registro (struct)

Durante el curso hemos trabajado con variables simples, que nos permitían almacenar un único valor de un tipo específico de datos, y con variables compuestas (también denominadas complejas o estructuradas) como los vectores y las matrices, donde podíamos almacenar un conjunto de valores de un tipo determinado bajo un mismo nombre, y acceder a cada elemento individual mediante la notación de subíndices o punteros. Vimos que al utilizar arrays en determinados ejercicios la resolución se simplificaba: no hacía falta declarar, por ejemplo, múltiples vectores, o variables simples, ya que se podían unir todas esas variables dentro de una única estructura de datos. La razón por la que los distintos lenguajes de programación proveen estos tipos de datos, es justamente lo señalado: hacer más sencilla la tarea del programador. Sin embargo los arrays vistos hasta el momento no resuelven todos los problemas que se pueden presentar, ya que sólo admiten un tipo único de datos. No es posible declarar una matriz que tenga una fila (o una columna) de enteros, y la otra de float.

Supongamos que tenemos que cargar en memoria un lote de datos de 100 registros como el siguiente:

- Código de producto (string de 6)
- Precio unitario (float)
- Stock (entero)
- Rubro (char)

Para poder luego, por ejemplo, saber el precio unitario de un determinado producto a partir de su código, deberíamos cargar todos los códigos en una matriz de char (ya que el código es un string, o sea un vector de char), y cada uno de los otros datos en vectores independientes, cada uno de ellos del tipo que corresponda. Supongamos ahora que para resolver el problema necesitamos 15 funciones, y que cada una de ellas necesita acceder al conjunto de los arrays mencionados, por lo que sería imprescindible enviar como parámetros punteros a esos arrays. Vayamos más allá: imaginemos que el lote no tiene sólo 4 elementos distintos de información para cada producto, sino 20: habría que pasarle a las funciones punteros a los 20 arrays, lo que le agrega una gran complejidad al programa.

Dejemos un momento la programación, y pensemos como resolveríamos el problema en caso de no contar con una computadora. La forma más sencilla sería tener en un papel una tabla con el siguiente formato:

Código de producto	Precio unitario	Stock	Rubro
“aaaaa”	3.2	100	‘A’
“bbbbb”	1.6	50	‘B’

En cada columna se pondría una característica del producto. Cada línea representaría los datos completos (todas las características) de un producto en particular. Para ubicar el precio de un producto bastaría con buscar en la columna correspondiente el código que lo identifica.

Volviendo ahora a la programación, intuitivamente surge que sería muy útil contar con una estructura de datos que permitiera representar estos datos de la misma manera que lo hacemos en el papel. Si la tabla entera tiene un nombre de variable único, se simplificaría la resolución del problema: sólo sería necesario pasarles a las funciones un parámetro. Afortunadamente, como ya se dijo, los lenguajes proveen las estructuras de datos más usuales, o mecanismos para crearlas; y para agrupar elementos de información heterogéneos la estructura se denomina genéricamente registro (record). En C los registros se denominan **struct.**

Un struct, entonces, es una estructura de datos en la cual es posible agrupar variables de distinto tipo bajo un mismo nombre, y luego acceder a cada elemento de información de manera individual. Cada una de las variables recibe el nombre de **campo**; es decir un struct contará con 1 ó más campos, de igual o diferente tipo de datos. El conjunto de campos recibe el nombre de **registro.**

En C para poder utilizar un struct, es necesario primero notificar al compilador que se utilizará una estructura compuesta de determinados campos. Estos campos podrán ser de cualquiera de los tipos conocidos, incluyendo arrays, punteros, e incluso variables struct de otro tipo definido previamente. Podríamos ver esto como la posibilidad de añadir al lenguaje un nuevo tipo de dato creado por el programador para responder a las necesidades del programa que está escribiendo. Como el lenguaje no conoce este "tipo" previamente, no sabe como tratarlo. Para que pueda hacerlo hay que definir un "molde" que indique cómo se llamará este tipo de datos compuesto, y cuáles serán los campos que lo compongan.

Por ejemplo, para el lote de registros indicado más arriba el molde sería:

```
struct articulo
        {
        char cod_p[6];
        float pu;
        int stock;
        char rubro;
        };
```

Nota: no olvidar de poner el ; luego de la llave de cierre

Al definir el struct articulo, o cualquier otro que necesitemos, no estamos reservando memoria ya que no estamos declarando una variable: estamos creando el molde para un nuevo tipo de datos. Luego será posible declarar variables de ese tipo nuevo.

Si el molde es definido fuera de cualquier función será visible para todas las funciones del programa fuente. Esto permite que cada función declare variables de tipo **struct articulo** dentro de su cuerpo. Por ejemplo:

```
int main()
        {
        struct articulo solo_uno, varios[100], matriz[3][3], *puntero;
        ----------
        ---------
        ---------
        }
```

En main() se declararon 4 variables de tipo struct articulo:

- solo_uno, es una variable simple: sólo puede almacenar un registro, es decir, un valor para cada uno de los campos (sería una sola línea de la tabla).

- varios, es un vector de 100 elementos, por lo que se podría almacenar el lote completo de registros (la tabla entera).

- matriz: es una matriz compuesta de 3 filas y 3 columnas.

- puntero, que es un puntero a struct articulo.

Observemos la similitud con los tipos primitivos del lenguaje(int, float, char). Al igual que con éstos, al definir el molde para struct articulo podemos luego declarar variables simples, vectores, matrices y punteros de ese "tipo" nuevo que creamos. En el desarrollo del curso usaremos principalmente variables simples, punteros y vectores de struct.

Veamos como se utilizan los struct, a partir de las variables declaradas en main().

```
-------
solo_uno.pu=2.5;
solo_uno.stock=15;
solo_uno.rubro='A';
strcpy(solo_uno.cod_p, "aaaaa");
-----------
```

Como lo muestra el fragmento de código anterior, para acceder a cada campo de un struct se usa la siguiente notación:

nombre_variable.nombre_de_campo

luego, son válidas las reglas conocidas de codificación, de acuerdo al tipo de dato al que pertenezca el campo. Por ejemplo:

```
------
if(solo_uno.rubro= = 'A')
        cout<< solo_uno.cod_p;
---------
```

Supongamos que hemos declarado en main() además de las variables de tipo struct articulo, una variable de tipo entero llamada st, y otra de tipo float llamada precio, podríamos hacer las siguientes asignaciones:

```
-------
st=solo_uno.stock;
```

precio=solo_uno.pu;
---
ya que solo_uno.stock es de tipo entero (al igual que st), y solo_uno.pu es float como precio.

También sería válida la siguiente asignación:
---
otra_var=solo_uno;
---
si otra_var hubiera sido declarada como una variable simple de struct articulo.

Para vectores la notación sería:
---
varios[0].stock=15;
varios[0].pu=5.32;
varios[0].rubro='B';
strcpy(varios[0].cod_p,"bbbbb");
---

ya que además del nombre de la variable debemos con el subíndice indicar en cual de los 100 registros queremos hacer la asignación.

En un vector de registros, cada uno de sus componentes es un registro, por lo que podríamos hacer la asignación

v[0]=solo_uno

Paso de struct como parámetro entre funciones

Al igual que con el resto de las variables, es posible pasar un struct por valor o por dirección. Está sujeto a las mismas restricciones que el resto de las variables de tipos primitivos:
- se puede pasar por valor cuando es una variable simple;
- se tiene que pasar por dirección cuando se trata de un vector, o una matriz.

Veamos ejemplos:
```
void mostrar (struct articulo simple);
int main()
        {
        struct articulo x;
        strcpy(x.cod_p,"aaaaa");
        x.pu=6.48;
        x.stock= 100;
        x.rubro='C';
        mostrar(x);
        system("pause");
        return 0;
        }
```

```
void mostrar(struct articulo simple)
        {
        cout<<simple.cod_p;
        cout<<simple.pu;
        cout<<simple.stock;
        cout<<simple.rubro;
        }
```

main() asigna valores a los campos del struct, y llama a la función mostrar con el struct como parámetro por valor. La función hace una copia local de x en simple, y luego muestra cada uno de los campos. Como sabemos, al enviar un parámetro por valor la función que recibe el parámetro no puede modificar el valor original de la variable en la función que llama. Supongamos que quisieramos hacer una función para asignar valores al struct x: en este caso debemos mandar la dirección de x, y recibirla en la función en una variable puntero.

```
void mostrar (struct articulo simple);
void cargar(struct articulo *p);

int main()
        {
        struct articulo x;
        cargar(&x);
        mostrar(x);
        system("pause");
        return 0;
        }

void cargar(struct articulo *p)
        {
        cin>>p->cod_p);
        cin>>p->.pu;
        cin>>p->stock;
        cin>>p->rubro;
        }
```

Al enviar la dirección del struct articulo x, y recibirla en la variable puntero p, la notación cambia: en vez de usar el punto para acceder a un campo en particular, se utiliza el operador flecha ->. Genéricamente podemos definirlo:

nombre_puntero->nombre_de_campo

La razón del uso de este nuevo operador es que reemplaza a la notación

(*puntero).nombre_de_campo

En efecto, como p es un puntero almacena una dirección. En este caso la dirección de x, o sea que
*p= = x, (también es cierto que p= = &x), luego:
x.pu= = (*p).pu

Para simplificar la escritura, se decidió implementar este nuevo operador, formado por el guión y el signo de mayor. No obstante, puede utilizarse con idéntico resultado cualquiera de las dos notaciones.

Para el caso de vectores, al igual que para el resto de los tipos de datos, el nombre del vector sin subíndice, es la dirección de inicio del vector.

```
void mostrarArticulos (struct articulo *p, int);
void cargarArticulos(struct articulo *p, int);

int main()
	{
	struct articulo x[100];
	cargarArticulos(x,100);
	mostrarArticulos(x,100);
	return 0;
	}

void cargarArticulos(struct articulo *p, int tam)
	{
	int i;
	for(i=0;i<tam;i++)
		{
		cin>> p[i].cod_p;
		cin>> p[i].pu;
		cin>> p[i].stock;
		cin>> p[i].rubro;
		}
	}

void mostrarArticulos(struct articulo *p, int tam)
	{
	int i;
	for(i=0;i<tam;i++)
		{
		cout<< p[i].cod_p;
		cout<< p[i].pu;
		cout<< p[i].stock;
		cout<< p[i].rubro;
		}
	}
```

Usando la notación de punteros en las funciones:

```
void cargarArticulos(struct articulo *p, int tam)
	{
	int i;
	for(i=0;i<tam;i++)
		{
		cin>> ( p+i)->cod_p;
		cin>> (p+i)->pu;
		cin>> (p+i)->stock;
		cin>> (p+i)->rubro;
		}
	}

void mostrarArticulos(struct articulo *p, int tam)
	{
	int i;
	for(i=0;i<tam;i++)
		{
		cout<< (p+i)->cod_p;
		cout<< (p+i)->pu;
		cout<< (p+i)->stock;
		cout<< (p+i)->rubro;
		}
	}
```

Lo que hemos visto anteriormente en aritmética de punteros para los vectores de tipos de datos primitivos se aplica para vectores de struct. Ya sabemos que las dos notaciones siguientes son equivalentes:

*(v+i)=v[i]

Pero al tratarse de un vector de struct, el nombre solo del vector y su posición no alcanzan: es necesario indicar que campo queremos mostrar o cargar. O sea que las dos notaciones siguientes también serían equivalentes:

*(v+i).pu=v[i].pu

También vimos que el operador flecha reemplaza a la notación *puntero.nombre_campo, con lo cual:

(v+i)->pu=v[i].pu

que es la notación utilizada en el ejercicio anterior.

Otra forma de escribir la función cargarArticulos() sería:

```
void cargarArticulos(struct articulo *p, int tam)
        {
        int i;
        for(i=0;i<tam;i++)
                {
                cin>> p->cod_p;
                cin>> p->pu;
                cin>> p->stock;
                cin>> p->rubro;
                p++;
                }
        }
```

En este caso el acceso a cada posición se logra incrementando el puntero, en lugar de sumarle el valor de i.

Si usáramos un for de punteros, la función sería:

```
void cargarArticulos(struct articulo *p, int tam)
        {
        struct articulo  *i;
        for(i=p;i<p+tam;i++)
                {
                cin>> i->cod_p;
                cin>> i->pu;
                cin>> i->stock;
                cin>> i->rubro;
                }
        }
```

El incremento del puntero lo obtenemos a partir de la variable del for.