Estructuras

Definición

Cuando se declara una variable en un programa, se está definiendo un espacio de memoria en la que se podrá almacenar un solo valor. Por ejemplo, cuando colocamos en un programa la sentencia `int a;` estamos definiendo la variable entera `a`, la cual sólo podrá almacenar un valor en un instante dado.

Por otro lado, cuando definimos un arreglo estamos definiendo un conjunto de variables, todas identificadas por un mismo nombre, diferenciándose cada una por uno índice, así por ejemplo cuando en un programa escribimos la sentencia `int a[5];` estamos definiendo las variables enteras `a[0], a[1], a[2], a[3] y a[4]`. Hemos visto en los capítulos anteriores la cantidad de aplicaciones que podemos desarrollar con este tipo de dato, sin embargo, a pesar de ello los arreglos tienen un inconveniente, que todos elementos de un arreglo tienen el mismo tipo de dato. Esto quiere decir que, en el ejemplo, en el arreglo `a` sus elementos sólo pueden albergar valores enteros. Si quisiéramos almacenar por medio de un programa una lista de personas en donde por cada una tenemos un código, un nombre y un sueldo, para poder hacerlo tendríamos que definir un arreglo por cada uno de los datos, esto es un arreglo de enteros para el código, un arreglo de cadenas de caracteres para el nombre y uno de punto flotante para el sueldo.

Las `"estructuras"` como se denominan en el lenguaje C++, permiten definir variables que pertenecen también a la categoría de variables estructuradas, en este sentido al declarar una estructura estaremos definiendo, como en el caso de los arreglos, un conjunto de datos, todos identificados por el mismo nombre, pero con la diferencia que cada elemento del conjunto puede ser de diferente tipo. La forma en que se diferenciará un elemento de otro del conjunto ya no se hará por medio de índices, sino que se le asignará a cada elemento un nombre. Cada elemento de una estructura se denominará campo. Las estructuras son un tipo primitivo de datos, su evolución ha dado lugar a las `"Clases"` definidas en la programación orientada a objetos (POO) e implementadas en C++.

Implementación de una estructura

La implementación de una estructura no es tan simple como en el caso de los arreglos, y por lo general tampoco se van a manejar con un simple puntero. Para implementar una estructura primero debemos definir un `"nuevo tipo de dato"`. Este tipo de dato describirá las características de los campos que contendrá la estructura. Luego, este tipo de dato se empleará para declarar variables.

Hemos dicho muchas veces que programar profesionalmente implica siempre la reutilización del código, una función implementada en un programa debe poder ser utilizada en otro sin la necesidad de volverla a implementar. Esta premisa la debemos extender al uso de estructuras, el tipo de dato que define una estructura debe ser implementado en un archivo independiente de cualquier módulo de un proyecto, incluso

no debe mezclarse en el mismo módulo donde se definan encabezados de funciones u otras estructuras. Cuando un módulo requiera definir un variable de tipo estructura tendrá que emplear la cláusula **#include** para anexar el archivo, donde se defina el tipo de dato.

Entonces, si queremos desarrollar un programa que permita almacenar en una variable el DNI, el nombre y el sueldo de un empleado, lo que tenemos que hacer es crear un archivo de encabezados (.h) que tenga como nombre algo muy relacionado con el tipo de dato que vamos a definir, por ejemplo "**empeado.h**" y en él definir el tipo de dato que contenga los campos con los que queramos trabajar, como se muestra a continuación:

```
/* Archivo:  Empleado.h */
#ifndef EMPLEADO_H
#define EMPLEADO_H

    struct Empleado{
        int dni;
        char nombre[60];
        double sueldo;
    };

#endif /* EMPLEADO_H */
```

Como se puede observar, el tipo de dato que acabamos de definir, que se denominará **struct Empleado**. Fíjese que el identificador "Empleado" empieza con una mayúscula, esto lo diferenciará de una variable o función. También se puede observar que, como cada dato de la estructura puede tener un tipo de dato diferente, entonces habrá que describir cada uno de los campos colocando el tipo de dato y el nombre del identificador, como si fuera una variable.

En resumen, de esa forma hemos definido el tipo de dato denominado **struct Empleado** el cual estará compuesto por tres campos: el primero (**dni**) podrá almacenar un valor entero, el segundo (**nombre**) guardará una cadena de caracteres y el tercero (**sueldo**) un valor de punto flotante.

Cuando en un programa se requiera definir una variable de este tipo, se deberá incluir el archivo donde esté definido el tipo de datos y a partir de allí definir la o las variables que se requieran de la siguiente manera:

```
#include <iostream>
#include <iomanip>
Using namespace std;
#include <cstring>
#include "Empleado.h"

int main(int argc, char** argv) {

    struct Empleado empleado; <===
    …
}
```

Observe que en la instrucción se ha declarado la variable **empleado** del tipo **struct Empleado**.

La variable **empleado**, de tipo **struct Empleado**, podrá manipular tres elementos, éstos se manejarán de la siguiente manera:

```
empleado.dni = 7722099;
strcpy(empleado.nombre, "Ana Cecilia");
empleado.sueldo = 3789.50;
```

Como se puede apreciar todas los campos inmersos en el tipo de dato se identifican por el mismo nombre (**empleado**) sin embargo cada uno se diferencia del otro por el nombre del campo. El nombre de la variable y el del campo se deben separar por un punto. Un campo de la estructura, como por ejemplo **empleado.dni**, se puede manipular como cualquier otra variable sin restricción alguna. Así, se le podrá asignar valores directamente como en el ejemplo anterior y también desde la consola o de algún archivo, se le podrá emplear en expresiones, etc., en fin, podrá utilizarla como cualquier variable común y corriente.

La desventaja con respecto a los arreglos es que no se puede generalizar el uso de los campos, esto es, que en el caso de los arreglos se puede emplear variables para manipular los índices como por ejemplo **a[i]** := 51;, en donde si **i** vale 3 nos estaremos refiriendo a **a[3]**, y si **i** vale 5 a **a[5]**; en el caso de las estructuras **NO** se podrá hacer algo como **empleado.i**, pensando que si i vale 'sueldo' nos estaremos refriendo al campo sueldo, el hacer esto dará como resultado un error de compilación.

Otra ventaja del uso de estructuras es que, a diferencia de los arreglos, se puede asignar en una sola operación todos los campos de una estructura a otra como en el ejemplo siguiente:

```
#include <iostream>
#include <iomanip>
using namespace std;
#include <cstring>
#include "Empleado.h"

int main(int argc, char** argv) {
    struct Empleado empleado, trabajador;

    empleado.dni = 775566;
    strcpy(empleado.nombre,"Ana Cecilia");
    empleado.sueldo = 5680.90;
    //  Asignación directa:
    trabajador = empleado;
    // Ahora manejamos la nueva estructura
    cout.precision(2);
    cout<<fixed;
    cout<<left<<setw(10)<<"Codigo :"<<right<<setw(10)<<trabajador.dni<<endl;
    cout<<left<<setw(10)<<"Nombre :"<<setw(30)<<trabajador.nombre<<endl;
    cout<<left<<setw(10)<<"Sueldo :"<<right<<setw(10)<<trabajador.sueldo<<endl;

    return 0;
}
```

Al ejecutar el programa obtendremos el siguiente resultado:

```
Codigo :     775566
Nombre : Ana Cecilia
Sueldo :    5680.90
```

Como se puede observar con una sola asignación (**trabajador=empleado;**) se pueden pasar todos los campos de una variable de tipo estructura a otra del mismo tipo.

Esta operación se puede realizar con cualquier tipo de estructura, sin embargo, en el lenguaje C++ hay que diferenciar muy claramente **"lo que se puede hacer"** de **"lo que se debe hacer"**, de lo contrario solo obtendremos dolores de cabeza y programas que se caen. En este sentido debemos saber que la operación que hicimos en el programa anterior se ha realizado satisfactoriamente porque los campos que se han definido en base a tipos de datos simples (int dni, double sueldo, etc.) y a arreglos estáticos (char nombre [60]). Si en la estructura hubiéramos definido el campo nombre como un puntero (char *nombre), la asignación se hubiera producido (sin que el sistema detecte un error) pero eso nos causaría una serie de problemas debido a que los campos nombre de las variables que intervienen en la asignación terminarían apuntando al mismo espacio de memoria y por lo tanto si en alguna instrucción del programa se decide cambiar el contenido del campo nombre de una de las variables la otra también seria afectada, perdiendo su valor.

El siguiente programa ilustra este concepto:

```
#ifndef EMPLEADO2_H
#define EMPLEADO2_H

    struct Empleado2{
        int dni;
        char *nombre;
        double sueldo;
    };

#endif /* EMPLEADO2_H */
```

```
#include <iostream>
#include <iomanip>
using namespace std;
#include <cstring>
#include "Empleado.h" // Reutilizado del programa anterior
#include "Empleado2.h"

int main(int argc, char** argv) {
    struct Empleado empleado, trabajador;
    struct Empleado2 empleado2, trabajador2;

    // Trabajamos con la primera estructura:
    empleado.dni = 775566;
    strcpy(empleado.nombre,"Ana Cecilia");
    empleado.sueldo = 5680.90;
    //  Asignación directa:
    trabajador = empleado;
    // modificamos empleado:
    strcpy(empleado.nombre,"Valentina Naomi");
    // Ahora imprimimos ambas estructuras
    cout<<left<<"Estructura 1:"<<endl;
    cout.precision(2);
    cout<<fixed;
    cout<<left<<setw(10)<<"Codigo [E1]:"<<right<<setw(10)<<empleado.dni<<endl;
    cout<<left<<setw(10)<<"Nombre [E1]:"<<setw(30)<<empleado.nombre<<endl;
    cout<<left<<setw(10)<<"Sueldo [E1]:"
                        <<right<<setw(10)<<empleado.sueldo<<endl;
    cout<<endl;
    cout<<left<<setw(10)<<"Codigo [T1]:"<<right<<setw(10)<<trabajador.dni<<endl;
    cout<<left<<setw(10)<<"Nombre [T1]:"<<setw(30)<<trabajador.nombre<<endl;
    cout<<left<<setw(10)<<"Sueldo [T1]:"
              <<right<<setw(10)<<trabajador.sueldo<<endl;
    cout<<endl;
    // Trabajamos con la segunda estructura:
    empleado2.dni = 775566;
    empleado2.nombre = new char[60];
```

```
    strcpy(empleado2.nombre,"Ana Cecilia");
    empleado2.sueldo  = 5680.90;
    // Asignación directa:
    trabajador2 = empleado2;
    // modificamos empleado:
    strcpy(empleado2.nombre,"Valentina Naomi");
    // Ahora imprimimos ambas estructuras
    cout<<left<<setw(10)<<"Codigo [E1]:"<<right<<setw(10)<<empleado2.dni<<endl;
    cout<<left<<setw(10)<<"Nombre [E1]:"<<setw(30)<<empleado2.nombre<<endl;
    cout<<left<<setw(10)<<"Sueldo [E1]:"
                        <<right<<setw(10)<<empleado2.sueldo<<endl;
    cout<<endl;
    cout<<left<<setw(10)<<"Codigo [T1]:"
                        <<right<<setw(10)<<trabajador2.dni<<endl;
    cout<<left<<setw(10)<<"Nombre [T1]:"<<setw(30)<<trabajador2.nombre<<endl;
    cout<<left<<setw(10)<<"Sueldo [T1]:"
                        <<right<<setw(10)<<trabajador2.sueldo<<endl;

    return 0;
}
```

Al ejecutar el programa obtendremos lo siguiente:

```
Estructura 1:
Codigo [E1]:      775566
Nombre [E1]: Valentina Naomi
Sueldo [E1]:     5680.90

Codigo [T1]:      775566
Nombre [T1]: Ana Cecilia
Sueldo [T1]:     5680.90

Estructura 2:
Codigo [E2]:      775566
Nombre [E2]: Valentina Naomi
Sueldo [E2]:     5680.90

Codigo [T2]:      775566
Nombre [T2]: Valentina Naomi
Sueldo [T2]:     5680.90
```

Solo se afecta el campo de la variable empleado.

Los campos de ambas variables son afectados.

Como podemos observar, en la primera estructura solo cambió el nombre de la estructura de empleado, mientras que en la segunda cambiaron ambas estructuras. Será el programador el que decida qué es lo que conviene hacer.

La asignación de estructuras puede hacerse también a nivel de funciones, esto quiere decir que una función puede también devolver una estructura completa. El siguiente ejemplo se puede ver cómo se puede asignar todos los valores de los campos de una estructura en una sola operación por medio de una función:

```
#ifndef FUNCIONESDEESTRUCTURAS_H
#define FUNCIONESDEESTRUCTURAS_H
#include "Empleado.h"

     struct Empleado leerDatos(void);
     void imprimirDatos(struct Empleado);

#endif /* FUNCIONESDEESTRUCTURAS_H */
```

```
#include "Empleado.h"
#include "FuncionesDeEstructuras.h"
```

```
int main(int argc, char** argv) {
     struct Empleado empleado;

     empleado = leerDatos();
     imprimirDatos(empleado);

     return 0;
}
```

```
#include <iostream>
#include <iomanip>
Using namespace std;
#include "Empleado.h"

struct Empleado leerDatos(void){
     struct Empleado auxiliar;

     cout<<"Ingresamos los datos de la estructura:"<<endl;
     cout<<"DNI: ";
     cin >> auxiliar.dni;
     while(cin.get ()!= '\n');
     cout<<"Nombre: ";
     cin.getline(auxiliar.nombre,60);
     cout<<"Sueldo: ";
     cin >> auxiliar.sueldo;

     return auxiliar;
}

void imprimirDatos(struct Empleado auxiliar){
     cout.precisio(2);
     cout<<fixed;
     cout<<endl;
     cout<<"Datos de la estructura:"<<endl;
     cout<<left<<setw(8)<<"Codigo:"<<right<<setx(10)<<auxiliar.dni<<endl;
     cout<<left<<setw(8)<<"Nombre:"<<setw(35)<<auxiliar.nombre<<endl;
     cout<<left<<setw(8)<<"Sueldo:"<<right<<setw(10)<< auxiliar.sueldo<<endl;
}
```

Al ejecutar el programa obtendremos lo siguiente:

```
Ingresamos los datos de la estructura:
DNI: 64738190
Nombre: Juan Perez
Sueldo: 3546.27

Datos de la estructura:
Codigo:   64738190
Nombre: Juan Perez
Sueldo:    3546.27
```

En el programa se puede observar dos situaciones en las que se produce una asignación entre dos estructuras: la primera en la instrucción ***return*** de la función ***leerDatos***, (***return auxiliar;***) que retorna la estructura ***auxiliar*** para luego ser recibida y asignada en la función ***main*** (***empleado = leerDatos();***), la segunda en la función ***main*** al pasar como parámetro por valor la estructura empleado a la función ***imprimirDatos*** (***imprimirDatos(empleado);***) y ser recibida luego por el argumento ***auxiliar***.

Debe recordar que esto se puede hacer sin inconvenientes porque ningún campo ha sido definido como un puntero.

Paso por referencia de estructuras

Otra forma, más simple de realizar esta operación es empleando un parámetro por referencia, como se muestra a continuación:

```
#ifndef FUNCIONESDEESTRUCTURAS_H
#define FUNCIONESDEESTRUCTURAS_H
#include "Empleado.h"

     void leerDatos(struct Empleado &);
     void imprimirDatos(struct Empleado);

#endif /* FUNCIONESDEESTRUCTURAS_H */
```

```
#include "Empleado.h"
#include "FuncionesDeEstructuras.h"

int main(int argc, char** argv) {
     struct Empleado empleado;

     leerDatos(empleado);
     imprimirDatos(empleado);

     return 0;
}
```

```
#include <iostream>
#include <iomanip>
Using namespace std;
#include "Empleado.h"

void leerDatos(struct Empleado &auxiliar){
     cout<<"Ingresamos los datos de la estructura:"<<endl;
     cout<<"DNI: ";
     cin >> auxiliar.dni;
     while(cin.get ()!= '\n');
     cout<<"Nombre: ";
     cin.getline(auxiliar.nombre,60);
     cout<<"Sueldo: ";
     cin >> auxiliar.sueldo;
}

void imprimirDatos(struct Empleado auxiliar){
     cout.precisio(2);
     cout<<fixed;
     cout<<endl;
     cout<<"Datos de la estructura:"<<endl;
     cout<<left<<setw(8)<<"Codigo:”<<right<<setx(10)<<auxiliar.dni<<endl;
     cout<<left<<setw(8)<<"Nombre:”<<setw(35)<<auxiliar.nombre<<endl;
     cout<<left<<setw(8)<<"Sueldo:”<<right<<setw(10)<< auxiliar.sueldo<<endl;
}
```

La ejecución del programa es idéntica al anterior.

Consejo para una buena práctica de programación

Las estructuras son tipos de datos complejos y por lo tanto la cantidad de memoria que se requiere para almacenar una variable de tipo estructura es también grande. Si recordamos, cuando se pasa un parámetro por valor a una función se produce una copia de la variable y por lo tanto se debe duplicar el espacio de memoria para almacenar el parámetro. Si la variable resulta ser una estructura, el proceso se torna muy ineficiente, no solo porque se debe duplicar el espacio para la estructura, que sabemos que es grande, sino que también el tiempo que toma realizar la copia de todos los campos es significativo y si la función es llamada muchas veces en el programa, el tiempo de respuesta del mismo puede incrementarse significativamente. Este

problema no se presenta con los arreglos porque cuando se pasa un arreglo como parámetro lo que se envía es solo la dirección de inicio del arreglo, ya que un arreglo es un puntero y las estructuras no.

Para corregir este problema es recomendable que ***siempre*** se pase por referencia una variable de tipo estructura, de modo que solo se envíe la dirección de la variable.

Sin embargo, como el paso por referencia implica que la variable que pasa como parámetro será modificada por la función es bueno que se diferencie la implementación en el caso que la variable no vaya a ser modificada por la función. A continuación, mostramos cómo hacer esta diferencia.

Cuando se desea modificar la estructura en la función:

```
tipo funcion(struct Empleado &auxiliar){…
```

Cuando no se va a modificar la estructura en la función:

```
tipo funcion(const struct Empleado &auxiliar){…
```

Ambas son pasos por referencia, por lo que la invocación a la función es igual en ambos casos (`función (empleado);`).

Estructuras manejadas desde punteros

Un puntero, como hemos visto anteriormente, puede ser definido de cualquier tipo de dato, en particular podemos definir un puntero de tipo estructura.

```
struct Empleado *pt;
```

Entiéndase aquí que `pt` es un puntero que puede apuntar a una estructura, `pt` no es una estructura.

Entonces, como cualquier puntero, debemos asignarle un espacio de memoria al puntero para poder trabajar con la variable referenciada, esta operación se realiza de la misma forma que en el caso de cualquier puntero.

```
pt = new struct Empleado;
```

Luego, para llegar a la variable referenciada solo habrá que anteponerle el `*` al putero.

```
*pt
```

Sin embargo, si lo que queremos es utilizar uno de los campos de la estructura, allí se presenta un inconveniente, ya que si escribimos la siguiente orden:

```
*pt.dni = 63736219;
```

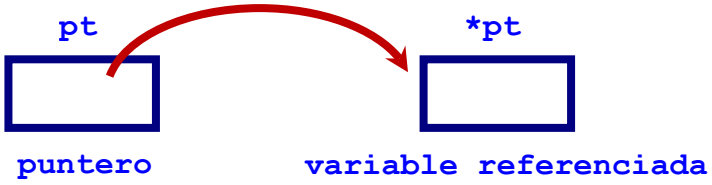
Estaríamos cometiendo un grave error conceptual ya que el campo `dni` no es un puntero, es una variable entera, por lo que, si le agregamos el `*`, el programa no compilará.

La expresión correcta para realizar esa orden sería la siguiente:

```
(*pt).dni = 63736219;
```

Los paréntesis son necesarios porque el puntero es `pt` no el campo `dni`.

Ahora, si es cierto que no es complicado emplear los paréntesis, en estructuras más complejas esto puede tornase un poco molesto e inducir a cometer algún error. Por este motivo el lenguaje C++ ha establecido una forma alternativa que simplifica mucho el manejo de los campos a través de punteros, esta forma está inspirada en el siguiente gráfico:



Entonces cuando se quiera utilizar un campo desde un puntero, en lugar de emplear el punto (.) emplearemos una "**flecha**" simulada con los caracteres '**-**' y '**>**', esto es, las expresiones que emplearemos serán las siguientes:

```
pt->dni = 63736219;}
strcpy(pt->nombre,"Ana Cecilia");
pt->sueldo = 3756.25;
```

De esta manera se elimina el asterisco (*) y los paréntesis.

Situaciones complejas en la implementación de una estructura

Una de las características más importante de la implementación de una estructura es la que un campo puede ser definido empleando cualquier tipo de dato. Esto quiere decir que un campo puede ser un tipo de dato simple (**int, double, char,** etc.) como estructurado (arreglo, estructura, etc.) y por lo tanto no hay límites. Por esta razón se deben analizar los diferentes casos que se puedan presentar a fin que se aprecie cómo se puedan resolver.

Estructuras anidadas:

Esta situación se presenta cuando existe un tipo de dato de tipo estructura y este tipo se emplea para definir el campo de otro tipo de dato estructura. El siguiente ejemplo muestra esta situación:

```
#ifndef ESTRUCTURAFECHA_H
#define ESTRUCTURAFECHA_H

struct Fecha{
     int dd;
     int mm;
     int aa;
};
#endif /* ESTRUCTURAFECHA_H */
```

```
#ifndef ESTRUCTURAEMPLEADO_H
#define ESTRUCTURAEMPLEADO_H
#include "EstructuraFecha.h"

struct Empleado{
     int dni;
     char nombre[60];
     double sueldo;
     struct Fecha fechaDeIngreso;
};
```

Aquí podemos apreciar que el campo **fechaDeIngreso** está definido como un tipo de dato estructura (**struct Fecha**) esto ocasionará que cuando se desee asignar valores a este campo, se le tendrá que manejar precisamente como una estructura, esto es:

```
empleado.dni = 77556688;
empleado.nombre = "Ana Cecilia";
empleado.fechadeIngreso.dd = 1;
empleado.fechadeIngreso.mm = 2;
empleado.fechadeIngreso.aa = 2003;
empleado.sueldo = 5680.90;
```

De esa misma forma, se pueden incrementar los niveles de anidación de modo que un campo sea definido con otro tipo estructura que a su vez tiene campos definidos como estructuras. En esos casos sólo habrá que incrementar el nombre de la variable, recordando que hay que emplear el punto (.) para separar cada nivel.

Arreglos de tipo estructura y campos de tipo arreglo:

Aunque esto parezca un trabalenguas, estas situaciones se pueden presentar por lo que será muy importante analizar la nomenclatura que se emplee para no cometer errores.

El primer caso que analizaremos es el del campo como arreglo en el siguiente ejemplo:

```
#ifndef ESTRUCTURAALUMNO_H
#define ESTRUCTURAALUMNO_H

struct Alumno{
     int codigo;
     char nombre[60];
     int notas[20];
};
#endif /* ESTRUCTURAALUMNO_H */
```

En este caso hemos defiendo un campo denominado **notas**, que guardará las notas parciales de un alumno en un curso determinado. Entonces para poder asignar por ejemplo la tercera nota del curso a la variable deberemos tomar en cuenta que el arreglo es el campo **notas** y no la estructura por lo que el índice del arreglo debe estar ligado al campo y no a la estructura, esto es:

```
#include <cstring >
#include "EstructuraAlumno.h"

int main(int argc, char** argv) {
     struct Alumno alumno;

     alumno.codigo = 20190105;
     strcpy(alumno.nombre, "Naomi Valentina");
     alumno.notas[3] = 18;

     return 0;
}
```

Por otro lado, qué pasa si queremos definir un arreglo en el que cada elemento sea un registro. En este caso, se tiene que tener mucho cuidado al manejar los elementos, ya que hay que tener claro quién es el arreglo. La forma de hacerlo será de la siguiente manera:

```
#include <cstring>
#include "EstructuraAlumno.h"
```

```
int main(int argc, char** argv) {
     struct Alumno alumnos[30];

     alumnos[7].codigo = 20171010;
     strcpy(alumnos[7].nombre,"Naomi Valentina");
     alumnos[7].notas[3] = 18;

     return 0;
}
```

Ordenar un arreglo de estructuras

En este ejemplo veremos cómo organizar la información en un arreglo de estructuras en lugar de un grupo de arreglos como se hizo en capítulos anteriores, aquí se podrá apreciar las ventajas que esto trae, como por ejemplo se verá que es una ventaja a la hora de ordenar los datos que se puede manejar toda la información de una persona como una unidad, esto hará que se intercambien los datos de manera más sencilla.

Los datos serán leídos desde un archivo del tipo CSV como el que se mostrará a continuación. Se contempla que la información del archivo corresponde a la de alumnos de un curso, esto es que se tiene su código, nombre y sus notas, un alumno puede tener hasta 20 notas y la cantidad no siempre es la misma en cada alumno. El programa leerá estos datos y los almacenará en estructuras, luego calculará los promedios correspondientes, que también serán almacenados en el registro y finalmente ordenará estos datos en función al promedio obtenido y los mostrará en un reporte:

```
19992345,Juan Lopez,12,14,10,8
20001010,Maria Ruiz,10,12,11,9,9,5,11,10
20101234,Ana Roncal,15,16,12,14,18,15
20002324,Pedro Sanchez,14,15
20020107,Paula Gomez,17,19,20,12,15,16,18
19971313,Carlos Castro,2,5,9,10,6
20041003,Alexandra Neyra,18,15,17,16,18,19,20
…
```

Programa:

```
#ifndef ESTRUCTURAALUMNO_H
#define ESTRUCTURAALUMNO_H

    struct Alumno{
         int codigo;
         char nombre[60];
         int notas[20];
         int numNotas;
         double promedio;
    };

#endif /* ESTRUCTURAALUMNO_H */
```

```
#include "EstructuraAlumno.h"
#include "FuncionesDeEstructuras.h"

int main(int argc, char** argv) {
    struct Alumno alumnos[50];

    int numAlum;
    leerDatos(alumnos, numAlum, "alumnos.csv");
    calcularPromedios(alumnos, numAlum);
    ordenarDatos(alumnos, numAlum);
    emitirReporte(alumnos, numAlum);

    return 0;
}
```

```
#ifndef FUNCIONESDEESTRUCTURAS_H
#define FUNCIONESDEESTRUCTURAS_H

    void leerDatos(struct Alumno *, int &, const char*);
    void leerAlumno(ifstream &, struct Alumno &);
    void calcularPromedios(struct Alumno *, int);
    void ordenarDatos(struct Alumno *, int);
    void emitirReporte(struct Alumno *, int);
    void cambiar(struct Alumno &, struct Alumno &);

#endif /* FUNCIONESDEESTRUCTURAS_H */
```

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
#include <cstring>
#include "EstructuraAlumno.h"
#include "FuncionesDeEstructuras.h"
void leerDatos(struct Alumno *alumnos, int &numAlum, const char *nombArch){
    // Se debe apreciar que alumnos es un arreglo, por eso
    // se usa el punto (.) y no la flecha  (->)
    ifstream archAlum(nombArch,ios::in);
    if(not archAlum.is_open()){
        cout << "ERROR: No se pudo abrio el archivo "<<nombArch<<endl;
        exit(1);
    }
    numAlum = 0;
    while(1){
        leerAlumno(archAlum,alumnos[numAlum]);
        if(archAlum.eof())break;
        numAlum++;
    }
}
void leerAlumno(ifstream &archAlum, struct Alumno &alumno){
    int n=0;
    archAlum>>alumno.codigo;
    if(archAlum.eof())return;
    archAlum.get(); // Sacamos la coma
    archAlum.getline(alumno.nombre,60,',');
    while(true){
        archAlum>>alumno.notas[n];
        n++;
        if(archAlum.get() == '\n')break;
    }
    alumno.numNotas = n;
}
void calcularPromedios(struct Alumno *alumnos, int numAlum){
    int suma;
    for(int al=0; al<numAlum; al++){
        suma = 0;
        for(int n=0; n<alumnos[al].numNotas; n++)
            suma += alumnos[al].notas[n];
        alumnos[al].promedio = (double)suma/alumnos[al].numNotas;
    }
}
void ordenarDatos(struct Alumno *alumnos, int numAlumn){
    for(int i=0; i<numAlumn-1; i++)
        for(int k= i+1; k<numAlumn; k++)
            if (alumnos[i].promedio<alumnos[k].promedio)
                cambiar(alumnos[i],alumnos[k]);
}

void cambiar(struct Alumno &alumnoI, struct Alumno &alumnoK){
    struct Alumno aux;
    aux = alumnoI;
    alumnoI = alumnoK;
    alumnoK = aux;
}
```

```
void emitirReporte(struct Alumno *alumnos, int numAlumn){
    cout.precision(2);
    cout<<fixed;
    cout<<left<<setw(7)<<" No."<<setw(10)<<"Codigo"<<setw(27)<<"Nombre"
        <<setw(10)<<"Promedio"<<setw(10)<<"Notas"<<endl;
    for(int al=0; al<numAlumn; al++){
        cout<<right<<setw(3)<<al+1<<") "<<setw(10)<<alumnos[al].codigo
            <<left<<"  "<<setw(25)<<alumnos[al].nombre
            <<right<<setw(9)<<alumnos[al].promedio<<"  ";
        for(int n=0; n<alumnos[al].numNotas; n++)
            cout<<setw(3)<<alumnos[al].notas[n];
        cout<<endl;
    }
}
```

Al ejecutar el programa obtendremos lo siguiente:

```
 No. Codigo   Nombre                    Promedio  Notas
  1) 20041003 Alexandra Neyra            17.57    18 15 17 16 18 19 20
  2) 20020107 Paula Gomez                16.71    17 19 20 12 15 16 18
  3) 20101234 Ana Roncal                 15.00    15 16 12 14 18 15
  4) 20002324 Pedro Sanchez              14.50    14 15
  5) 19992345 Juan Lopez                 11.00    12 14 10 08
  6) 20001010 Maria Ruiz                  9.63    10 12 11 09 09 05 11 10
  7) 19971313 Carlos Castro               6.40    02 05 09 10 06
```