ESTRUCTURAS

DECLARACION DE ESTRUCTURAS

La estructura es una colección de variables, la cual puede poseer distintos tipos de datos (a diferencia de los arreglos que solamente pueden tener un solo tipo de dato). Es un tipo de dato definido por el usuario. Son también conocidas como Registros. Ayudan a organizar y manejar datos complicados en programas debido a que agrupan diferentes tipos de datos a los que se los trata como una sola unidad en lugar de ser vistas como unidades separadas.

La declaración de estructuras en programas C, es un nuevo tipo de datos denominado tipo Estructura y es posible declarar una variable de este tipo una vez que declare o defina el tipo estructura de la siguiente manera:

```
 Struct    Identificador_ tipo_estructura
{          Tipo miembro_1;   /*Declaración de los miembros*/
….
           Tipo miembro_n;

};
```

En la definición del tipo de estructura, se especifican los elementos que la componen así como sus tipos. Cada elemento es llamado miembro (similar a un campo de un registro).

Por ejemplo:

```
struct alumno
{          int legajo;
           char apellido[20];
           float promedio;
           };
```

Después de definir un tipo estructura, se puede declarar una o más variables de ese tipo de la siguiente forma:

```
Struct    Identificador_ tipo_estructura  var1,…,varn;
```

Esta sería una definición de variables con esta estructura; o también se pueden definir la estructura y las variables registro que serán de este tipo al mismo tiempo:

Ejemplo de estructura

```
struct alumno
{          int legajo;
           char apellido[20];
           float promedio;
           } var_registro_ alum1, var_registro_ alum2   ;
```

El Acceso a los Miembros de la Estructura es por medio del operador Punto de la siguiente manera:

Ejemplo de acceso a miembros:

var_registro_ alum1.promedio=8.5;

var_registro_ alum1.legajo=341

Los miembros de las Estructuras también se pueden Inicializar individualmente o bien, todos simultáneamente. La inicialización individual es por medio del operador punto, para la estructura completa es con caracteres llaves y los valores de los miembros separados por coma.

var_registro_ alum1{341,"Garcia",8.5};

UN EJEMPLO MÁS COMPLETO

Declaramos el tipo estructura alumno

```
struct alumno
{       char nombre[20];
        char apellido[20];
        int edad;
        int año;
        int nota[4];
        float promedio;
        };
```

Con la instrucción **struct alumno** declaro un tipo de dato llamado alumno, que contendrá variables de tipo enteras, de punto flotante y cadenas de caracteres.

Otra forma es la siguiente.

```
struct alumno
{       char nombre[20];
        char apellido[20];
        int edad;
        int año;
        int nota[4];
        float promedio;
} registro1, registro2, registro3;
```

Aquí se ve que se declara la estructura y se aprovecha para declarar las variables que corresponden a **alumno**, y serán registro1, registro2 y registro3. En este caso estas variables serán globales.

Todas las variables que se encuentran entre las llaves las llamaremos miembros de la estructura, por lo que las variables **registro1**, **registro2** y **registro3** cada una contendrá 6

miembros diferentes. Pudiéndose acceder a cada uno de ellos a través del operador punto (**.**), es decir, se escribirá el nombre de la variable **struct alumno** seguido por un punto y el nombre del miembro al que nosotros queremos acceder. Por ejemplo, queremos obtener el promedio del siguiente alumno:

Nombre: Victor
Apellido: Perez
Edad: 20
Año que cursa: 1
notas: 8, 6, 10, 6

El programa sería el que se observa a continuación:

```
# include <stdio.h>

struct alumno
{       char no [25];
        char ap [10];
        int edad;
        int anio;
        int nota[4];
        float prom;
};

void main ( )
{       int i, acum = 0;
        struct alumno R1;
        R1 = {“Victor”, “Perez”, 20, 1};
        for ( i = 0; I < 4 ; i++)
        {       printf (“Ingrese la nota %d: “,  i+1 );
                scanf (“%d”, &R1.nota[i] );
                acum = acum + R1.nota[i];
        }
        R1.prom = (float) acum / 4;
        printf ( “El promedio del alumno %s, %s es %f”, R1.ap, R1.no, R1.prom);
}
```

Con la línea `-< R1 = {“Victor”, “Perez”, 20, 1}; >-` inicializamos los distintos miembros de la estructura `alumno`; luego mediante un ciclo `for` inicializamos los distintos elementos del miembro `nota` para después calcular, imprimir y almacenar el `promedio` de las distintas notas.

Si se desea, se puede realizar un arreglo de estructuras, con lo cual se podría juntar a todos los alumnos de un colegio determinado, en el registro **R[i]** en lugar de realizar un registro **Rx** por cada alumno de ese año, en donde la **x** representa el número del alumno. La declaración para un colegio con 500 alumnos será entonces:

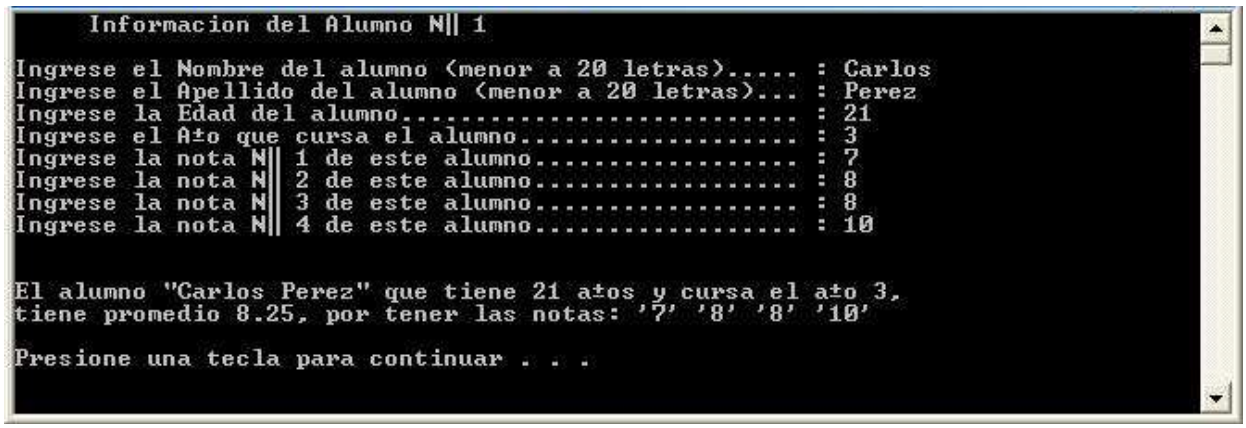
Struct alumno R[500];

En lugar de

Struct alumno R1, R2, R3, ... R498, R499, R500;

Veamos un ejemplo completo:

```c
#include <stdio.h>
#include <stdlib.h>

#define t 500

void main(void)
{
int i, j;
struct alumno
   {
      char nombre[20];
      char apellido[20];
      int edad;
      int anio;
      int nota[4];
      float promedio;
      } reg[t];

for (i=0; i<t; i++)
   {
   reg[i].promedio = 0;
   printf ("     Informacion del Alumno Nº %d\n\n", i+1);
   printf ("Ingrese el Nombre del alumno (menor a 20 letras)..... : ");
   scanf ("%s",&reg[i].nombre[0]);
   printf ("Ingrese el Apellido del alumno (menor a 20 letras)... : ");
   scanf ("%s",&reg[i].apellido[0]);
   printf ("Ingrese la Edad del alumno..... : ");
   scanf ("%d",&reg[i].edad);
   printf ("Ingrese el Año que cursa el alumno..... : ");
   scanf ("%d",&reg[i].anio);
   for (j=0; j<4; j++)
      {
      printf ("Ingrese la nota Nº %d de este alumno..... : ", j+1);
      scanf ("%d",&reg[i].nota[j]);
      }
   for (j=0; j<4; j++)
      reg[i].promedio = reg[i].promedio + reg[i].nota[j];
   reg[i].promedio = reg[i].promedio / 4;
   }

for (i=0; i<t; i++)
   {
   printf ("\n\nEl alumno \"%s %s\" ", reg[i].nombre, reg[i].apellido);
   printf ("que tiene %d años y cursa el año %d,\n", reg[i].edad, reg[i].anio);
   printf ("tiene promedio %.2f, por tener las notas: ", reg[i].promedio);
   printf ("'%d' '%d' ", reg[i].nota[0], reg[i].nota[1]);
   printf ("'%d' '%d' \n\n", reg[i].nota[2], reg[i].nota[3]);
   }

system("PAUSE");
}
```



Un aspecto muy importante a tener en cuenta, es que no se pueden comparar estructuras entre si, lo que si se puede es comparar un miembro de una con el mismo miembro de otra, esto es debido que los miembros no se almacenan en forma consecutiva en memoria sino que puede haber espacios de memoria vacíos entre un miembro y otro de una misma estructura. Otro aspecto es que al ser la estructura un tipo de dato, se puede pasar y recibir como argumento de una función como si fuera un tipo de datos como los que ya fueron estudiados.

La principal ventaja de usar estructuras reside en poder manejar dinámicamente el espacio en memoria y, según la cantidad de registros que se quieran introducir, se ocupará mas o menos memoria y no un valor fijo como ocurre cuando se trabaja con arreglos, además de poder borrar cualquier registro que no nos interese mas tenerlo con el consiguiente ahorro de memoria. Pero hay que tener mucho cuidado por que la cantidad de memoria que se necesita variará en tiempo de ejecución y compilación.

Una estructura puede contener como miembro, un puntero para que apunte a una estructura

```
struct alumno
{       char nombre[20];
        char apellido[20];
        int edad;
        int anio;
        int nota[4];
        float promedio;
        struct alumno *puntero;
};
```

Al introducir la sintaxis **struct alumno *puntero** se incorpora un miembro que contendrá un puntero que apuntara hacia otra estructura del mismo tipo, y aquí a cada estructura se le llama nodo.

La instrucción para generar dinámicamente un nodo en C es

```
puntero= malloc (sizeof (struct alumno));
```

En donde la función **sizeof** toma como argumento el tipo de dato y devuelve un valor entero que representa la cantidad de bytes que utiliza ese dato y la función **malloc** toma como argumento la cantidad de bytes y devuelve la dirección de memoria en donde genero el nodo, devolviendo un puntero con el valor null si no encuentra memoria disponible.

Para eliminar el nodo y liberar espacio en memoria, se utiliza la instrucción:

```
free (puntero);
```

Donde la función **free** toma como argumento la dirección de memoria en donde está almacenado el nodo que no se utilizara más en lo que resta de la ejecución del programa.

Para realizar el manejo dinámico de memoria, se cuenta con:

- **Colas**
- **Listas simples**
- **Listas dobles**
- **Árbol binario**

Estos temas se desarrollaran en Informática II.

Trabajo Práctico numero 17

Actividad 1:

Diseñar un programa modificando el ejemplo anterior que ingrese los datos personales correspondiente a cada alumno de su curso de primer año de informática y determinar que alumnos promocionaron, que alumnos regularizaron y quienes no.

Hacer diagrama de flujo, pseudocódigo y codificar en C.

Actividad 2:

Diseñar un programa modificando el ejemplo anterior que ingrese los datos correspondientes a todos los alumnos de primer año de informática utilizando un arreglo de estructuras y muestre un menú donde se puedan seleccionar estas distintas opciones:

(1)-Ingresar datos (2)-Mostrar datos por alumno (3)-estadísticas que muestren cantidad de alumnos inscriptos, alumnos regulares, porcentaje de aprobación, promocionados por curso y ejecutar las operaciones de acuerdo a la opción elegida.

Hacer diagrama de flujo, pseudocódigo y codificar en C.