

# **El lenguaje de programación C**

## **- Enumerados - Struct**



# Enumerados

- En ocasiones el programador puede estar interesado en trabajar con tipos simples cuyos valores no coinciden con las características de ninguno de los tipos simples predefinidos. Entonces puede definir un nuevo tipo enumerado, cuyos valores serán aquellos que explícitamente se enumeren. De esta forma se consigue disponer tipos que expresan mejor las características de las entidades manipuladas por el programa, por lo que el programa será más legible y fácil de entender. Por ejemplo, si en un programa quisiéramos tratar con colores, con lo visto hasta ahora, para hacer referencia a un determinado color habría que seleccionar un tipo predefinido (por ejemplo, char) y suponer que cada color será representado por un cierto carácter (por ejemplo, 'A' para el color azul). En su lugar, podemos definir un nuevo tipo Color que contenga AZUL como uno de sus valores predefinidos. Para ello, haríamos lo siguiente:

```
enum Color { ROJO, AZUL, AMARILLO } ;
```

# Enumerados

- El tipo enum permite utilizar un grupo de constantes a través de nombres asociados que son más representativos que dichas constantes. Las constantes pueden ser de los tipos siguientes: byte, short, int o long. En su forma más simple una enumeración puede tener el siguiente aspecto:

```
enum Colores { Rojo=2, Verde=3, Azul=4 };
```

# Enumerados

Si no se especifica ningún valor, por ejemplo:

```
enum Colores { Rojo, Verde, Azul };
```

El compilador asigna los valores por defecto siguientes : Rojo es 0, Verde 1 y Azul 2 y son de tipo int. En general, el primer elemento es 0 y cada elemento sucesivo aumenta en una unidad.

# Enumerados

Es posible cambiar el valor inicial y/o cualquier valor:

```
enum Colores { Rojo=1, Verde, Azul };
```

En este caso, Rojo es 1, Verde 2 y Azul 3 y son de tipo int.

# Enumerados

```
enum Colores2 { Rojo=20, Verde=30, Azul };
```

Rojo es 20, Verde 30 y Azul 31 y son de tipo int

Y también es posible cambiar el tipo:

```
enum Colores : byte { Rojo, Verde, Azul };
```

En este caso, Rojo, Verde y Azul serán de tipo byte.

# Eejmplo

```
enum e_compania {  
    Audi,  
    BMW,  
    Cadillac,  
    Ford,  
    Jaguar,  
    Lexus  
    Maybach,  
    RollsRoyce,  
    Saab  
};  
  
e_compania mi_auto;  
mi_auto= RollsRoyce;  
//...  
  
if (mi_auto == Ford)  
    cout << "Hola, dueño de Ford" << endl;
```

# Tipos de datos estructurados

- Tipo de datos que no son simples
- Simples
  - Números
  - Letras
  - Verdadero/falso
- Estructurados o compuestos
  - Combinaciones de tipos simples
  - Los arreglos son tipo de datos estructurados



# Tipos de datos definidos por el usuario

- Al momento de hacer un programa, el usuario puede definir sus propios tipos de datos
  - Mayor claridad.
  - Aumenta el significado semántico del código.
  - Simplificar declaración de variables.

# Tipos de datos definidos por el usuario

- Typedef
  - Define un nuevo nombre para un tipo de dato.
  - El nombre original sigue siendo válido.

```
typedef <tipo> <nuevo nombre>;
```

```
typedef int positivo;
```

# Tipos de datos definidos por el usuario

```
typedef int positivo;
typedef int negativo;

int main(){

    positivo a,b;
    negativo c,d;

    a=1;
    b=2;

    c=-a;
    d=-b;

    printf("%d %d %d %d\n",a,b,c,d);
}
```

# Tipos de datos definidos por el usuario

- Otra forma de definir tipos de datos es componer varios datos simples en uno solo.
- Esto se denomina estructura.
- Una estructura es un tipo de dato que contiene un conjunto de valores relacionados entre si de forma lógica.
- Estos valores pueden ser de distinto tipo.
- Generalmente, se refiere a un concepto más complejo que un número o una letra.

# Estructuras

- Una estructura puede verse como una colección de variables que se referencia bajo un nombre en común.
- Cada una de estas variables se denominan “miembros” de la estructura. Otras denominaciones son:
  - Campo
  - elemento
  - atributo

# Declaración de estructuras

- La definición de una estructura se realiza fuera de cualquier función, generalmente en la parte superior del archivo.
- Para definir una estructura requerimos:
  - Un nombre
  - Una lista de miembros
    - Nombre
    - Tipo

# Declaración de estructuras

**Reservada**

**Nombre único**

`struct mi_estructura{`

`int miembro1;`

`char miembro2;`

`double miembro3;`

`...`

`} ;`

**Lista de  
miembros**

**Termino de la declaración**

**Declaración**

# Declaración de estructuras

- La declaración de una estructura no crea variables.
- Solo se define el nombre y sus miembros.
- Debe estar definida para poder ser utilizada (posición en el código).



# Uso de estructuras

- Una vez que se ha declarado la estructura puede ser utilizada.
- Para utilizarla hay que definir variables del tipo “estructura”.
- Para definir estas variables se utiliza la siguiente sintaxis:


```
struct nombre_estructura nombre_variable;
```

# Uso de estructuras

```
struct mi_estructura{  
    int miembro1;  
    char miembro2;  
    double miembro3;  
};
```

...

```
struct mi_estructura m1;  
struct mi_estructura m2;
```



**Dos variables del  
tipo mi\_estructura**

# Operaciones con estructuras

- Una vez definidas las variables, es necesario realizar operaciones con ellas.
- Lo realmente útil no es la estructura, sino sus miembros.
- Para acceder a los valores de los miembros de una variable de tipo estructura se utiliza el operador unario “.”.
- Cada miembro es una variable común y corriente.

# Operaciones con estructuras

```
struct mi_estructura{  
    int miembro1;  
    char miembro2;  
    double miembro3;  
};
```

...

```
struct mi_estructura1;  
m1.miembro1=1024;  
m1.miembro2='x';  
m1.miembro3=12.8;
```

# Operaciones con estructuras

```
struct mi_estructural;  
Printf("m1=%d, m2=%c, m3=%f\n",  
    m1.miembro1,  
    m1.miembro2=,  
    m1.miembro3);
```

# Ejemplo

- A la antigua

```
char nombreAlumno [64];  
int edadAlumno;  
double promedioAlumno;
```

- Con estructuras

```
struct alumno{  
    char nombre[64];  
    int edad;  
    double promedio;  
};
```

# Operaciones con estructuras

- Operador de asignacion
  - Copia una variable de estructura a otra (miembro por miembro)
- Operadores de comparación
  - No tiene sentido a nivel de estructuras, solo a nivel de miembros.

# Estructuras y funciones

- Para pasar miembros de una estructura a una función, se utiliza el mismo esquema de las variables comunes.

```
void mostrarNota(int nota);
```

```
int validarNota(int *nota);
```

```
...
```

```
Struct alumno a1;
```

```
if(validarNota(&a1.nota))
```

```
    mostrarNota(a1.nota);
```



# Estructuras y funciones

- Para pasar estructuras completas como parámetros se debe especificar el tipo completo de la estructura en la definición del parámetro.

# Estructuras y funciones

```
void mostrarAlumno(struct alumno a) {  
    printf("rol: %d-%c, edad: %d\n",  
           a.rol, a.dig, a.edad);  
}  
  
void inicializarAlumno(struct alumno *a) {  
    (*a).rol=0;  
    (*a).dig='0';  
    (*a).edad=0;  
}  
  
...  
struct alumno a1;  
inicializarAlumno(&a1);  
mostrarAlumno(a1);
```

# Estructuras y funciones

- La notacion ‘(\*).’ Se puede resumir con ‘->’.
- Agrega claridad al código.
- Se denomina operador “flecha”.

```
void inicializarAlumno(struct alumno *a) {  
    a->rol=0;  
    a->dig='0';  
    a->edad=0;  
}
```

# Estructuras y funciones

- Para devolver estructuras como resultado de una función, se utiliza el mismo esquema de siempre.
- El resultado se copia a la variable que lo recibe.

# Estructuras y funciones

```
struct vector{  
    double x;  
    double y;  
};
```

```
struct vector sumar(struct vector v1, struct vector v2) {  
    struct vector vres;  
    vres.x = v1.x + v2.x;  
    vres.y = v1.y + v2.y;  
  
    return vres;  
}
```

# Estructuras y funciones

```
int main() {  
    struct vector va;  
    struct vector vb;  
    struct vector vc;  
  
    va.x=0.5;  
    va.y=1;  
    vb.x=1;  
    vb.y=0.5;  
    vc = sumar(va,vb) ;  
    printf("res: %.2f,%.2f\n",vc.x,vc.y) ;  
  
}
```

# Estructuras anidadas

- Nada impide que los miembros de una estructura sean a su vez tipos de datos estructurados, es decir:
  - Otras estructuras
  - Arreglos
- Estas estructuras se denominan anidadas.
- Incluso pueden ser estructuras recursivas.

# Estructuras anidadas

```
struct punto{  
    double x;  
    double y;  
};
```

```
struct circunferencia{  
    struct punto centro;  
    double radio;  
};
```



# Estructuras anidadas

```
double perimetro(struct circunferencia c) {  
    return 2*PI*c.radio;  
}
```

```
double area(struct circunferencia c) {  
    return PI*c.radio*c.radio;  
}
```

# Estructuras anidadas

```
double distancia(struct punto p1, struct punto p2) {  
    return sqrt( pow(p2.x+p1.x,2) + pow(p2.y+p1.y,2) );  
}  
  
int intersectan(struct circunferencia c1, struct  
    circunferencia c2) {  
  
    double dist = distancia(c1.centro, c2.centro);  
    printf("%.2f vs %.2f\n", dist, c1.radio+c2.radio);  
    return (dist < c1.radio+c2.radio);  
}
```

# Estructuras anidadas

```
int main() {  
    struct circunferencia ca;  
    struct circunferencia cb;  
  
    ca.centro.x=0;  
    ca.centro.y=0;  
    ca.radio = 1;  
  
    cb.centro.x=1.9;  
    cb.centro.y=0;  
    cb.radio = 1;  
  
    printf("p:%.2f, a:%.2f,  
    int?%s\n",perimetro(ca),area(ca),  
            (intersectan(ca,cb)?"Si":"No"));  
  
}
```

# Estructuras anidadas

```
struct alumno{  
    int rol;  
    char dig;  
    double notas[3];  
};
```

```
double promedio(struct alumno a){  
    return (a.notas[0] + a.notas[1] +  
        a.notas[2])/3.0;  
}
```

# Estructuras anidadas

```
int main() {  
    struct alumno a;  
    a.rol=1;  
    a.dig='1';  
    a.notas[0]=55;  
    a.notas[1]=50;  
    a.notas[2]=61;  
    printf("Prom: %.2f\n",promedio(a));  
}
```

# Arreglos de estructuras

- Se puede crear arreglos cuyos elementos sean variables de estructura.
- Se definen de manera similar al caso común.

```
tipo arreglo[N]
```

```
struct estructura arreglo[N];
```

# Arreglos de estructuras

```
struct alumno{
    int rol;
    int promedio;
};
int main(){
    int i, suma=0;
    struct alumno alumnos[N];
    double promedio;
    for(i=0;i<N;i++){
        printf("Ingrese rol y nota: ");
        scanf("%d
%d", &alumnos[i].rol, &alumnos[i].promedio);
    }
    for(i=0;i<N;i++)
        suma+=alumnos[i].promedio;
    promedio = (1.0*suma)/N;
    printf("Promedio del curso: %.1f",promedio);
}
```

# Arreglos de estructuras

```
struct alumno{
    int rol;
    char dig;
    double notas[3];
};
int main(){
    struct alumno alumnos[N];
    int i=0;
    for(i=0;i<N;i++){
        alumnos[i].rol=i;
        alumnos[i].dig='1'+i;
        alumnos[i].notas[0]=40+5*i;
        alumnos[i].notas[1]=alumnos[i].notas[0]*0.5;
        alumnos[i].notas[2]=alumnos[i].notas[0]*1.6;
    }
    for(i=0;i<N;i++){
        printf("%d) Prom: %.2f\n",i+1,promedio(alumnos[i]));
    }
    return 1;
}
```



# Arreglos de estructuras

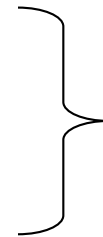
```
int main(){
    struct alumno alumnos[N];
    int i=0;
    for(i=0;i<N;i++){
        printf("Ingrese rol:");
        scanf("%d-%c",
            &alumnos[i].rol, &alumnos[i].dig);
        printf("Ingrese notas:");
        scanf("%lf %lf %lf",
            &alumnos[i].notas[0],
            &alumnos[i].notas[1],
            &alumnos[i].notas[2]);
    }
    for(i=0;i<N;i++){
        printf("%d-%c: %.2f\n",
            alumnos[i].rol,
            alumnos[i].dig,
            promedio(alumnos[i]));
    }
    return 1;
}
```

# Búsqueda de estructuras

- La búsqueda de estructuras es similar a la búsqueda de datos simples.
- Existen dos detalles importantes:
  - Definir el concepto de igualdad entre estructuras
    - No se puede usar “= =”
    - Puede ser a través de un campo
    - Puede ser a través de varios campos
  - Definir valor “no encontrado”

# Búsqueda de estructuras

```
struct album{  
    char grupo[32];  
    char titulo[32];  
    int precio;  
};
```



**Ambos campos  
definen la igualdad**

# Búsqueda de estructuras

```
int main() {  
    struct album coleccion[N];  
    char g[32], t[32];  
    llenar(coleccion);  
    printf("Ingrese grupo: "); gets(g);  
    printf("Ingrese titulo: "); gets(t);  
  
    /*Buscar album*/  
    if(/*verificar si se encontro*/)  
        printf("Precio: %d\n", /*Mostrar precio*/);  
    else  
        printf("No esta en stock\n");  
}
```

# Búsqueda de estructuras


```
int buscar1(char grupo[], char titulo[], struct album coleccion[]){
    int i;
    for(i=0;i<N;i++){
        if(strcmp(grupo, coleccion[i].grupo)==0 &&
            strcmp(titulo, coleccion[i].titulo)==0)
            return i;
    }
    return -1;
}
...
int pos;
pos = buscar1(g,t, coleccion);
if(pos>=0)
    printf("1)Precio: %d\n", coleccion[pos].precio);
else
    printf("1)No esta en stock\n");
```

Se devuelve la posición donde se encontró el elemento

Posición inválida

# Búsqueda de estructuras

```
struct album buscar2(char grupo[], char titulo[], struct album
coleccion[]){
    struct album a;
    int i;
    a.precio=-1;
    for(i=0;i<N;i++){
        if(strcmp(grupo, coleccion[i].grupo)==0 &&
            strcmp(titulo, coleccion[i].titulo)==0)
            a =coleccion[i];
    }
    return a;
}
...
struct album a;
a = buscar2(g,t, coleccion);
if(a.precio>0)
    printf("2)Precio: %d\n", a.precio);
else
    printf("2)No esta en stock\n");
```



Se devuelve la estructura encontrada.

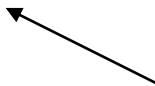
Devuelve precio inválido

# Búsqueda de estructuras

```
void buscar3(struct album *a, struct album coleccion[]){
    int i;
    a->precio=-1;
    for(i=0;i<N;i++){
        if(strcmp(a->grupo, coleccion[i].grupo)==0 &&
            strcmp(a->titulo, coleccion[i].titulo)==0)
            a->precio = coleccion[i].precio;
    }
}

...
struct album a;
strcpy(a.grupo,g);
strcpy(a.titulo,t);
buscar3(&a, coleccion);
if(a.precio>0)
    printf("3) Precio: %d\n", a.precio);
else
    printf("3) No esta en stock\n");
```

“Rellena” los datos que faltan



“Llena” parcialmente la estructura



# Ordenamiento de estructuras

- Al igual que en las búsquedas, el procedimiento es similar.
- Solo falta definir la relación de orden
  - Puede estar definida por un solo campo
  - Puede estar definida por varios campos
    - Por lo general, se define un campo principal y otro para el “desempate”.



# Ordenamiento de estructuras

```
void bubblesort_up(struct album coleccion[]){
    int i,j;
    for(i=1;i<N;i++)
        for(j=0;j<(N-i);j++)


            if(coleccion[j].precio>coleccion[j+1].precio) {
                struct album aux = coleccion[j+1];
                coleccion[j+1] = coleccion[j];
                coleccion[j] = aux;
            }
}
```

# Ordenamiento de estructuras

```
struct cliente{  
    char apellido[32];  
    char nombre[32];  
    int gasto_mensual;  
};
```

# Ordenamiento de estructuras

```
void bubblesort_up(struct cliente cartera[]){
    int i,j;
    for(i=1;i<N;i++)
        for(j=0;j<(N-i);j++)
            if(cmp(cartera[j],cartera[j+1])>0){
                struct cliente aux = cartera[j+1];
                cartera[j+1] = cartera[j];
                cartera[j] = aux;
            }
}
```



- Menor que cero, primero menor que el segundo
- Igual a cero, primero igual al segundo
- Mayor que cero, primero mayor que el segundo

# Ordenamiento de estructuras

```
int cmp(struct cliente c1, struct cliente c2){  
    return c1.gasto_mensual- c2.gasto_mensual);  
}
```

```
int cmp(struct cliente c1, struct cliente c2){  
    int m = strcmp(c1.apellido, c2.apellido);  
    if (m!=0)  
        return m;  
    else{  
        m=strcmp(c1.nombre, c2.nombre);  
        if (m!=0)  
            return m;  
        else  
            return c1.gasto_mensual-c2.gasto_mensual;  
    }  
}
```