Estructuras de datos

Al estudiar el capítulo de funciones, aprendimos la estrategia: **Divide y Vencerás**, que se aplica en dos pasos: dividir y coordinar la interacción; ahora aprenderemos la inversa: **La unión hace la esfuerza,** la cual se implementa también en dos pasos:

1) Conocer los componentes (**campos**) individuales, por ejemplo:

```
int    codigo;
char nombre[25];
int    edad;
```

Estas tres variables pueden referirse, por ejemplo, a un estudiante; pero no lo reflejan por separado, les falta **cohesión**.

2) Cohesionar los campos para formar una estructura con un **nombre** que represente la cohesión.

```
struct Estudiante {          // Define la estructura Estudiante
      int    codigo;
      char nombre[25];
      int    edad;
};
```

Sintaxis
Una estructura define a un tipo de dato complejo que agrupa a varios tipos de dato (simples o complejos); suele definirse como global (antes de todas las funciones) para ser usada en todo el programa; pero recuerde que no es la mejor práctica usar variables globales. Puede definir una estructura de dos modos:

// define estructura
struct [MiStru] {
  int campo1; // cualquier tipo de dato
  char campo2[20]; // cualquier tipo de dato
  int campo3; // cualquier tipo de dato
} [miStru1, miStru2, ...]; // [Declara variables tipo **struct MiStru**]
Donde:
  MiStru es el nombre de la estructura
  miStru1, **miStru2**, ... son variables del tipo **struct MiStru**.
 	
No tiene sentido omitir las dos partes opcionales:
struct {
  tipo1 campo1;
  ……..
}; // se define sin nombre, no se usa, y no podrá ser referida.
	// define estructura para untilizarla como tipo nativo
	typedef struct {
	  int campo1; // cualquier tipo de dato
	  char campo2[20]; // cualquier tipo de dato
	  int campo3; // cualquier tipo de dato
	} **MiStru**;
	Donde:
	  MiStru es el nombre de la estructura

// define una variable **Struct MiStru** estud1 = {1, "Juan Carlos", 25};	// define una variable **MiStru** estud1 = {1, "Juan Carlos", 25};
Cada variable de estructura ocupa un área continua de longitud = suma de las longitudes de todos sus campos	

¡Admírate, alégrate… !!!! Este procedimiento da el fundamento lógico para aplicar la estrategia **La unión hace la fuerza** en el mundo virtual; en forma análoga se aplica en el mundo real.

Para aplicarlo a las ciencias sociales; se debe complementar con los sentimientos, responsabilidad personal, complementariedad, afectos, coraje, etc.

Definir, asignar, leer e imprimir variables de tipo struct

Estas operaciones son similares a las que se hacen con datos de tipo primario; también puede definir y asignar valores al mismo tiempo. Se puede definir de dos formas equivalentes:

```
struct Estudiante {
      int codigo;
      char nombre[25];
} est1, .... ;                    // forma 1, también puede asignar valores
```

```
struct Estudiante est2, ...;                    // forma 2.

struct Estudiante est3 = {124, "José Vargas"};  // forma 2, también asigna valores

est1.codigo = 123;                              // asina valor a la variable est1.codigo
strcpy(est1.nombre, "María Mar");               // asina valor a la cadena est1.nombre

scanf("%d", &est1.codigo);                      // lee un entero
scanf("%s", est1.nombre);                       // lee una cadena
printf("%d", est1.codigo);
printf("%s", est1.nombre);
```

Ejemplo:

```
struct  {
    int codigo;
    char nombre[25];
    int edad;
} est1, est2, est3;                 // define las variables est1, est2 y est3 de tipo Estudiante.
```

La definición anterior es equivalente a:

```
struct Estudiante {                 // define la estructura
    int codigo, edad;
    char nombre[25];
} est1;                             // define las variables est1
struct Estudiante est2, est3;       // define las variables est2 y est3 de tipo Estudiante.
```

Otra alternativa es:

```
typedef struct {                    // define la estructura
    int codigo, edad;
    char nombre[25];
} Estudiante;                       // define el tipo Estudiante
Estudiante est1, est2, est3;        // define las variables est2 y est3 de tipo Estudiante.
```

Atento:

Arreglos	**Estructuras**
// Se puede definir y asignar todos los elementos: int a1[3] = {1, 2, 3 }, a2[3], a3[3];	// Se puede definir y asignar todos los campos: Estudiante e1 = {1, 20, “Juan Vargas”}, e2, e3;

a2 = {6, 7, 8}; // ERROR, se asigna uno por uno:
a2[0] = 6;
a2[1] = 7;
a2[2] = 8;
	e2 = {2, 21, “José Ramírez”}; // ERROR, se asigna uno por uno:
	e2.codigo = 2;
	e2.edad = 21;
	strcpy(e2.nombre, “José Ramírez”); // cadena

a3 = a1; // Error, se asigna uno por uno:
a3[0] = a1[0]; a3[1] = a1[1]; a3[2] = a1[2];
	e3 = e1; // Correcto

```
// 09_01.c : Definir, asignar valores e imprimir datos de una estructura
#include<stdio.h>
#include<string.h>
void main(void){
    struct Estudiante {
        int  codigo;
        char nombre[25];
        int  edad;
    } est1 = {123, "Carlos Rodríguez", 25};          // asigna valores a est1
    struct Estudiante est2 = {124, "José Vargas", 51}, est3;
    //    est3 = {125, "María Mar", 23};             Error: No compila
    est3.codigo = 125;                               // asignación a cada campo
    strcpy(est3.nombre, "María Mar");                // atento: nombre es un string
    est3.edad   = 23;
    printf("%s\n%s\n%s\n", est1.nombre, est2.nombre, est3.nombre);
}
```

Salida:

```
Carlos  Rodríguez
```

José Vargas
María Mar

Arreglo de estructuras

Se definen arreglos como de costumbre:

```
struct Estudiante {
   ....
}  est[10];                    // forma 1: arreglo de 10 elementos

struct Estudiante est[2];      // forma 2: arreglo de 2 elementos
struct Estudiante est[3] = {{123, "Carlos Rodríguez", 25}, {125, "María Mar", 23}};  // no se asignó valores a est[2].

scanf("%d", &est[2].codigo);          // lee  est[2].codigo
est[3].nombre = "Maria Mar";          // Error: nombre es una cadena
strcpy(est[2].nombre, "María Mar");
printf("%s", est[2].nombre);          // Imprime  est[2].nombre
```

Anidamientos de estructuras

Una estructura puede anidarse dentro de otra, se puede formar una cascada, ejemplo: Defina, lea e imprima datos de una estructura.

```
// 09_02.c : Definir, asignar valores e imprimir datos de una estructura con anidamiento
#include <stdio.h>
#include <string.h>
void main(void){
        struct Nombre {                                 // estructura simple
              char nombre[15];
              char apellido[15];
         };
        struct Estudiante {                             // estructura más compleja que anida
              int  codigo;
              struct Nombre nombre;                     // estructura simple anidada
              int  edad;
        } est1 ={123, {"José", "Romero"}, 19}, est2;    // define dos variables y asigna valores a la primera
        printf("Codigo: ");     scanf("%d", &est2.codigo);          // tipee 12
        printf("Nombre: ");     scanf("%s", est2.nombre.nombre);    //  tipee Carlos
        strcpy(est2.nombre.apellido, "100 Fuegos");
        printf("Edad: ");       scanf("%d", &est2.edad);            // tipee 15
        printf("\nReporte\n");
        printf("Codigo: %d\n", est2.codigo);
        printf("Nombre: %s\n", est2.nombre.nombre);
        printf("Apellido: %s\n", est2.nombre.apellido);
        printf("Edad: %d\n", est2.edad);
}
```

Salida:

```
Codigo: 12
Nombre: Carlos
Edad: 15

Reporte
Codigo: 12
Nombre: Carlos
Apellido: 100 Fuegos
Edad: 15
```

Apuntador a estructura

Apuntador a estructura

Un apuntador puede apuntar a cualquier tipo de variable, en particular a una de tipo estructura. Sea:

```
struct MiStru {
    int  campo1;                // 4 bytes
    char campo2[25];            // 25 bytes
};
```

struct MiStru miStru, ***pmiStru** = **&miStru**; // Atento se usa **&**, a diferencia de apuntador a arreglo.

Mientras pmiStru apunte a miStru, se cumplen las siguientes equivalencias:
miStru.Campo1 == (*pmiStru).campo1 == pmiStru->campo1
Se usa la flecha para hacer menos incómoda la notación de puntero, se escribe: - (menos) seguido de >

Atento, la instrucción:
if(pmiStru == &miStru.campo1) // es true, porque son la misma dirección
Pero, compila con warning debido a que pmiStru es un puntero a tipo MiStru y &(miStru.campo1) es de tipo int; sin embargo. El casting automático funciona y se ejecuta bien.

```
// 09_03.c : Define y utiliza un apuntador a estructura
#include<stdio.h>
#include<string.h>
void main(void){
        struct Dato {
             int  campo1;
             char campo2[25];
        } dato, *pdato = &dato;              // define dato y *pdato
        dato.campo1 = 11;                    // asigna valor
//      (*pdato).campo1 = 11;                // equivalente
//      pdato->campo1 = 11;                  // equivalente
//      scanf("%d", &dato.campo1);           // lee
//      scanf("%d", &pdato→campo1);          // equivalente
        strcpy ( pdato->campo2, "Campo 2" );
        printf("%d\n", dato.campo1);
        printf("%s\n", pdato->campo2);
        if(pdato == &dato.campo1) printf("El if() funciona\n");
        else                      printf("El if() no funciona\n");
}
```

Salida:
11
Campo 2
El if() funciona

Paso de estructura como parámetro de una función

Leer y escribir datos de estructuras

/* **09_04a.c** usar:
argumentos: Campos de estructura **alum1**
argumento: Estructura **alum2**
*/

```
#include<stdio.h>
#define MAX 10
#define LONGE 60
struct Alumno{
        int codigo;
        char nombre[LONGE];
};
void leer    (int *codigo, char nombre[], struct Alumno *alum);
void escribir(int  codigo, char nombre[], struct Alumno  alum);
int leerString(char *cc, int n);
void main(void) {

        struct Alumno alum1, alum2;
        leer(&alum1.codigo, alum1.nombre, &alum2); // atento al &
        escribir(alum1.codigo, alum1.nombre, alum2);
}
// Si la estructura es grande puede tomar mucho tiempo, pasar
alum2, por lo que se recomendaría pasar &alum2 (un solo
puntero)
```

/* **09_04b.c** usar:
argumento: Arreglo de estructuras **alum**
*/

```
void leer      (struct Alumno *alum, int *n);
void escribir(struct Alumno *alum, int n);
int leerString(char *cc, int n);
void main(void) {
        int n=2;
        struct Alumno alum[n];
        leer     (alum, n);    // atento al  &
        escribir(alum, n);
}
```

```
void leer(int *codigo, char nombre[], struct Alumno *alum) {
// alum es un apuntador, utilizará: alum->codigo

        printf("Ingrese 2 estructuras\n");

        printf("Código1: ");   scanf("%d", codigo);
        while(getchar()!=10);                   // limpia el buffer
        printf("Nombre1: "); leerString(nombre, LONGE);
        printf("Código2: ");  scanf("%d", &alum->codigo);
        while(getchar()!=10);                   // limpia el buffer
        printf("Nombre2: "); leerString(alum->nombre, LONGE);
}
```

```
void leer(struct Alumno *alum, int n) {
// alum es un apuntador, utilizará: alum->codigo
      int i;
      printf("Ingrese %d estructuras\n", n);
      for(i=0; i<n; i++, alum++){
         printf("Código%d: ", i+1); scanf("%d", &alum->codigo);
         while(getchar()!=10);       // para limpiar el buffer
         printf("Nombre%d: ", i+1);  leerString(alum->nombre, LONGE);
      }
}
```

```
void escribir(int codigo, char nombre[], struct Alumno alum) {

        printf("-------\ncodigo  nombre\n");
        printf( "%d\t\%s\n", codigo, nombre);
        printf( "%d\t\%s\n", alum.codigo, alum.nombre);
}
```

```
void escribir(struct Alumno *alum, int n){
        int i;
        printf("-------\ncodigo  nombre\n");
        for(i=0; i<n; i++, alum++)
             printf( "%d\t\%s\n", alum->codigo, alum->nombre);
}
```

```
int leerString(char *cc, int n){
   int c, m=0;
   while((c=getchar())!=10) if(m<n-1) *(cc+m++)= c;
   *(cc+m) = '\0';
   return m;
}
```

Salida:

```
Código1 : 12
Nombre1: José
Código1 : 44
Nombre2: Carlos
-------
Código   Nombre
12       José
 44      Carlos
```

Ordenar arreglo de estructuras

```
/* 90_05a.c:  Leer, ordenar (por nombre) y escribir arreglo de estructuras sin funciones
*/
```

```
/* 90_05b.c:  Leer, ordenar (por nombre y nota) y escribir arreglo de estructuras con funciones
*/
```

```
// Utilizar un arreglo de apuntadores que apuntan al arreglo de estructuras
```

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#define MAX 10
#define LONGE 60
int leerString(char *cc, int n);
typedef struct{                         // se coloca antes de todas las funciones para que sea global (visible de todas partes)
        int codigo;
        char nombre[LONGE];
        int nota;
} Alumno;
```

```
int leer         (Alumno    *alum, Alumno **palum);
void ordenar  (Alumno **palum, int n, int k);
void imprimir0(Alumno    *alum, int n);
void imprimir1(Alumno **palum, int n);
```

```
void main(void){
        Alumno alum[MAX], *palum[MAX], *pmin;
        int n=0, i, j, imin;
```

```
void main(void){
        Alumno alum[MAX], *palum[MAX];
        int n;
        n = leer(alum, palum);
        printf("Datos cargados");
```

```
                                        imprimir0(alum, n);             // antes de ordenar
                                        ordenar (palum, n, 1);          // ordenar por nombre
                                        printf("\nOrden: Nombre");
                                        imprimir1(palum, n);            // después de ordenar
                                        ordenar (palum, n, 2);          // ordenar por nota
                                        printf("\nOrden: Nota");
                                        imprimir1(palum, n);            // después de ordenar
                                }
                                int leer(Alumno *alum, Alumno **palum){
                                        int n = 0;
        printf ("Teclee Código y Nombre; para finalizar (código = 0):\n");
        do {
                printf ("Código: ");   scanf("%d", &alum[n].codigo);
                if(alum[n].codigo == 0) break;
                                                if(alum[n].codigo == 0) return n;
                while(getchar()!=10);                                   // limpia el buffer
                printf ("Nombre: "); leerString(alum[n].nombre, LONGE);
                printf ("Nota  : ");    scanf("%d", &alum[n].nota);
                palum[n] = &alum[n];
                n++;
        } while (n<MAX);
                                        return n;
                                }
                                void ordenar(Alumno **palum, int n, int k){
                                        Alumno *pmin;
                                        int i, j, imin, ik;
// ordenar
        for(i=0; i<n-1; i++){
                imin = i;
                pmin = palum[i];
                for(j=i+1; j< n; j++)
                   if(strcmp(pmin->nombre,palum[j]->nombre)>0){

                                imin = j;
                                pmin = palum[j];
                   }
                                                for(j=i+1; j< n; j++) {
                                                        if(k==1) ik = strcmp(pmin->nombre,palum[j]->nombre);
                                                        if(k==2) ik = pmin->nota > palum[j]->nota;
                                                        if(ik>0){
                                                                imin = j;
                                                                pmin = palum[j];
                                                        }
                                                }
               if(imin > i){
                        palum[imin] = palum[i];
                        palum[i] = pmin;
                }
        }
        printf("Datos cargados");
                                }
                                void imprimir0(Alumno *alum, int n){
                                        int i;
        printf ("\nCódigo  Nombre  Nota");
        for (i=0;i<n;i++)
        printf("\n%d\t%s\t%d", alum[i].codigo, alum[i].nombre, alum[i].nota);
        printf ("\n");
        printf("\nOrden: Nombre");
                                }
                                void imprimir1(Alumno **palum, int n){
                                        int i;
        printf ("\nCódigo  Nombre  Nota");
        for (i=0;i<n;i++)
                printf("\n%d\t%s\t%d", palum[i]->codigo, palum[i]->nombre, palum[i]->nota);
        printf ("\n");
}
int leerString(char *cc, int n){
   int c, m=0;
```

```
    while((c=getchar())!=10) if(m<n-1) *(cc+m++)= c;
    *(cc+m) = '\0';
    return m;
}
```

Salida:

```
Teclee Código, Nombre y Nota; para finalizar (código = 0):
Código :  10
Nombre: José
Nota   :  15
Código :  20
Nombre: Carlos
Nota   : 16
Código :  30
Nombre: Abel
Nota   : 14
Código:  0
```

Datos cargados:

Código	Nombre	Nota
10	José	15
20	Carlos	16
30	Abel	14

Orden: Nombre

Código	Nombre	Nombre
30	Abel	14
20	Carlos	16
10	José	15

Orden: Nota

Código	Nombre	Nota
30	Abel	14
10	José	15
20	Carlos	16

Unión

Hemos estudiado estructuras, por ejemplo:

```
struct  Ahorro {
    int    campo1[8];
    long campo2[6];
};
```

Una estructura contiene todos los valores de sus campos en tiempo de ejecución; su espacio es la suma de los espacios de todos sus campos.

Una unión tiene una definición parecida, por ejemplo:

```
union  [Ahorro] {
    int    campo1[8];
    long campo2[6];
} [miAhorro, ….];                    // variables de tipo union  Ahorro
```

Una union opera con un solo campo en un momento dado en tiempo de ejecución; su espacio es el máximo(campo1. Campo2, ...); por ejemplo:

Es responsabilidad del programador que se trabaje en la secuencia:

Asignar dato al campoN → usar datos del campoN
Asignar dato al campoM → usar datos del campoM
…..
Si: Asigna dato al campoN → usa dato del campoM   // el campoM no tiene el valor correcto
No se producen errores en compilación; es probable que tampoco en tiempo de ejecución; pero la lógica es incorrecta y no nos damos cuenta, lo más probable es que los resultados sean errados.

Definición, asignación y uso de los campos de union

La notación para uniones es parecida a la de estructuras, ejemplo:

```
//  09_06.c : Definir, leer e imprimir datos de una unión
#include<stdio.h>
typedef union{
        int  campo1[8];
        long campo2[2];
} Ahorro;                                   // el tamaño de esta union es 4 * 8 = 32 bytes
void main(void){
        Ahorro miAhorro, *pmiAhorro;      // define a las variables miAhorro y a pmiAhorro (puntero)
        pmiAhorro =  &miAhorro;           // pmiAhorro apunta a miAhorro;
        miAhorro.campo1[0] = 4;           // asigna valor a  campo1[0] = 4
        pmiAhorro->campo1[1] = 6;         // asigna valor a  campo1[1] = 6
        printf("%d\t %d\n",   pmiAhorro->campo1[0], pmiAhorro→campo1[1]);     // 4   6
        printf("%lu\t %lu\n", pmiAhorro->campo2[0], miAhorro.campo2[1]);          // error lógico.
}
```

Salida:

```
4          6
25769803780        4195392                    // error lógico: no se había asignado valor a campo2.
```

Ejercicios

1. Escriba un programa que:
   - Defina la estructura:
     - Estudiante {int codigo; char nombre[25];};
   - Defina dos arreglos tipo Estudiante con los datos:
     - est1[2] con los datos {{1, “Juan”}, {5, “José”}}
     - est2[3] con los datos {{3, “Rosita”}, {7, “María”}, {9, “Juana”}}
     - Como puede ver ambos arreglos están ordenados por código.
   - Imprima los datos de los dos arreglos ordenados por código; para los datos anteriores, la salida será:
     - 1 Juan
     - 3 Rosita
     - 5 José
     - 7 María
     - 9 Juana

   Sugerencia (no es obligatorio juntar los dos arreglos y ordenarlos), puede hacer:
   - Iteración:
     - Inicio: ponga en 0 a los índices de los arreglos est1 y est2;
     - Repita: Compare los códigos; escriba la estructura de menor código y avance su índice en 1;
     - Criterio de fin: Finaliza uno de los arreglos.
   - Proceso final: Escriba todos los datos remanentes del arreglo que no finalizó

2. Escriba un programa que:
   - Defina la estructura:
     - Profe {int codigo; char nombre[25]; int sueldo;};
   - Defina dos arreglos tipo Profe con los datos:
     - profe[5] con los datos: {{1, "Juan", 2000}, {3, "zoila", 1500}, {5, "Pedro", 1200}, {7, "Carlo", 1200}, {9, "Luis", 2000}}
     - nuevo[2] con los datos: {{3, "Zoila", 3000}, {7, "Carlos", 2000}};
       - nuevo[ ] actualiza los datos de profe que tengan el mismo código. Note que profe[] y nuevo[] están ordenados por código.
   - Imprima los Datos Iniciales y Datos actualizados, para el ejemplo será:

Datos Inciales			**Datos Actualizados**		
Código	Nombre	sueldo	Código	Nombre	sueldo
1	Juan	2000	1	Juan	2000
3	**zoila**	1500	3	**Zoila**	3000
5	Pedro	1200	5	Pedro	1200
7	**Carlo**	**1200**	7	**Carlos**	**2000**
9	Luis	2000	9	Luis	2000