

# Datos definidos por el usuario

Estructuras de datos

# Estructuras

- Los arreglos son estructuras de datos que contienen un número determinado de elementos (su tamaño) y todos los elementos han de ser del mismo tipo de datos; es una estructura de datos homogénea.
- Esta característica supone una gran limitación cuando se requieren grupos de elementos con tipos diferentes de datos cada uno.



# Estructuras

- Los componentes individuales de una estructura se llaman campos.
- Cada campo (elemento) de una estructura puede contener datos de un tipo diferente de otros campos.
- Por ejemplo, se puede utilizar una estructura para almacenar diferentes tipos de información sobre una persona, tal como nombre, estado civil, edad y fecha de nacimiento.
- Cada uno de estos elementos se denominan nombre del campo.



# Estructuras

- La estructura CD contiene cinco campos.
- Tras decidir los campos, se debe decidir cuáles son los tipos de datos para utilizar por los campos.

Nombre campo	Tipo de dato
Título	Arreglo de caracteres de tamaño 30
Artista	Arreglo de caracteres de tamaño 25
Número de canciones	Entero
Precio	Coma flotante
Fecha de compra	Arreglo de caracteres de tamaño 11



# Declaración de una estructura

- Una estructura es un tipo de dato definido por el usuario, que se debe declarar antes de que se pueda utilizar.
- El formato de la declaración es:

```
struct <nombre de la estructura>
{
    <tipo de dato campo> <nombre campo>;
    <tipo de dato campo> <nombre campo>;
    ...
    <tipo de dato campo> <nombre campo>;
};
```



# Declaración de una estructura

- La declaración de la estructura CD es:

```
struct colección_CD
{
    char título[30];
    char artista[25];
    int num_canciones;
    float precio;
    char fecha_compra[11];
};
```

# Ejemplo

```
struct complejo
{
    float parte_real;
    float parte_imaginaria;
};
```

- En este otro ejemplo se declara el tipo estructura venta:

```
struct venta
{
    char vendedor[30];
    unsigned int codigo;
    int unidades_articulos;
    float precio_unidades;
};
```



# Definición de variables de estructuras

- A una estructura se accede utilizando una variable o variables que se deben definir después de la declaración de la estructura.
- Del mismo modo que sucede en otras situaciones, en C existen dos conceptos similares a considerar, *declaración* y *definición*.
- Una **declaración** especifica simplemente el nombre y el formato de la estructura de datos, pero no reserva almacenamiento en memoria; la declaración especifica un nuevo tipo de dato: *struct <nombre\_structura>*.



# Definición de variables de estructuras

- Cada **definición** de variable para una estructura dada crea un área en memoria en donde los datos se almacenan de acuerdo con el formato estructurado declarado.
- Las variables de estructuras se pueden definir de dos formas:
  1. Listando el tipo de la estructura creado seguido por las variables correspondientes en cualquier lugar del programa antes de utilizarlas.

```
struct colecciones_CD cd1, cd2, cd3;
```



# Definición de variables de estructuras

2. Listándolas inmediatamente después de la llave de cierre de la declaración de la estructura.

```
struct colecciones_CD
{
    char titulo[30];
    char artista[25];
    int num_canciones;
    float precio;
    char fecha_compra[11];
} cd1, cd2, cd3;
```

# Definición / Declaración

- Considérese un programa que gestione libros y procese los siguientes datos: título del libro, nombre del autor, editorial y año de publicación.

```
struct info_libro
{
    char titulo[60];
    char autor[30];
    char editorial[30];
    int anyo;
};
```

```
struct info_libro
{
    char titulo[60];
    char autor[30];
    char editorial[30];
    int anyo;
} libro1, libro2, libro3;
```

```
struct info_libro libro1, libro2, libro3;
```

Definición



Definición



# Definición / Declaración

- Ahora se nos plantea una aplicación de control de los participantes en una carrera popular, cada participante se representa por los datos: nombre, edad, sexo, categoría, club y tiempo.

```
struct corredor
{
    char nombre[40];
    int edad;
    char sexo;
    char categoria[20];
    char club[26];
    float tiempo;
};
```

```
struct corredor v1, s1, c1;
```



# Uso de estructuras en asignaciones

- Como una estructura es un tipo de dato similar a un int o un char, se puede asignar una estructura a otra.
- Por ejemplo, se puede hacer que libro3, libro4 y libro5 tengan los mismos valores en sus campos que libro1.

```
libro3 = libro1;  
libro4 = libro1;  
libro5 = libro1;
```

- De modo alternativo se puede escribir:

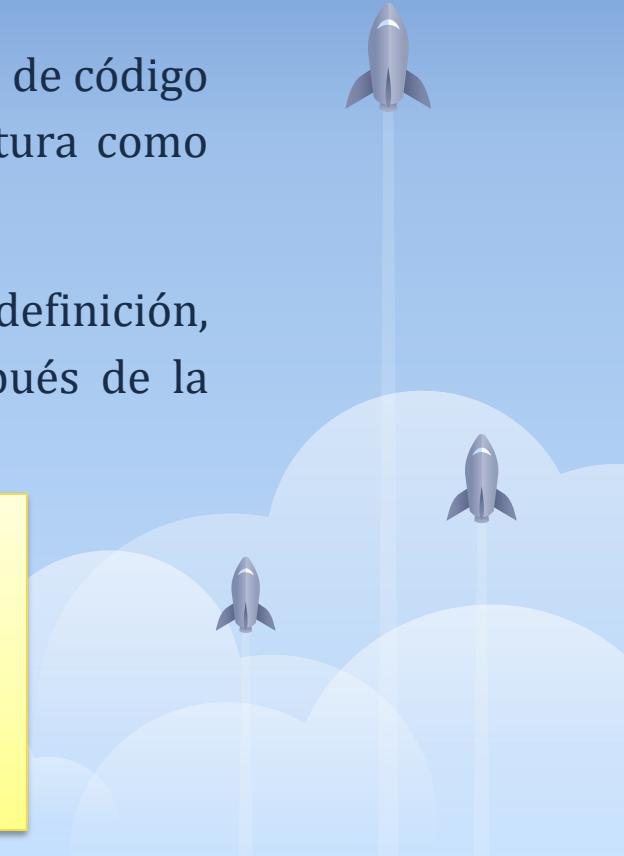
```
libro4 = libro5 = libro6 = libro1;
```



# Inicialización de una declaración de estructuras

- Se puede inicializar una estructura de dos formas.
- Se puede inicializar una estructura dentro de la sección de código de su programa, o bien se puede inicializar la estructura como parte de la definición.
- Cuando se inicializa una estructura como parte de la definición, se especifican los valores iniciales, entre llaves, después de la definición de variables estructura.

```
struct <tipo> <nombre variable estructura> =  
{  
    valor_campo,  
    valor_campo,  
    ...  
    valor_campo  
};
```



# Inicialización de una declaración de estructuras

```
struct info_libro
{
    char titulo[60];
    char autor[30];
    char editorial[30];
    int anyo;
} libro1 = {"Maravilla del saber" , "Lucas García" , "Mc Graw Hill" , 1999};
```

```
struct colección_CD
{
    char título[30];
    char artista[25];
    int num_canciones;
    float precio;
    char fecha_compra[11];
} cd1 = {"El humo nubla tus ojos" , "Porter" , 15 , 254.5 , "02/10/2021"};
```

# Inicialización de una declaración de estructuras

Declaración

```
struct corredor  
{  
    char nombre[40];  
    int edad;  
    char sexo;  
    char categoria[20];  
    char club[26];  
    float tiempo;  
};
```

Inicialización

```
struct corredor v1 =  
{  
    “Salvador Gomez”,  
    29,  
    ‘M’,  
    “Senior”,  
    “Independiente”,  
    0.0  
};
```



# El tamaño de una estructura

- El operador sizeof se aplica sobre un tipo de datos, o bien sobre una variable.
- Se puede aplicar para determinar el tamaño que ocupa en memoria una estructura.
- Al ejecutar el programa se produce la salida:

sizeof(persona): 40

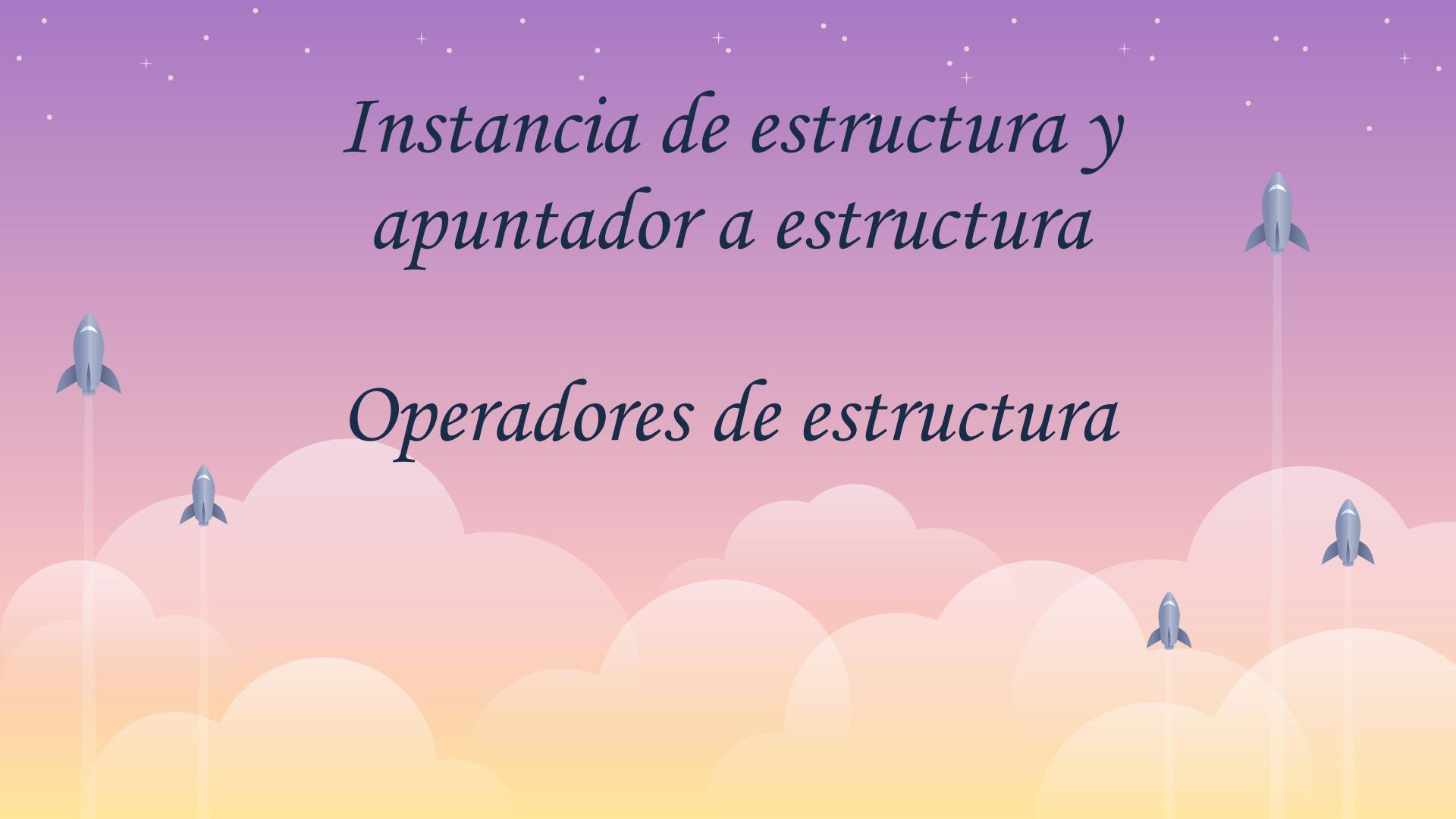
```
#include <stdio.h>

//declarar una estructura Persona
struct persona
{
    char nombre[30];           //30 bytes
    int edad;                  //2 bytes
    float altura;              //4 bytes
    float peso;                //4 bytes
};

int main()
{
    struct persona mar;
    printf("sizeof(persona): %d \n", sizeof(mar));
    return 0;
}
```

*Instancia de estructura y  
apuntador a estructura*

*Operadores de estructura*



# Acceso a estructuras

- Cuando se accede a una estructura, o bien se almacena información en la estructura o se recupera la información de la estructura.
- Se puede acceder a los campos de una estructura de una de estas dos formas:
  - ▷ Utilizando el operador punto ( . )
  - ▷ Utilizando el operador puntero ->



# Almacenamiento de información en estructuras

- Se puede almacenar información en una estructura mediante inicialización, asignación directa o lectura del teclado.



## Acceso a una estructura mediante el operador punto

- La asignación de datos a los campos de una variable estructura se hace mediante el operador punto.

```
<nombre variable estructura> . <nombre campo> = datos;
```

- Algunos ejemplos:

```
strcpy(cd1.titulo,"Granada") ;
cd1.precio = 3450.75;
cd1.num_canciones = 7;
```

```
struct colecciones_CD
{
    char titulo[30];
    char artista[25];
    int num_canciones;
    float precio;
    char fecha_compra[11];
}cd1;
```

# Acceso a una estructura mediante el operador punto

- El operador punto proporciona el camino directo al campo correspondiente.
- Los datos que se almacenan en un campo dado deben ser del mismo tipo que el tipo declarado para ese campo.

```
struct corredor
{
    char nombre[40];
    int edad;
    char sexo;
    char categoria[20];
    char club[26];
    float tiempo;
};
```

```
struct corredor cr;

printf("Nombre: ");
gets(cr.nombre);

printf("edad: ");
scanf("%d", &cr.edad);

printf("Sexo: ");
scanf("%c ", &cr.sexo);

printf("Club: ");
gets(cr.club);

printf("Tiempo: ");
scanf("%f", &cr.tiempo);

if (cr.edad <= 18)
    strcpy(cr.categoria,"Juvenil");
else if(cr.edad <= 40)
    strcpy(cr.categoria,"Senior");
else
    strcpy(cr.categoria,"Veterano");
```

# Acceso a una estructura mediante el operador puntero

- El operador puntero, -> sirve para acceder a los datos de la estructura a partir de un puntero.
- Para utilizar este operador se debe definir primero una variable puntero para apuntar a la estructura.
- A continuación, utilice simplemente el operador puntero para apuntar a un campo dado.

```
<puntero de estructura> -> <nombre campo> = datos;
```

```
struct estudiante
{
    char nombre[40];
    int num_estudiante;
    int anyo_matricula;
    float nota;
};
```



# Acceso a una estructura mediante el operador puntero .

- Se puede definir ptr\_est como un puntero a la estructura:

```
struct estudiante *ptr_est;  
struct estudiante mejor;
```

- A los campos de la estructura estudiante se pueden asignar datos como sigue (siempre y cuando la estructura ya tenga creado su espacio de almacenamiento)

```
ptr_est = &mejor;      //ptr_est tiene la dirección(apunta a) mejor  
  
strcpy( ptr_est->nombre , “Alondra” );  
ptr_est->num_estudiante = 3425;  
ptr_est->nota = 8.5;
```

# Lectura de información de una estructura

- Si ahora se desea introducir la información en la estructura basta con acceder a los campos de la estructura con el operador punto o flecha (puntero).
- Se puede introducir la información desde el teclado o desde un archivo, o asignar valores calculados.

```
<nombre variable> = <nombre variable estructura>.<nombre campo>;
```

```
<nombre variable> = <puntero de estructura>-><nombre campo>;
```

# Recuperación de información de una estructura

```
#include<stdio.h>
#include<math.h>
int main()
{
    struct complejo
    {
        float pr;
        float pi;
        float modulo;
    };

    struct complejo z;
    struct complejo *pz;

    pz = &z;
    z.pr = 3;
    z.pi = 4;
    z.modulo = sqrt(z.pr*z.pr + z.pi*z.pi);

    printf("\nNúmero complejo (%.1f,% .1f), modulo: %.2f", pz->pr, pz->pi, pz->modulo);
}
```

# *Estructuras de datos*



# Tipos de datos definidos por el programador

- Los lenguajes de programación le permiten al programador crear y definir sus propios tipos de datos.
- En C los nuevos tipos se definen con la palabra reservada **typedef**.
- La palabra reservada **typedef** proporciona un mecanismo para la creación de sinónimos (o alias) para los tipos de datos en C.

# Tipos de datos definidos por el programador

- La posibilidad de definir nuestros propios tipos nos permite crear un nivel de abstracción sobre el cual podemos ocultar gran parte de la complejidad que emerge de las técnicas que utilizamos para resolver algoritmos.

```
#include <stdio.h>
typedef int Entero; // defino un nuevo tipo de datos

int main()
{
    // declaro una variable de tipo Entero
    Entero e = 5;
    printf("%d\n", e);

    return 0;
}
```



# Estructuras o registros

```
// defino el tipo Fecha  
typedef long Fecha;
```

Una estructura representa un conjunto de variables, probablemente de diferentes tipos, cuyos valores están relacionados entre sí, razón por la cual resulta conveniente tratarlos como unidad. Por ejemplo: un par de coordenadas que marcan un punto dentro de un plano, una terna de enteros que indican el día, mes y año de una fecha o la matrícula, el nombre y la fecha de ingreso de un empleado de una compañía.

Las siguientes líneas de código muestran dos formas de definir la estructura Empleado

Sin usar typedef	Usando typedef
<pre>struct Empleado {     int matricula;     char nombre[20];     Fecha fechaIngreso; };</pre>	<pre>typedef struct Empleado {     int matricula;     char nombre[20];     Fecha fechaIngreso; }Empleado;</pre>

Las dos formas de definir la estructura son correctas. La primera (sin `typedef`) define el tipo de datos `struct Empleado`. En cambio, la segunda define dos tipos de datos: `struct Empleado` y `Empleado`. En general, utilizaremos la segunda opción.

Luego de la definición de la estructura y su correspondiente `typedef` podemos declarar y utilizar variables de este nuevo tipo de datos de la siguiente manera:

```
// declaro una variable de tipo Empleado  
Empleado e;  
  
e.matricula=31234; // asigno la matricula  
strcpy(e.nombre, "Juan"); // asigno el nombre  
e.fechaIngreso=crearFecha(21,6,1992); // asigno la fecha
```

Las variables que componen una estructura se llaman “campos”. Así, la estructura `Empleado` está compuesta los campos `matricula`, `nombre` y `fechaIngreso`.

Gráficamente, podemos representar una estructura de la siguiente manera:

`Empleado`

<code>matricula</code>	<code>nombre</code>	<code>fechaIngreso</code>
<code>int</code>	<code>char[20]</code>	<code>Fecha</code>

Representación gráfica de la estructura `Empleado`.

# *Arreglos de estructuras*



# Arreglos de estructuras

- Se puede crear un arreglo de estructuras tal como se crea un arreglo de otros tipos.
- Los arreglos de estructuras son idóneos para almacenar un archivo completo de empleados, un archivo de inventario, o cualquier otro conjunto de datos que se adapte a un formato de estructura.
- Mientras que los arreglos proporcionan un medio práctico de almacenar diversos valores del mismo tipo, los arreglos de estructuras le permiten almacenar juntos diversos valores de diferentes tipos, agrupados como estructuras.



# Arreglos de estructuras

- La declaración de un arreglo de estructuras info\_libro se puede hacer de un modo similar a cualquier arreglo, es decir, asigna un arreglo de 100 elementos denominado libros.

```
struct info_libro
{
    char titulo[60];
    char autor[30];
    char editorial[30];
    int anyo;
};
```

```
struct info_libro libros[100];
```



# Arreglos de estructuras

- Para acceder a los campos de cada uno de los elementos de la estructura se utiliza una notación de arreglo.
- Para inicializar el primer elemento de libros, por ejemplo, su código debe hacer referencia a los campos de libros[0] de la forma siguiente:

```
strcpy(libros[0].titulo, "C++ a su alcance");
strcpy(libros[0].autor, "Luis Joyanes") ;
strcpy(libros[0].editorial, "McGraw-Hill') ;
libros[0].anyo = 1999;
```



# Arreglos de estructuras

- También puede inicializarse un arreglo de estructuras en el punto de la declaración encerrando la lista de inicializadores entre llaves {}.

```
struct info_libro lib[3] = {  
    "C++ a su alcance", "Luis Joyanes", "McGraw-Hill", 1999,  
    "Estructura de datos", "Luis Joyanes", "McGraw-Hill", 1999,  
    "Problemas en Pascal", "Angel Hermoso", "McGraw-Hill", 1997 };
```

# Arreglos de estructuras

- En el siguiente ejemplo se declara una estructura que representa a un número racional, un arreglo de números racionales es inicializado con valores al azar.

```
struct racional
{
    int N;
    int D;
};

struct racional rac[4] = { 1,2, 2,3, -4,7, 0,1};
```

# *Estructuras anidadas*



# Estructuras anidadas

- Una estructura puede contener otras estructuras llamadas estructuras anidadas.
- Las estructuras anidadas ahorran tiempo en la escritura de programas que utilizan estructuras similares.



```
struct empleado
{
    char nombre_emp[30];
    char direccion[25];
    char ciudad[20];
    char provincia[20];
    long int cod_postal;
    double salario;
};
```

```
struct clientes
{
    char nombre_cliente[30];
    char direccion[25];
    char ciudad[20];
    char provincia[20];
    long int cod_postal;
    double saldo;
};
```



# Estructuras anidadas

- Estas estructuras contienen muchos datos diferentes, aunque hay datos que están solapados.
- Así, se podría disponer de una estructura, info\_dir, que contuviera los miembros comunes.

```
struct info_dir
{
    char direccion[25];
    char ciudad[20];
    char provincia[20];
    long int cod_postal;
};
```



# Estructuras anidadas

- Esta estructura se puede utilizar como un campo de las otras estructuras, es decir, anidarse.

```
struct empleado
{
    char nombre_emp[30];
    struct info_dir direccion_emp;
    double salario;
};
```

```
struct clientes
{
    char nombre_cliente[30];
    struct info_dir direccion_cliente;
    double saldo;
};
```



```
#include<stdio.h>
#include<string.h>
int main()
{
    struct info_dir
    {
        char direccion[25];
        char ciudad[20];
        char provincia[20];
        long int cod_postal;
    };

    struct empleado
    {
        char nombre_emp[30];
        struct info_dir direccion_emp;
        double salario;
    };

    struct clientes
    {
        char nombre_cliente[30];
        struct info_dir direccion_cliente;
        double saldo;
    };
}
```

```
struct empleado empleado1;

strcpy(empleado1.nombre_emp,"Juan");
empleado1.salario = 235.6;
strcpy(empleado1.direccion_emp.direccion,"direccion");
strcpy(empleado1.direccion_emp.ciudad,"ciudad");
strcpy(empleado1.direccion_emp.provincia,"provincia");
empleado1.direccion_emp.cod_postal = 55555;
```

```
}
```

# Estructuras anidadas

Cuando el tipo de datos de algún campo de la estructura es en sí otra estructura decimos que son estructuras anidadas.

En el siguiente ejemplo, definimos la estructura `Dirección` cuyos campos permiten almacenar los datos que componen una dirección postal. Luego definimos la estructura `Persona` cuyo campo `dirección` es de tipo `Direccion`.

```
typedef struct Direccion
{
    char calle[50];
    int numero;
    int piso;
    char depto;
}Direccion;

typedef struct Persona
{
    char nombre[30];
    long dni;
    Direccion direccion;
}Persona;
```

# Estructuras anidadas

Persona y Dirección son estructuras anidadas. Luego, si declaramos una variable de tipo Persona y queremos asignar valores a sus campos podemos hacerlo de la siguiente manera:

```
// declaro una variable de tipo Direccion  
Direccion d;  
strcpy(d.calle, "Los Patos");  
d.numero=222;  
d.depto='A';  
d.piso=12;  
  
// declaro una variable de tipo Persona  
Persona p;  
strcpy(p.nombre, "Pablo"); // asigno el nombre  
p.dni=23354212;           // asigno el DNI  
p.direccion=d;            // asigno la direccion
```

# Estructuras anidadas



La representación gráfica de la estructura `Persona` es la siguiente:

Persona

nombre	dni	direccion			
		calle	numero	depto	piso
char [30]	long	char [50]	int	char	int

Representación gráfica de estructuras anidadas.



# *Estructuras* Más ejemplos

# Estructuras con campos de tipo “Arreglo de estructuras”

Modificaremos la estructura Alumno para que incluya un campo de tipo Nota[].

```
typedef struct Nota
{
    int puntaje;
    long fecha;
}Nota;
```

```
typedef struct Alumno
{
    int matricula;    // matricula del alumno
    char nombre[30]; // nombre
    Nota notas[3];   // notas obtenidas en los examenes
}Alumno;
```

# Estructuras con campos de tipo “Arreglo de estructuras”

Podemos representarla de la siguiente manera:

Alumno

matricula	nombre	notas
int	char[20]	Nota[3]

puntaje	fecha

Estructura con un campo de tipo *array* de estructura.

En las siguientes líneas, asignamos el valor 10 en la segunda nota de un alumno.

```
Alumno a;  
a.notas[1].puntaje=10;
```

# Estructuras con campos de tipo arreglo

La siguiente estructura incluye un campo de tipo `int[]`.

```
typedef struct Alumno
{
    int matricula;          // matricula del alumno
    char nombre[30];        // nombre
    int notas[3];           // notas obtenidas en los examenes
}Alumno;
```

Podríamos representarla de la siguiente manera:



# Estructuras con campos de tipo arreglo

Podríamos representarla de la siguiente manera:

Alumno

matricula	nombre	notas			
int	char[20]	<table border="1"><tr><td>0</td><td>1</td><td>2</td></tr></table> int[3]	0	1	2
0	1	2			

Representación de una estructura con un campo de tipo array.

Luego, con las siguientes líneas de código asignamos el valor 10 en la segunda nota de un alumno.

```
Alumno a;  
a.notas[1]=10;
```

# Punteros a estructuras

```
typedef struct Empleado  
{  
    int matricula;  
    char nombre[20];  
    Fecha fechaIngreso;  
}Empleado;
```

```
// defino el tipo Fecha  
typedef long Fecha;
```

Sean las siguientes líneas de código:

```
Empleado e;  
Empleado* p = &e;
```

Para asignar un valor en el campo `matricula` de la estructura que está siendo direccionada por `p` tendremos que hacer:

```
// asigno 10 a la matricula del empleado  
(*p).matricula = 10;
```

# Punteros a estructuras

Como vemos, fue necesario utilizar paréntesis para indicar que el operador de indirección `*` aplica sobre el puntero `p`. Esta situación es habitual cuando, en una función, necesitamos modificar los valores de los campos de una estructura que se recibe por referencia.

Por ejemplo, en la siguiente función recibimos un puntero a una estructura de tipo `Empleado` y asignamos valores en todos sus campos.

```
void cargarEmpleado(Empleado* e)
{
    (*e).matricula=10;
    strcpy((*e).nombre, "Pablo");
    (*e).fechaIngreso=crearFecha(2,10,2009);
}
```

# Punteros a estructuras

## El operador flecha



El operador `->` (léase “flecha”) permite hacer referencia directa a los campos de una estructura que es accedida a través de un puntero. Utilizando este operador, la función anterior podría reescribirse de la siguiente manera:

```
▼  
void cargarEmpleado(Empleado* e)  
{  
    e->matricula=10;  
    strcpy(e->nombre, "Pablo");  
    e->fechaIngreso=crearFecha(2, 10, 2009);  
}
```

