Fundamentos de Programación

Autor: Emiliano López <emiliano [dot] lopez [at] gmail [dot] com>
Fecha: 04/09/2014

Unidad 4: Arreglos y Structs

Objetivos

- Comprender la estructura de datos arreglo.
- Comprender el uso de arreglos para almacenar y buscar valores.
- Comprender cómo declarar e inicializar un arreglo, y cómo hacer referencia a los elementos individuales del mismo.
- Declarar y manipular arreglos con múltiples subíndices.
- Comprender la limitación de los arreglos en cuanto al uso de tipos de datos homogéneos
- Comprender la noción de abstracción de datos y la necesidad de crear una estructura definida por el usuario.
- Comprender el mecanismo utilizado para arreglos y structs como argumentos de funciones.

Bibliografía y referencias

Libros

- C++ Cómo programar. 5ta Ed. 2007. Deitel y Deitel. Pearson Educación- Prentice Hall. Capítulo 4 y 6.1 a 6.4.
- El Lenguaje de Programación C++. 3ra Ed. 2001.Stroustrup Bjarne.Addison Wesley. Capítulos 5.2 y 5.7.
- How to think like a computer scientist. 2012. Allen B. Downey. Capítulo 8.
- Jumping into C++. Alex Allain (www.cprogramming.com). Capítulo 10 y 11.

Web

- http://www.cplusplus.com/doc/tutorial/arrays/
- http://www.cplusplus.com/doc/tutorial/structures/
- www.cprogramming.com/tutorial/lesson7.html
- www.cprogramming.com/tutorial/lesson8.html
- http://es.wikibooks.org/wiki/Programación_en_C++/Estructuras

Structs

Los struct sirven para declarar nuevos tipos de datos. Son una estructura de datos que sirven para agrupar otros tipos de datos y denominarlos con un nombre que luego puede ser utilizado para declarar variables de este tipo.

Un struct se declara de la siguiente manera:

```
struct empleado{
        int CUIT;
        char nomape[50];
```

```
        int edad;
};
```

Declaración y asignación

Se lo usa como un nuevo tipo de dato, así como antes declarabamos una variable de tipo int, ahora -siguiendo el ejemplo previo- podemos declarar una variable de tipo empleado:

```
// declaro dos empleados
empleado e1,e2;

// Leo los datos de uno
cin.getline(e1.nomape,50);
cin>>e1.CUIT;
cin>>e1.edad;
```

Fijate que ahora, como la asignación es una operación **únicamente** permitida entre variables del mismo tipo, puedo asignar el empleado e1 al e2 de la siguiente manera:

```
// asigno todo lo de e1 a e2
e2 = e1;

//muetro lo de e2
cout<<e2.nomape;
cout<<e2.CUIT;
cout<<e2.edad;
```

Por supuesto que podría haber hecho la asignación miembro a miembro, sería así:

```
// asigno miembro a miembro
strcpy(e2.nomape,e1.nomape);
e2.CUIT = e1.CUIT;
e2.edad = e1.edad
```

Obviamente que, a menos que me paguen por caracter, me conviene la asignación completa.

Structs en arreglos

Ahora bien, también podemos declarar un arreglo o matriz de este nuevo tipo de datos. Acá tenemos una ventaja, salvamos la limitación de los arreglos en donde sólo se permite un único tipo de datos, ahora podemos declarar un arreglo de un tipo de datos **creado** por nosotros, combinando los tipos de datos ya conocidos (int, float, char, bool). Por ejemplo, para un listado de 50 empleados:

```
//declaro una estructura arreglo
// de 50 empleados
empleado listado[50];

// Los cargo
for (int i = 0; i<50;i++){
        cin.getline(listado[i].nomape,50);
        cin>>listado[i].CUIT;
        cin>>listado[i].edad;
}
```

Structs en funciones

Una función puede recibir un struct como parámetro de entrada, puede retornarlo en su nombre, o también puede recibirlo como parámetro por referencia para ser modificado en la función.

Lo único que se debe tener en cuenta a la hora de usarlos en funciones es que deben ser declarados antes del **main()**. Un programa fuente con funciones y structs como argumentos de esas funciones quedaría de la siguiente forma:

```
#include <iostream>
using namespace std;
//declaro struct
struct empleado{
        int CUIT;
        char nomape[50];
        int edad;
};
//prototipos de funciones
void show(empleado e);

int main(int argc, char *argv[]) {
        // aca va el codigo que usa las funciones
        return 0;
}
// implementacion de funciones
void show(empleado e){
        cout<<e.nomape<<endl;
        cout<<e.CUIT<<endl;
        cout<<e.edad<<endl;
}
```

En el ejemplo previo, se observa un ejemplo de una función *show* que imprime el contenido de un struct empleado que se recibe como argumento.

Ahora, para pasar por referencia un struct lo único que hay que hacer es anteponer el símbolo **&** a la variable struct. Hagamos una función **leer** que nos deje claro el funcionamiento:

```
// implementacion de funciones
void leer(empleado &e){
        cin.getline(e.nomape,50);
        cin>>e.CUIT;
        cin>>e.edad;
}
```

Es muy simple, recibe un empleado por referencia y lo lee, ese cambio en la variable de tipo empleado se verá reflejado desde el programa principal. Su uso desde el *main()* podría ser así:

```
//prototipos de funciones
void show(empleado e);
void leer(empleado &e);
int main(int argc, char *argv[]) {
        empleado e1;

        // lleno con cualquier cosa para ver si anda
        strcpy(e1.nomape,"prueba");
        e1.CUIT = 111;
```

```
        e1.edad = 999;

        //llamo a la funcion
        leer(e1);

        // muestro a ver si modifico
        show(e1);

        return 0;
}
```

Entonces, si por ejemplo queremos leer los 50 empleados podríamos haber hecho lo siguiente desde el código del *main*:

```
int main(int argc, char *argv[]) {
        empleado listado[50];

        // llamo 50 veces a leer
        for (int i = 0; i<50;i++){
                // le mando UN SOLO struct
                leer(listado[i]);
        }

        // llamo 50 veces a show
        for (int i = 0; i<50;i++){
                // le mando UN SOLO struct
                show(listado[i]);
        }

        return 0;
}
```

Ojo, notar que en las funciones creadas NO LES ENVIAMOS el arreglo de structs, sino UN ÚNICO STRUCT, por qué? porque así programamos la función, para que reciba UN STRUCT. Si queremos una función que reciba un arreglo de struct hagamos eso. Por ejemplo, una que nos devuelva el empleado de mayor edad.

Para eso es necesario recibir TODOS los empleados y devolver de alguna manera el empleado que cumple con la consigna.

```
//prototipos de funciones
void show(empleado e);
void leer(empleado &e);
empleado mayor(empleado lista[], int N);

int main(int argc, char *argv[]) {
        empleado listado[50];
        // ya los leimos
        // el codigo para leerlos iria aca

        empleado mayor_emp = mayor(listado, 50);
        cout<<"Empleado mayor: "<<endl;
        show(mayor_emp)
        return 0;
}
```

```
//aca irian las funciones hechas previamente
empleado mayor(empleado lista[], int N){
        int edadm lista[0].edad;
        int posm = 0;
        for (int i = 1; i<N;i++){
                if (listado[i].edad > edadm){
                        edadm = listado[i].edad;
                        posm = i;
                }
        }
return listado[posm];
}
```

Lo interesante de la función mayor es que devuelve un struct empleado completo, con todas las variables miembro. Esto mismo se podría haber hecho utilizando una función *void* que devuelva los valores en uno de sus argumentos pasados por referencia, podría haber sido así:

Notar que se hace una asignación completa, de todos sus miembros de una única vez, de este modo cada vez que encontramos una edad mayor, asignamos a *em* ese empleado con todos sus datos, y NO SE USA un return porque los datos se actualizan en la variable recibida por referencia. Bien, la usaríamos desde el programa principal de la siguiente manera;

```
int main(int argc, char *argv[]) {
        empleado listado[50];

        // ya los leimos
        // el codigo para leerlos iria aca

        empleado mayor_emp
        mayor(listado, 50,mayor_emp);
        cout<<"Empleado mayor: "<<endl;
        show(mayor_emp)
        return 0;
}
```

Ojo que los **arreglos de strucs como argumento** de funciones pasan por referencia al igual que un arreglo de un tipo de dato estándar. Por ejemplo, si queremos incrementar en uno la edad de todos los empleados, podríamos hacer una función así:

```
void incanio(empleado lista[], int N){
        for (int i = 0; i<N;i++)
                listado[i].edad++;
}
```

No hizo falta usar el **&** precediendo **lista[]** porque **LOS ARREGLOS POR DEFECTO PASAN POR REFERENCIA**, con lo cual podemos cambiar el contenido dentro de la función y veremos esos cambios reflejados en el programa principal.

Pero pregunto, si pasamos un struct que dentro tiene como miembro un arreglo y pasamos ese struct a una función sin indicar con el **&** que es por referencia, y dentro de la función cambiamos el contenido del arreglo, esos cambios se ven reflejados?? Si/No??

Con una siemple prueba se evacúa esa duda.