Introducción

En programación el concepto de registro es muy importante y sumamente útil en múltiples aplicaciones ya que muchas veces es necesario establecer una estructura de datos que agrupe información de diversos atributos bajo un único concepto. Algo muy común que se pide en programas o algoritmos es que utilicen fechas como datos para realizar alguna tarea específica y aunque hay diversas representaciones posibles, la más elemental es mediante el ingreso de tres enteros que representen, día, mes y año. Como los tres datos son del mismo tipo, una ocurrencia sería usar un array para guardar los tres enteros. Pero qué ocurre si los datos que queremos guardar bajo una misma estructura son de distinto tipo. Por ejemplo, la información de un sistema destinado a la admisión de pacientes a una determinada clínica médica . ¿Qué datos se necesitan guardar? Como mínimo se necesitan, nombre, apellido y DNI, en este caso afortunadamente los tres podrían ser de tipo cadenas de caracteres. Pero sería normal tambien guardar otros datos como edad (entero), sexo (carácter) y porque no otros datos clínicos, por ejemplo temperatura corporal (real), grupo sanguíneo (cadena), etc., en este caso sería de gran utilidad contar con algún tipo de dato que sirva entonces para guardar información de distinto tipo en una estructura única.

Los registros

Un registro es una estructura de datos formado por un conjunto de elementos llamados campos, no necesariamente del mismo tipo y que permiten almacenar una serie de datos relacionados entre sí bajo un nombre común.

De aquí, sus características básicas:

- Tiene un identificador que representa la estructura.
- Se divide en campos.
- Cada campo puede ser de cualquier tipo primitivo.
- Cada campo tiene su identificador.

Definición de un registro

El ámbito para definir una estructura de tipo registro es, o bien entre las directivas de preprocesamiento y la cabecera de la función (main) o dentro del cuerpo de la función al momento de declarar las variables.

La forma general de definir una estructura es la siguiente:

```
struct<Identificador_Etiqueta> //Nombre la estructura
    {
        //Campos de la estructura
        <Especificador_de_tipo_1><Identificador_1>;
        <Especificador_de_tipo_2><Identificador_2>;
        <Especificador_de_tipo_3><Identificador_3>;
        ...
        <Especificador_de_tipo_4><Identificador_n>;
    };
```

Nos referimos a definición de estructura porque en este punto del código no se han declarado variables. Solo se ha definido la forma de la estructura que pasará a ser un especificador de tipo o un nuevo tipo de dato.

Declaración de una variable

Como mencionamos anteriormente, cuando se define una estructura esencialmente se está definiendo un tipo de dato compuesto de diferentes elementos o campos. No existe realmente aún una variable de ese tipo hasta que se declara.

La forma general de declarar una variable de este tipo es:

```
struct<Identificador_Etiqueta><Identificador_variable>;
```

De esta forma se declara una variable de tipo <Identificador_Etiqueta> con el nombre o identificador de variable que hayamos designado.

Veamos un ejemplo concreto:

```c
#include <stdio.h>
#include <stdlib.h>
struct datos//Definición de la estructura
{
    //Campos de la estructura
    int Legajo;
    int edad;
};
int main()
{
    struct datos alumno;//Declaración de una variable
    ...
return 0;
}
```

Se define una estructura denominada "datos", y luego se declara una variable "alumno" del tipo "datos" que tiene la estructura definida anteriormente.

Como pueden observar la estructura está definida antes de main(), y en la primera línea de código dentro del cuerpo de la función se define la variable, utilizando el nuevo tipo de dato, en este caso denominado "datos".

Según lo expuesto más arriba, aunque esto no es lo más usual, podríamos definir la estructura al momento de declarar las variables es decir dentro de la función main, por ejemplo:

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    struct datos //Definición de la estructura "datos"
    {
        //Campos de la estructura
        int Legajo;
        int edad;
    } alumno; //Declaración de una variable
    ...
return 0;
}
```

Si bien los dos ámbitos son válidos, no se suele definir la estructura en el propio cuerpo de main dado que el alcance de dicha definición se limita al cuerpo de la función principal.

Por lo general necesitamos aquella estructura como tipo de dato en otras funciones; por lo cual es mejor definirla fuera de main y que dicha estructura quede disponible como tipo de dato, para ser utilizada por cualquier otra función.

Uso de typedef

En el lenguaje C existen muchos tipos de datos primitivos (int, char, float etc.) y estructuras de datos. Mediante la palabra clave typedef podemos definir un alias para un tipo de dato existente.

La sintaxis es la siguiente:

```
typedef <Especificador_de_tipo> Alias
```

Donde `<Especificador_de_tipo> es algún tipo de dato conocido.`

Veamos el siguiente ejemplo:

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

typedef int entero;//Tengo un nuevo tipo de dato que lo llamé "entero"

int main()
{
    entero num1,num2;
    num1=5;
    num2=2;
    printf("%d + %d = %d",num1,num2,num1+num2);
    getch();
    return 0;
}
```

Como mencionamos en las líneas anteriores , el lenguaje C permite definir un nuevo nombre de tipo de dato usando la palabra reservada typedef. Es muy común asignarle un alias o sinónimo al nombre de una estructura, como es el caso de los registros para así, evitar poner la palabra reservada struct cada vez que utilizamos una variable de este tipo.

Veamos un ejemplo con este uso:

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

typedef struct
{
    //Campos de la estructura
    int Legajo;
    int edad;
}datos;//Definición de un "nuevo" tipo de datos

int main()
{
    datos alumno;//Declaración de una variable de tipo "datos"
    ...
return 0;
}
```

Note que en este caso no necesito utilizar la palabra reservada struct en la declaración de la variable.

Operaciones con estructuras

Inicialización

Del mismo modo que vimos con vectores y matrices podemos inicializar un registro al momento de declarar las variables.

Veamos el siguiente ejemplo:

```
...
```

```c
//Definición de la estructura "datos"
struct datos
{
    //Campos de la estructura
    int Legajo;
    char nombre[10];
    int edad;
};

int main()
{
    //Declaracion e inicializacion de una variable
    struct datos alumno = {99050,"Raul",25};
    return 0;
}
```

Es importante destacar que, si se inicializa una variable de esta forma, es necesario colocarles un valor a todos los campos de la estructura.

Acceso a los campos

El mecanismo de acceso a cada dato es mediante el nombre de la variable declarada previamente del tipo de estructura deseado, seguido del operador punto (.) y el nombre del campo que se quiere acceder.

Veamos el siguiente ejemplo:

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

//Definición de la estructura "datos"
struct datos
{
    //Campos de la estructura
    char nombre[10];
    int edad;
};

int main()
```

```
{
   //Declaracion e inicializacion de una variable
   struct datos alumno = {"Carlos", 25};

   //Muestra los datos por pantalla
   printf(" %s tiene %d a%cos", alumno.nombre, alumno.edad, 164);
   return 0;
}
```

Veamos un ejemplo completo, en donde se lee una estructura y se muestra por pantalla, utilizando funciones de entrada/salida estándar. En este caso solo leemos un dato y lo mostramos.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct
{
    char Nombre[10];
    int edad;
} datos;
int main()
{
  datos persona;
  printf("Ingrese el nombre de la persona ");
  scanf("%s", persona.Nombre);
  printf("Ingrese la edad de la persona ");
  scanf("%d", &persona.edad);
  printf("%s tiene %d a%cos", persona.Nombre, persona.edad, 164);
  return 0;
}
```

Otra operación muy común a realizar con los registros o estructuras es copiar o asignar un registro en otro, esto se puede hacer íntegramente sin mayor problema, teniendo la precaución de que ambas variables tengan la misma estructura.

A continuación, utilizando el código del ejemplo anterior, mostramos cómo sería esta operación:

```c
#include <stdio.h>
typedef struct
{
    char Nombre[10];
    int edad;
} datos;
int main(){
    datos persona, nuevo;
    printf("Ingrese el nombre de la persona ");
    scanf("%s", persona.Nombre);
    printf("Ingrese la edad de la persona ");
    scanf("%d", &persona.edad);
    nuevo = persona; //Copia en nuevo el registro completo
    printf("%s tiene %d a%cos", nuevo.Nombre, nuevo.edad, 164);
    return 0;
}
```

Registros anidadas

De la misma forma que podemos anidar ciclos, condiciones, etc. También una estructura puede estar dentro de otra estructura y a esto se le conoce como anidamiento o estructuras anidadas.

Veamos un ejemplo:

```c
...
struct infopersona //Definición de la primer estructura
{
    char direccion[25];
    char ciudad[20];
    int codigo_postal;
};
struct empleado //Definición de la segunda estructura
```

```
{
  char NombrEmpleado[25];
  struct infopersona DirecEmpleado;//Uso la definición anterior
  double Salario;
};
int main()
{
    struct empleado persona;
    ...
    return 0;
}
```

La variable persona está declarada como tipo empleado que a su vez está compuesta por el tipo infopersona.

Para acceder a los distintos campos, el mecanismo es igual que en un struct simple, es decir mediante el nombre de la variable, seguida del operador punto (.) y el nombre del campo al que se quiere acceder. Solo que en este caso si hago referencia a otro struct se necesita a su vez otro nombre de campo también seguido de un punto y el campo correspondiente al que voy a acceder.

Veamos el siguiente ejemplo utilizando las estructuras creadas anteriormente:

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

struct infopersona
{
    char direccion[30];
    int numero;
    char ciudad[20];
    int codigo_postal;
};
struct empleado
{
    char NombrEmpleado[25];
    struct infopersona DirecEmpleado;
    float Salario;
};
int main()
{
```

```
    struct empleado persona;

    printf("Ingrese el nombre de la persona ");
    scanf("%s", persona.NombrEmpleado);
    printf("Ingrese la direccion de la persona ");
    fflush(stdin);
    gets(persona.DirecEmpleado.direccion);
    printf("Ingrese el salario de la persona ");
    scanf("%f", &persona.Salario);

    //Muestra por pantalla los datos del empleado
    printf("%s ", persona.NombrEmpleado);
    printf("vive en %s",persona.DirecEmpleado.direccion);
    printf(" tiene un salario de $ %.2f", persona.Salario);

return 0;
}
```

Note que no es necesario leer o completar el registro completo, leo solo los campos que me interesan o que voy a utilizar.

Estructuras en arrays

Como ya mencionamos en otro apartado, los vectores permiten agrupar varios elementos de un mismo tipo. Cada elemento de un vector es accesible a través de un índice.
Como también se mencionó anteriormente, los registros o estructuras son agrupaciones heterogéneas de datos cuyos elementos (denominados campos) son accesibles mediante identificadores.

Es posible agrupar un conjunto de elementos de tipo estructura en un array y acceder a diferentes elementos de un grupo a través del identificador con su correspondiente índice.

Veamos el siguiente ejemplo que lee y muestra 10 registros correspondientes a legajos y edad de un grupo de alumnos:

```
#include <stdio.h>
#include <stdlib.h>
#define n 5
typedef struct
{
```

```
    int Legajo;
    int edad;
} datos;
int main ()
{
   datos alumnos[n];//Vector de 10 elementos de tipo "datos"
   for (int i = 0; i < n; i++) //Ciclo para hacer los 10 ingresos
    {
        printf("Ingrese el legajo del alumno ");
        scanf("%d", &alumnos[i].Legajo);
        printf("Ingrese la edad del alumno ");
        scanf("%d", &alumnos[i].edad);
    }
   printf("legajo\tEdad\n ");
   //Ciclo para mostrar los 10 registros
   for (int i = 0; i < n; i++)
    {
       printf("%d\t%d \n ", alumnos[i].Legajo, alumnos[i].edad);
    }
   return 0;
}
```

Note que al nombre de la variable le sigue el índice entre [] que indica que estoy trabajando con un arrays unidimensional seguido de operador punto (.) y luego el nombre del campo al que voy a acceder.

Estructuras con arrays

Así como los campos de la estructura pueden ser de cualquier tipo de dato primitivo, también pueden ser un array. Veamos a continuación un ejemplo con este uso:

```
#include <stdio.h>
#define n 5
typedef struct
{
    int Legajo;
    int notas[n];//Un campo de tipo vector de enteros
} datos;
int main()
{
    datos alumnos;
```

```
    printf("Ingrese el legajo del alumno ");
    scanf("%d", &alumnos.Legajo);

    for (int i = 0; i < n; i++) //Ciclo para leer las 5 notas
    {
        printf("Ingrese la nota %d del alumno ", i + 1);
        scanf("%d", &alumnos.notas[i]);
    }

    printf("\nLegajo %d\n ", alumnos.Legajo);
    for (int i = 0; i < n; i++) //Ciclo para mostrar las 5 notas
    {
        printf("Nota %d: %d\n ", i + 1, alumnos.notas[i]);
    }
    return 0;
}
```

En este caso estoy leyendo un alumno con sus 5 notas, si tengo por ejemplo 10 alumnos en código cambiaría sustancialmente y necesitaríamos dos índices con dos ciclos como se ve a continuación.

Cargamos 10 alumnos con 5 notas cada uno:

```
#include <stdio.h>
#define n 5 //Cantidad de notas
#define m 10 //Cantidad de alumnos
typedef struct
{
    int Legajo;
    int notas[n];//Un campo de tipo vector de enteros
} datos;
int main()
{
    datos alumnos[m];
    int i,j;
    for (i=0;i<m;i++)//Ciclo para leer las 10 alumnos

    {
            printf("Ingrese el legajo del alumno %d ",i+1);
            scanf("%d", &alumnos[i].Legajo);
            for (int j= 0; j< n; j++) //Ciclo para leer las 5 notas
            {
```

```
                printf("Ingrese la nota %d ", j+1);
                scanf("%d", &alumnos[i].notas[j]);
            }
    }
     for (i=0;i<m;i++)//Ciclo para mostra las 10 alumnos
    {
          printf("\nLegajo %d\n ", alumnos[i].Legajo);
          for (int j = 0; j < n; j++) //Ciclo para mostrar las 5 notas
          {
               printf("Nota %d: %d\n ", j + 1, alumnos[i].notas[j]);
          }
    }
    return 0;
}
```

Registros como parámetros a Funciones.

Como hemos visto en apartados anteriores una función puede recibir tipos de datos simples como int, char y float y lo que sucede es que se hace una copia del parámetro o dicho de otra forma funciona como espejo de ese parámetro.

.

También hemos visto y trabajado con parámetros de tipo vector y matriz y el funcionamiento es muy distinto a los tipos de datos simples. Si en la función modificamos el parámetro lo que sucede es que se modifica el contenido del vector o la matriz que le pasamos desde la función main.

Ahora veamos cómo trata el lenguaje C los parámetros de tipo struct o registros. Su funcionamiento es idéntico a los tipos de datos simples, es decir que se hace una copia del registro en el parámetro. Por lo tanto no podemos modificar el registro que le enviamos a la función sino sólo acceder para consultarlo, salvo que se trate de un arrays de ese tipo.

Analicemos el siguiente ejemplo: Leer un registro compuesto por un campo legajo y un campo edad. Define una función "Mostrar" a la cual se le pasa como parámetro un registro y lo que hace la función es imprimir por pantalla el registro completo.

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

```c
typedef struct
{
    int Legajo;
    int edad;
} datos;
void Mostrar(datos);//Prototipo con parámetro del tipo "datos"
int main()
{
    datos alumno;
    printf("Ingrese el legajo ");
    scanf("%d", &alumno.Legajo);
     printf("Ingrese la edad ");
     scanf("%d", &alumno.edad);
     Mostrar(alumno);//Pasa el registro completo
return 0;
}
void Mostrar(datos alu)
{
       printf("\nLegajo %d\n", alu.Legajo);
       printf("Edad %d\n ", alu.edad);

}
```

Note que en este caso se lee un solo registro y se muestra por pantalla a través de la función "Mostrar".

Observemos la siguiente situación:

```c
#include <stdio.h>

typedef struct
{
    int Legajo;
    int edad;
} datos;
//Prototipos de las funciones
void Cargar(datos);
void Mostrar(datos);

int main()
{
     datos alumno={98765,34};//Precarga un dato
```

```
     printf("\nLegajo %d\n", alumno.Legajo);
     printf("Edad %d\n\n", alumno.edad);
     Cargar(alumno);//LLama a la función cargar
     Mostrar(alumno);//Muestra los datos originales
return 0;
}

void Cargar(datos alu)
{
    printf("Ingrese el legajo ");
    scanf("%d", &alu.Legajo);
    printf("Ingrese la edad ");
    scanf("%d", &alu.edad);
 }
void Mostrar(datos alu)
{
       printf("\nLegajo %d\n", alu.Legajo);
       printf("Edad %d\n ", alu.edad);
}
```

En la función "Carga", se lee un nuevo dato pero no se ve reflejado en main por tratarse de un parámetro por valor tal y como funcionan los parámetros en C.

Sin embargo, cuando necesitemos que una función cambie los campos de una variable ya sea de algún tipo primitivo o de tipo struct como vimos en la situación anterior, es necesario que los argumentos formales se declaren de tipo puntero o se devuelvan en el nombre de la función. Aquí se muestran las dos opciones:

Primer ejemplo usando return:

```
#include <stdio.h>

typedef struct
{
    int Legajo;
    int edad;
} datos;

datos Cargar(); //La función cargar retorna un tipo "datos
void Mostrar(datos);
int main()
{
     datos alumno;
```

```
     alumno=Cargar();//Asigna “Cargar” a una variable tipo "datos"
     Mostrar(alumno);
return 0;
}
datos Cargar()
{
    datos aux;
    printf("Ingrese el legajo ");
    scanf("%d", &aux.Legajo);
    printf("Ingrese la edad ");
    scanf("%d", &aux.edad);
  return aux;
 }
void Mostrar(datos alu)
{
       printf("\nLegajo %d\n", alu.Legajo);
       printf("Edad %d\n ", alu.edad);
}
```

Segundo ejemplo usando punteros:

```
#include <stdio.h>
typedef struct
{
    int Legajo;
    int edad;
} datos;
//El parámetro de las funciones es un puntero al tipo "datos"
void Cargar(datos *);
void Mostrar(datos *);
int main()
{
     datos alumno={89888,23};
     printf("\nLegajo %d\n", alumno.Legajo);
     printf("Edad %d\n ", alumno.edad);
     Cargar(&alumno); //LLama con la dirección de alumno
     Mostrar(&alumno);
return 0;
}
void Cargar(datos *alu)
{
    printf("Ingrese el legajo ");
```

```
    scanf("%d", &alu->Legajo);
    printf("Ingrese la edad ");
    scanf("%d", &alu->edad);
}
void Mostrar(datos *alu)
{
printf("\nLegajo %d\n", alu->Legajo);// ->reemplaza (*alu).Legajo
printf("Edad %d\n ", alu->edad);
}
```

El lenguaje C proporciona el operador -> (guión medio y mayor que) con el propósito de simplificar la escritura. Esto es, por ejemplo alu->Legajo que quiere decir que alu es un puntero a un struct y que estamos accediendo al campo Legajo.

Si utilizamos un vector para cargar más de un dato se procede de la misma forma que cuando trabajamos las funciones con vectores, solo que en este caso entra un vector del tipo struct.

Veamos el ejemplo:

```
#include <stdio.h>
#define n 5
typedef struct
{
    int Legajo;
    int edad;
} datos;
void Cargar(datos[],int);
void Mostrar(datos[],int);
int main()
{
     datos alumnos[n];
     Cargar(alumnos,n);
     Mostrar(alumnos,n);
return 0;
}

void Cargar(datos alu[],int c)
{
   int i;
```

```
   for(i=0;i<c;i++)
   {
       printf("Ingrese el legajo ");
       scanf("%d", &alu[i].Legajo);
       printf("Ingrese la edad ");
      scanf("%d", &alu[i].edad);
   }

 }
void Mostrar(datos alu[],int c)
{
       int i;
        printf("\nLegajo \tEdad\n");
        for(i=0;i<c;i++)
        {
               printf("%-10d%d\n",alu[i].Legajo,alu[i].edad);
        }
}
```

También podemos necesitar pasar solo un campo a la función sin necesidad de pasar todo el registro completo en ese caso se pasa el dato como un primitivo con su correspondiente especificador de formato.

Ejemplo