Índice general

TEMA 5. REGISTROS

5.1. INTRODUCCIÓN

5.1.1. MOTIVACIÓN Y DEFINICIÓN

Los vectores almacenan de forma compacta información del mismo tipo sobre una misma entidad.

Los *registros* (*struct*) almacenan de forma compacta información de distinto tipo sobre una misma entidad.

```
TipoAlumno:
   NIF
   Nombre
   Edad
   Curso
   Notas
```

Diremos que NIF, Nombre, Edad, etc, son los *campos* de `TipoAlumno`

```
TipoHora:
   horas
   minutos
   AM
```

Es la segunda vez que vamos a definir un *tipo de dato* para luego declarar variables de dicho tipo (la primera fue con los enumerados)

```
struct TipoAlumno{
   char NIF [9];
   char Nombre [200];
   bool EsBecario;
   int Edad;
   int Curso;
};
```

El tipo struct puede definirse:

- ▷ **Dentro de una función. Sólo podrá ser usado dentro de la función (poco usual)**
- ▷ **Al principio del fichero. Será accesible por todas las funciones definidas con posterioridad**

Las variables de tipo struct pueden definirse en los mismos sitios que cualquier otra variable.

```
struct TipoAlumno{
   char NIF [9];
   char Nombre [200];
   bool EsBecario;
   int Edad;
   int Curso;
};
int main(){
   struct TipoAlumno un_alumno;
   .........
```

5.1.2. ACCESO A LOS CAMPOS

```
<nombre_variable>.<nombre_campo>
```

es una variable del tipo declarado en el struct.
Las operaciones serán las mismas que las aplicables al correspondiente tipo del campo.
Al declarar una variable de tipo struct, los campos estarán a basura.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

struct TipoAlumno{
   char NIF [9];
   char Nombre [200];
   int EsBecario;
   int Edad;
   int Curso;
};

void Incrementa (int *dato){
   *dato = *dato+1;
}
int AniosParaTerminar (int curso_actual){
   return 5 - curso_actual;
}
int main(){
 struct TipoAlumno un_alumno;
```

```
un_alumno.Edad = 21;
un_alumno.EsBecario = 1;
scanf("%d", &un_alumno.Curso);
scanf("%s", un_alumno.Nombre);

                    /* pasamos un int a AniosParaTerminar: */
printf("%d", AniosParaTerminar (un_alumno.Curso));

           /* pasamos por referencia un int a Incrementa: */
Incrementa (&un_alumno.Curso);

                           /* pasamos un string a strlen: */
printf("Núm caract. Nombre = %d", strlen(un_alumno.Nombre));

                            /* pasamos un char a toupper: */
un_alumno.Nombre[0] = toupper (un_alumno.Nombre[0]);
```

5.2. VECTORES Y STRUCTS

▷ **Un campo de un struct puede ser un vector. Ya lo hemos visto con** `Nombre` **y** `NIF`

```
struct TipoAlumno{
   char NIF [9];
   char Nombre [200];
   int EsBecario;
   int Edad;
   int Curso;
   float notas_practicas[3];
};
int main(){
   struct TipoAlumno un_alumno;

   un_alumno.notas_practicas[0] = 3.5;
   scanf("%f", &un_alumno.notas_practicas[0]);
```

▷ Se pueden construir vectores de structs.

```
struct TipoAlumno{
   char NIF [9];
   char Nombre [200];
   int EsBecario;
   int Edad;
   int Curso;
   float notas_practicas[3];
};
int main(){
   struct TipoAlumno clase[100];
   int util_clase = 60;
   int i;
                          /* Primer alumno: */
   .......
   clase[0].Edad = 21;
   clase[0].notas_practicas[0] = 6.7;
   clase[0].notas_practicas[1] = 5.4;
   .......

                          /* Segundo alumno: */
   clase[1].notas_practicas[0] = 9.5;
   clase[1].notas_practicas[1] = 8.4;
   .......

   for (i=0; i<util_clase; i++){
      printf("\nNombre = %s", clase[i].Nombre);
      ......
   }
```

▷ Un campo de un struct puede ser otro struct.

```
struct TipoNotas{
   float Practicas[3];
   float Teorico;
   int PracticaComplementaria;
};

struct TipoAlumno{
   char NIF [9];
   char Nombre [200];
   int EsBecario;
   int Edad;
   int Curso;
   struct TipoNotas Notas;
};

int main(){
   struct TipoAlumno clase[100];

   ..........
   clase[0].Notas.Practicas[0] = 3.4;
   clase[0].Notas.Teorico = 5.4;
   clase[0].Notas.PracticaComplementaria = 1;
}
```

Hay veces en las que, aunque podríamos haber usado un vector para representar la información, es conveniente usar un struct.

Ejemplo. **Coordenadas de un punto.**

```
int main(){
   float coordenadas_centro_circulo[2];

   coordenadas_centro_circulo[0] = 3.4;
   coordenadas_centro_circulo[1] = 4.5;
   .......
```

Ejercicio. **Representad la anterior información usando structs.**

```
struct TipoPunto{
   float abscisa;
   float ordenada;
};

int main(){
   struct TipoPunto centro_circulo;

   centro_circulo.abscisa = 3.4;
   centro_circulo.ordenada = 4.5;
   .......
```

5.3. Trabajando con todo el struct

5.3.1. Asignación

Sabemos que un vector no puede asignarse a otro vector.

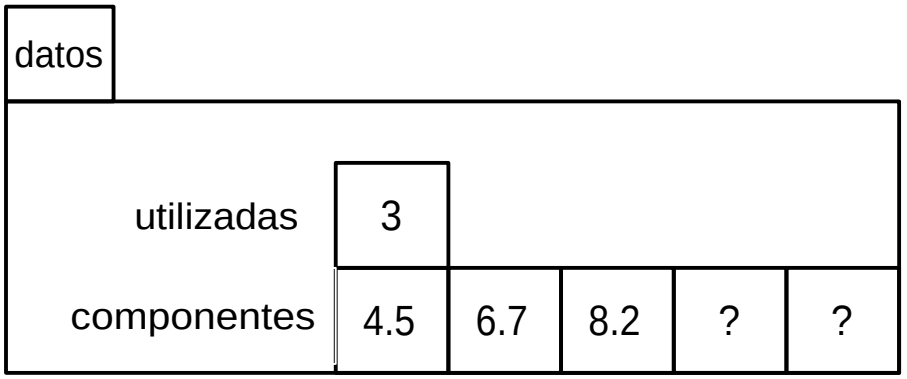
Un struct sí puede asignarse a otro struct. Se copian automáticamente los valores de todos los campos.

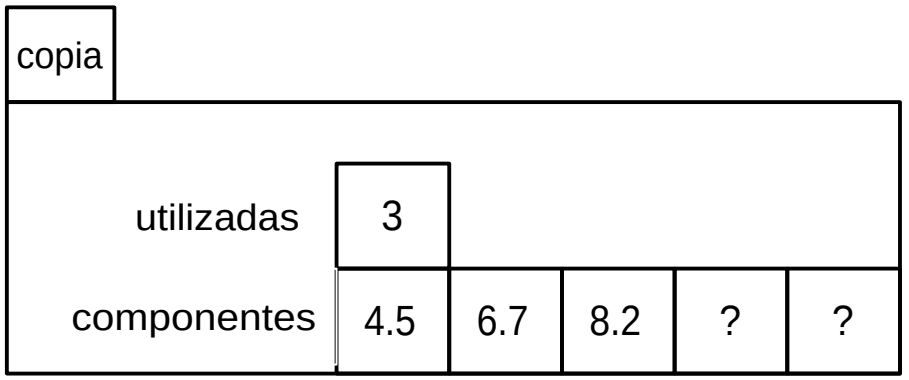
```
int main(){
   struct TipoAlumno un_alumno, otro_alumno;

   un_alumno.Edad = 21;
   un_alumno.Curso = 3;
   strcpy(un_alumno.Nombre, "Juan Carlos");
   strcpy(un_alumno.NIF, "22222222E");

   otro_alumno = un_alumno;
```

Curiosamente, si el struct contiene un vector, se asignan todas las componentes una a una (recordad que los vectores, tal cual, no podían asignarse entre sí)

```
struct TipoAlumno{
   ......
   float NotasPracticas[3];
};

int main(){
   struct TipoAlumno un_alumno, otro_alumno;
   ........
   un_alumno.NotasPracticas[0] = 3.4;
   un_alumno.NotasPracticas[1] = 4.5;
   un_alumno.NotasPracticas[2] = 5;
   otro_alumno = un_alumno;          /* <- Correcto */
```

```
struct TipoVector5float{
   int utilizadas;
   float componentes[5];
};

int main(){
   struct TipoVector5float datos, copia;
   int i;

   datos.utilizadas = 3;
   for (i=0 ; i<datos.utilizadas; i++)
      scanf("%f", &datos.componentes[i]);

   copia = datos;
   .......
}
```


datos
utilizadas
3
componentes
4.5
6.7
8.2
?
?


copia
utilizadas
3
componentes
4.5
6.7
8.2
?
?

Ejercicio. **Representad un punto tridimensional**

Ejercicio. **Representad una esfera**

Ejercicio. **Representad la fecha de nacimiento de una persona.**

5.3.2. FUNCIONES Y STRUCTS

Como la asignación está permitida entre variables de tipo `struct`, también es posible pasar un struct completo como parámetro a una función. Puede hacerse por valor o por referencia.

5.3.2.1. PASO POR VALOR

Como cualquier paso por valor, se realiza una copia del parámetro actual en el parámetro formal (copia de structs)

```
struct TipoAlumno{
    ....
}
void ImprimeDatosAlumno (struct TipoAlumno alumno);

int main()
    struct TipoAlumno un_alumno;

    <asignación a los campos de un_alumno>

    ImprimeDatosAlumno (un_alumno);
}

void ImprimeDatosAlumno (struct TipoAlumno alumno){
    printf("\nNombre = %s", alumno.Nombre);
    printf("\nNIF = %s", alumno.NIF);
    printf("\nCurso = %d", alumno.Curso);
    .....
}
```

Como cualquier paso por valor, la modificación del formal no altera el parámetro actual. Lo mismo ocurre con sus campos. Las modificaciones de los campos del formal no modifican los campos del actual.

```
struct TipoAlumno{
   char Nombre[200];
   ................
}
QuitaExcesoEspaciosEnBlanco (char una_cadena[]);
void ImprimeDatosAlumno (struct TipoAlumno alumno);

int main()
   struct TipoAlumno un_alumno;

   <asignación a los campos de un_alumno>

   ImprimeDatosAlumno (un_alumno);  /* Imprime sin blancos */
   printf("%s", un_alumno.Nombre);  /* Imprime con blancos */
}

void ImprimeDatosAlumno (struct TipoAlumno alumno){

   /* Modificamos alumno.Nombre pero no el parámetro actual: */

   QuitaExcesoEspaciosEnBlanco (alumno.Nombre);
   printf("\nNombre = %s", alumno.Nombre);
   printf("\nNIF = %s", alumno.NIF);
   printf("\nCurso = %d", alumno.Curso);
   ......
}
```

5.3.2.2. FUNCIONES QUE DEVUELVEN UN STRUCT

Una función también puede devolver un struct. Para ello, como siempre, declaramos una variable local de tipo struct y al final ejecutamos un return.

```
struct TipoAlumno LeeDatosAlumno (){
    struct TipoAlumno alumno;

    scanf("%s", alumno.NIF);
    scanf("%s", alumno.Nombre);
    .....

    return alumno;
 }

 int main(){
    struct TipoAlumno un_alumno, clase[100];
    int i;

    un_alumno = LeeDatosAlumno();

    for (i=0; i<util_clase; i++)
       clase[i] = LeeDatosAlumno();
    ......
```

Ejemplo. Representación del tipo complejo de Matemáticas.

```
struct TipoComplejo{
   float real;
   float imaginaria;
};

struct TipoComplejo Suma (struct TipoComplejo complejo1,
                          struct TipoComplejo complejo2){
   struct TipoComplejo aux;

   aux.real = complejo1.real + complejo2.real;
   aux.imaginaria = complejo1.imaginaria +
                    complejo2.imaginaria;
   return aux;
}
int main(){
   struct TipoComplejo dato1, dato2, suma_datos;

   dato1.real = 2;
   dato1.imaginaria = 3;
   dato2.real = 4;
   dato2.imaginaria = 5;

   suma_datos = Suma (dato1, dato2);
   .......
}
```

5.3.2.3. PASO POR REFERENCIA

Como siempre, es decir, el parámetro formal va ligado al actual.

```
void IntercambiaAlumnos (struct TipoAlumno *un_alumno,
                         struct TipoAlumno *otro_alumno){
   struct TipoAlumno aux;

   aux = *un_alumno;
   *un_alumno = *otro_alumno;
   *otro_alumno = aux;
}
int main(){
   struct TipoAlumno clase[100], el_mejor, el_segundo;

   <asignación de valores>

   el_mejor = clase[20];    /* Pedro es el 20 */
   el_segundo = clase[47];  /* María es el 47 */

   IntercambiaAlumnos(&el_mejor, &el_segundo);

   /* el_mejor tendrá ahora una copia de los datos de María
      el_segundo tendrá ahora una copia de los datos de Pedro
      Pedro sigue siendo la componente 20 de clase
      María sigue siendo la componente 47 de clase */

   IntercambiaAlumnos(&clase[20], &clase[47]);

   /* Pedro está ahora en la componente 47
      María está ahora en la componente 20 */
```

Si pasamos por referencia un struct, las modificaciones de los campos del formal, serán modificaciones de los campos del actual.

```
void LeeDatosAlumno (struct TipoAlumno *alumno){
   scanf("%s", (*alumno).NIF);
   scanf("%s", alumno->Nombre);
   .....
}

int main(){
   struct TipoAlumno un_alumno, clase[100];
   int util_clase = 40;
   int i;

   LeeDatosAlumno(&un_alumno);

   for (i=0; i<util_clase; i++)
      LeeDatosAlumno(&clase[i]);
   ......
```

5.3.2.4. PASO POR REFERENCIA CONSTANTE

Un paso por valor de un struct duplica memoria en la pila. Si el struct es muy grande, y las restricciones de memoria son importantes, puede usarse un *paso por referencia constante*:

```
void ImprimeDatosAlumno (const struct TipoAlumno *alumno){
   printf("\nNombre = %s", (*alumno).Nombre);
   printf("\nNIF = %s", alumno->NIF);
   ......
}
```

Se puede modificar el puntero pero no el objeto al que apunta.

```
void ImprimeDatosAlumno (const struct TipoAlumno *alumno){
 QuitaExcesoEspaciosEnBlanco(alumno->Nombre); /*Error com.*/
 printf("\nNombre = %s", alumno->Nombre);
 printf("\nNIF = %s", alumno->NIF);
 ......
}
```

```
void ImprimeDatosAlumno (const struct TipoAlumno *alumno){
   char nombre_local[200];
   strcpy (nombre_local, alumno->Nombre);
   QuitaExcesoEspaciosEnBlanco (nombre_local);
   printf("\nNombre = %s", nombre_local);
   printf("\nNIF = %s",  alumno->NIF);
   ......
}
```

5.3.2.5. MODULARIZACIÓN

Si vamos a crear una función que acceda a las componentes del struct, pasaremos todo el struct y no las componentes individualmente.

```
struct TipoPunto{
   float abscisa;
   float ordenada;
}

                                                    /* :-( */

float DistanciaEuclidea (float abscisa1, float ordenada1,
                         float abscisa2, float ordenada2){
   return sqrt( pow(abscisa1-abscisa2 , 2)  +
                pow(ordenada1-ordenada2, 2)  );
}

int main(){
struct TipoPunto punto1, punto2;
float distancia;

<Asignación de valores>
distancia = DistanciaEuclidea(punto1.abscisa, punto1.ordenada,
                              punto2.abscisa, punto2.ordenada);
}
```

```
struct TipoPunto{
   float abscisa;
   float ordenada;
}

                                                    /* :-) */

float DistanciaEuclidea (struct TipoPunto pt1,
                         struct TipoPunto pt2){
   return sqrt( pow(pt1.abscisa - pt2.abscisa , 2)  +
                pow(pt1.ordenada - pt2.ordenada, 2)  );
}

int main(){
   struct TipoPunto punto1, punto2;
   float distancia;

   <Asignación de valores>
   distancia = DistanciaEuclidea(punto1, punto2);
   ........
}
```