Tema 2: Estructuras y Uniones

Introducción a la programación II

Ana Isabel Sierra de las Heras
Marcos Novalbos
Tiago Manuel Louro Machado de Simas
Rodrigo Alonso Solaguren-Beascoa
Alfonso Castro Escudero

Índice

Estructuras. Concepto

- Una estructura es una colección de uno o más tipos de elementos definidos por el usuario y denominados miembros, cada uno de los cuales puede ser un tipo un dato diferente y que se agrupan bajo el mismo nombre.
- Una estructura puede contener cualquier número de miembros, cada uno de los cuales tiene un nombre único, denominado nombre del miembro.
- Se debe declarar antes de ser utilizada
- Una instancia de una estructura es una variable que ha sido declarada utilizando una estructura definida previamente
- Para acceder a cada uno de los miembros de una variable estructura se utiliza el operador "."

Estructuras. Declaración

- **El formato de la declaración** o especifica **un nuevo tipo de dato:**

```
struct <nombre de la estructura>
{
    <tipo de datos miembro1> <nombre miembro1>
    <tipo de datos miembro2> <nombre miembro2>
    …
    <tipo de datos miembroN> <nombre miembroN>
};
```

Declaración de un tipo struct

- La **declaración** especifica simplemente el nombre y el formato de la estructura de datos, pero no reserva almacenamiento en memoria.

- Es en la **definición** de una variable de tipo estructura cuando se crea un área en memoria en donde los datos se almacenan de acuerdo al formato estructurado declarado.

```
struct <nombre de la estructura> variable;
```

Definición de una variable tipo struct

Estructuras. Ejemplo

- La estructura Persona tiene tres miembros: nombre, edad y altura. Luego, se declara una variable (instancia) de esta estructura llamada persona1, y se asignan valores a sus miembros utilizando el operador de acceso a miembros (.). Finalmente, se accede a los miembros de la estructura persona1 para mostrar su información por pantalla

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

// Definición de la estructura
struct Persona {
    char nombre[50];
    int edad;
    float altura;
};

void main(int argc, char *argv[]){ {
    // Declaración de una instancia de la estructura Persona
    struct Persona persona1;

    // Asignación de valores a los miembros de la estructura
    strcpy(persona1.nombre, "Juan");
    persona1.edad = 30;
    persona1.altura = 1.75;

    // Acceso a los miembros de la estructura y muestra de información
    printf("Nombre: %s\n", persona1.nombre);
    printf("Edad: %d\n", persona1.edad);
    printf("Altura: %.2f\n", persona1.altura);
}
```

Estructuras & Bases de datos

- Las estructuras y las bases de datos tratan de modelar datos, representar y gestionarlos.
- Las estructuras pueden contener diferentes tipos de datos y se utilizan para organizar y representar datos de manera estructurada. De manera similar, en una base de datos, las tablas se utilizan para organizar y almacenar conjuntos de datos relacionados.
- Cada fila en una tabla de base de datos puede considerarse análoga a una instancia de una estructura en C.
- Tipos de datos:
  - Cuenta con una variedad de tipos de datos básicos (enteros, flotantes, caracteres, etc.) que se utilizan para definir las características los miembros de estructuras. Del mismo modo, en una base de datos, los tipos de datos se utilizan para definir el formato y la estructura de los datos almacenados en las columnas de una tabla.
- Manipulación de datos:
  - En C, se utilizan operadores y funciones para manipular datos almacenados en variables y estructuras. De manera similar, en una base de datos, se utilizan instrucciones SQL (como SELECT, INSERT, UPDATE y DELETE) para manipular datos almacenados en tablas.

Ejemplo

- Imaginemos que una biblioteca necesita almacenar datos de libros y relacionarlos con sus autores:
  - Características a almacenar:
    - **Autor**:
      - Nombre
      - Apellido
      - Nacionalidad
    - **Libro:**
      - Título
      - ISBN
      - Número Páginas
      - Editorial
      - Autor
- Declarar las estructuras necesarias

Ejemplo: Declaración de estructuras

```
struct autor_t
{
   char nombre[10];
   char apellido[10];
   char nacionalidad [10];
};
```

```
struct libro_t
{
   char titulo [20];
   char ISBN [13];
   int nPags;
   char editorial[20];
   struct autor_t autor;
};
```

Para poder usar aquí el tipo "autor_t" debe estar declarado en nuestro programa antes de declarar el tipo "libro_t"

Estructuras. Definición de variables

- Las variables de tipo estructura se pueden definir de dos formas:
  - **Listando el tipo de la estructura creado seguida por las variables** correspondientes en cualquier lugar del programa antes de utilizarlas
  - **Listándolas inmediatamente después de la llave de cierre de la declaración de la estructura**
- Se puede inicializar una estructura de dos formas:
  - Dentro de la sección de código de su programa
  - Como parte de la definición

Estructuras. Definición variable

- Definición de una variable estructura:
  - Listando el tipo de la estructura creado seguida por las variables

```
struct <tipo estructura> var1, var2, …,varN;
```

  - Listando el tipo de la estructura creado seguida por las variables

```
struct <tipo estructura>
{
  tipo miembro1;
  tipo miembro2;
  …
  tipo miembroN;
}var1, var2, … ,varN;
```

Ejemplo Estructuras

- Construye un programa que permita almacenar siguientes datos de los 25 alumnos de una clase. Edad, altura, nombre y apellido.
- Declara una variable (una única variable) que almacene los datos del siguiente alumno en particular: Santiago Castillo, 20 años y 1,81 metros.

Estructuras. Operadores

- El operador sizeof se puede aplicar para determinar el tamaño que ocupa una estructura

- Se puede acceder a los miembros de una estructura de dos maneras dependiendo del tipo de dato de cada uno de los miembros:

  - Utilizando el operador punto (.) Cuando la variable es de tipo estructura

    <nombre de la variable estructura> . <nombre miembro> = datos;

  - Utilizando el operador flecha (→) Cuando la variable es de tipo puntero a estructura

    <nombre del puntero a estructura> → <nombre miembro> = datos;

Estructuras: Paso de parámetros

- Una estructura se puede pasar como parámetro de una función.

  ¿Cómo podemos imprimir una variable de tipo struct?

  OJO!! No funciona printf que sólo sirve para imprimir tipos básicos (%d, %s,…)

  Hacer una función "Imprimir estructuras por pantalla"
  - Crear una nueva función "void imprimeLibro(struct datosAlumno_t Alumno1)"

- El paso de parámetros con estructuras es igual que con cualquier otro tipo:

  - **Paso parámetros por valor ("por copia**"):
    - void imprimeLibro(struct datosAlumno_t Alumno1)
      - Es más lento, depende del tamaño de la estructura. Si es muy grande habrá que copiar muchos datos a pila por cada llamada

  - **Paso de parámetros por referencia ("por puntero")**:
    - void imprimeLibro(struct datosAlumno_t *Alumno1);
      - Más rápido, sólo se copia la dirección de memoria (entre 4 y 8 bytes, dependiendo de arquitectura)

Ejemplo Paso de parámetros

```
#include <stdio.h>
#define TAM_MAX 25

struct datosAlumno {
      char nombre_apellido [TAM_MAX];
      int edad;
      float altura;
};
void imprimeDatosAlmuno1 (struct datosAlumno Alumno);
void imprimeDatosAlmuno2 (struct datosAlumno *Alumno);

int main (int argc, char *argv[]){
       struct datosAlumno Alumno1 = {"Santiago
       Castillo", 20, 2.81};

      imprimeDatosAlmuno1 (Alumno1);
      imprimeDatosAlmuno2 (&Alumno1);

return 0;
}
```

```
void imprimeDatosAlmuno1 (struct datosAlumno Alumno){
   printf("La edad del alumno %s es %d y su altura es
   %3.2f\n",Alumno.nombre_apellido,Alumno.edad,Alumno.
   altura);
}
```

```
void imprimeDatosAlmuno2 (struct datosAlumno *Alumno){
   printf("La edad del alumno %s es %d y su altura es
   %3.2f\n",Alumno->nombre_apellido,Alumno-
   >edad,Alumno->altura);
}
```

Ejercicio

- Gestión de biblioteca
  - Escribir un programa que permita:
    - La introducción de nuevos libros. Para cada uno de ellos se debe almacenar la información incluida en el ejemplo anterior.
    - Un listado completo de todos los libros de la biblioteca.
    - La opción de búsqueda de libros por nombre del autor.

```
struct libro_t
{
   char titulo [20];
   char ISBN [13];
   int nPags;
   char editorial[20];
   struct autor_t autor;
};
```

```
struct autor_t
{
   char nombre [10];
   char apellido [10];
   char nacionalidad[10];
};
```

Estructuras: casting

- **Casting**: (casting de tipo)
  - No existe el casting de estructuras, no tiene sentido. Es necesario crear un método que lo transforme "a mano" de un tipo de estructura a otro.

- Excepción: Casting de punteros a estructuras (casting binario)
  - Dado que los punteros son sólo direcciones de memoria, sí se pueden realizar casting de punteros a estructuras.

Puntero a estructuras

El uso de punteros a estructuras puede suponer un ahorro de memoria considerable y por tanto su uso puede ser conveniente

- Veamos el siguiente ejemplo: ¿Y si hay dos libros con el mismo autor?

```
libro1
{
   Nombre1
   ...
   Autor1
      NombreAutor1
      Apellido
      ….
 }
```

```
libro2
{
Nombre2
...
Autor1
      NombreAutor1
      Apellido
      ….
}
```

  - Se está almacenando dos veces la misma información.
  - Si se quiere editar el nombre del autor, será necesario buscar todos los libros que lo compartían y modificarlos.

Puntero a estructuras

- La solución está en definir un puntero que apunte a la estructura y utilizando dicho puntero, acceder a sus campos:

```
struct Alumno
{
   int numlista;
   char nombre [30];
};
struct Alumno JJ;
struct Alumno *pJJ;
…
pJJ = &JJ;
(*pJJ).numlista = 5;
strcpy ((*pJJ).nombre,"Juan");
```

Para hacer menos incómodo el trabajo con punteros a estructuras, C tiene el operador flecha → que se utiliza de la forma:

ptr->campo que es equivalente a:
(*ptr).campo

El ejemplo anterior quedaría así

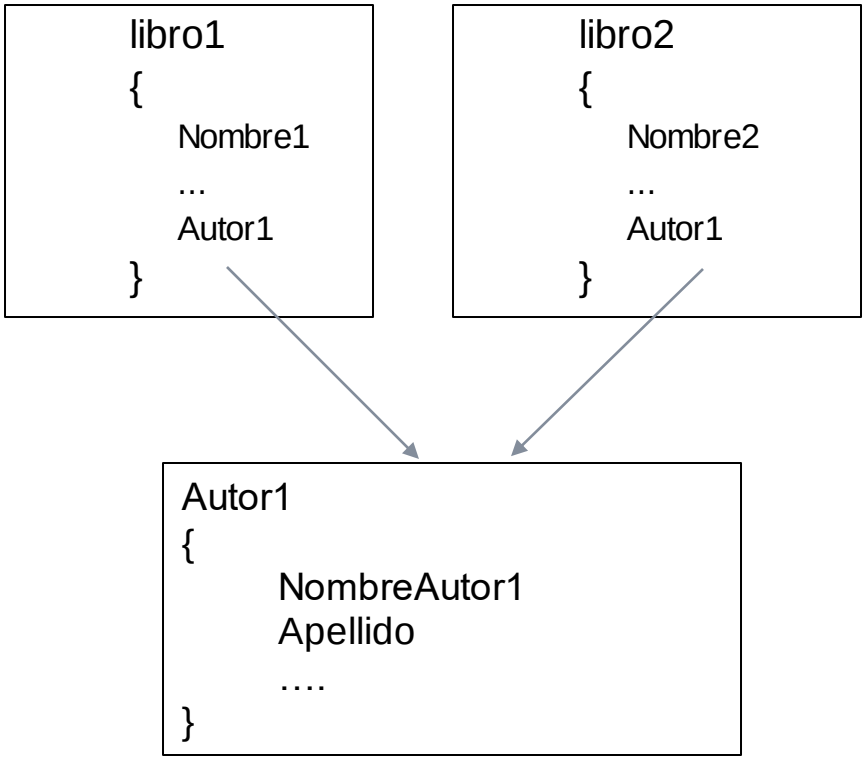
```
pJJ = &JJ;
PJJ->numlista = 5;
strcpy (pJJ->nombre,"Juan");
```

Puntero a estructuras: Ejemplo

- Por tanto, conveniente almacenar punteros a estructuras cuando se requiera:

```
struct libro_t
{
  char titulo [20];
  char ISBN [13];
  int nPags;
  char editorial[20];
  struct autor_t *autor;
};
```

- Antes:
  - Sizeof (libro_t)=87 bytes
- Ahora:
  - Sizeof (libro_t)=65 bytes



Ejercicio 1 (estructuras)

- Escribir una aplicación que pida datos de un usuario por consola, y los almacene en una estructura.
- Una vez leídos esos datos, se mostrarán por pantalla.
- Será necesario almacenar los siguientes datos:
  - Nombre (10 carácteres MAX)
  - Apellido1 (15 carácteres MAX)
  - DNI (9 carácteres MAX)
  - Edad (Número entero)
  - Peso (Número en coma flotante)
  - Teléfono (9 caracteres MAX)
- Desarrollo:
  - Definir la estructura "`struct usuario_t`", con los datos pedidos anteriormente.
  - Crear una función que reciba una copia de la estructura, y la imprima por pantalla. Éste método tendrá la siguiente cabecera:
    - `void ImprimeUsuario( struct usuario_t)`
  - Crear una función que imprima los usuarios más jóvenes.

Ejercicio 2 (estructuras)

- Números complejos
  - Los números complejos son un tipo de números formados por dos componentes:
    - Parte real (números en coma flotante)
    - Parte imaginaria (números en coma flotante)
  - Se pide implementar funciones que permitan el almacenamiento y operaciones con números complejos
    - Definición de estructuras necesarias para almacenar datos
    - Definición e implementación de las siguientes operaciones sobre números complejos:
      - Suma, resta, multiplicación y división
    - Cada operación implementada, recibirá como parámetros dos números complejos, y devolverá como resultado un número complejo
  - Implementar un programa "main" que reciba por parámetros los operandos y la operación, usando el siguiente formato:
    - `./complejos.exe <operación> <real operando 1> <imag operando 1> <real operando 2> <imag operando 2>`
    - Los tipos de operación serán un sólo carácter : + - * / (suma resta multiplicar dividir)
    - Los operandos son números en formato coma flotante
    - Ej:
      - ./complejos.exe + 3.5 2.0 1.5 1.0
      - El resultado es : 5.5+3.0i

Ejercicio 2

- Operaciones con números complejos:
  - a=parte real primer número
  - b=parte imaginaria primer número
  - c=parte real segundo número
  - d=parte imaginaria segundo número

- Adición

$$(a,b)+(c,d)=(a+c,\ b+d)$$

- Producto por escalar

$$r(a,b)=(ra,\ rb)$$

- Multiplicación

$$(a,b)\cdot(c,d)=(ac-bd,ad+bc)$$

- Resta

$$(a,b)-(c,d)=(a-c,\ b-d)$$

- División

$$\frac{(a,b)}{(c,d)}=\frac{(ac+bd,\ bc-ad)}{c^2+d^2}=\left(\frac{ac+bd}{c^2+d^2},\frac{bc-ad}{c^2+d^2}\right)$$

Ejercicio 2 (estructuras)

- Suma de dos números complejos:
  - Resultado.real=num1.real+n2.real
  - Resultado.imaginario=num1.imaginario+num2.imaginario
- Resta de dos números complejos:
  - Resultado.real=num1.real-n2.real
  - Resultado.imaginario=num1.imaginario-num2.imaginario
- Multiplicación:
  - Resultado.real=num1.real*num2.real-num1.imaginario*num2.imaginario
  - Resultado.imaginario=num1.real*num2.imaginario+num1.imaginario*num2.real
- División:
  - Resultado.real=(num1.real*num2.real+num1.imaginario*num2.imaginario)/ (num2.real^2 + num2.imaginario^2)
  - Resultado.imaginario=(num1.imaginario*num2.real-num1.real*num2.imaginario)/ (num2.real^2 + num2.imaginario^2)

Uniones

- Ya hemos visto que en una estructura, cada campo tiene espacio en memoria para almacenar su valor, pero hay situaciones especiales en la que las estructuras de datos pueden desperdiciar memoria.

- Supongamos que una aplicación puede identificar a los usuarios mediante uno solo de los siguientes posibles cinco datos:

    - NIF: ocho dígitos seguidos de una letra.
    - CIF: letra seguida de 8 dígitos.
    - Pasaporte: ocho letras y/o números.
    - NIE: letra seguida de 7 dígitos y una segunda letra.
    - Nombre de usuario: cadena de texto de hasta 16 letras.

Una posible estructura de datos para almacenar esta información se muestra en la siguiente slide:

Uniones

```
struct user_id {
    struct NIF {
        unsigned int number;
        char ch;
    } nif;

    struct CIF {
        unsigned int number;
        char ch;
    } cif;

    char Passport[8];

    struct NIE {
        char prefix;
        unsigned int number;
        char suffix;
    } nie;
    char username[16];
};
```

Declaración

Definición

Declaración

Definición

Declaración

Definición

Si asumimos que los enteros ocupan 4 bytes y las letras 1 byte**, la estructura necesita 40 bytes para ser almacenada en memoria**.

Pero de todos estos campos sólo uno contendrá información (dependiendo de la identificación que tenga cada usuario), el resto estarán vacíos.

Esto quiere decir que la estructura de datos sólo utilizará entre un 12.5% y un 40% del espacio que ocupa.

Más de la mitad de la memoria se desperdicia!!!

Uniones

La declaración de la estructura **user_id** anterior es equivalente a:

```
struct NIF {
	unsigned int number;
	char ch;
};

struct CIF {
	unsigned int number;
	char ch;
};

struct NIE {
	char prefix;
	unsigned int number;
	char suffix;
};
```

```
struct user_id {

	struct NIF nif;
	struct CIF cif;
	char Passport[8];
	struct NIE nie;
	char username[16];
};
```

Uniones

- C ofrece un tipo de dato en el que todos los campos comparten el mismo espacio de memoria y que está pensada precisamente para aquellos casos en las que sólo uno de esos campos se utiliza en cada momento.
- Esta estructura se define reemplazando la palabra reservada "struct" por "**union**"
- Cuando se define una "union" se reserva sólo el espacio del mayor de los campos. Los datos se almacenan comenzando por la misma posición de memoria y con la estructura del campo seleccionado.
- Esta construcción no guarda en ningún lugar cuál de los campos se está utilizando. Si esa información es necesaria, el programador debe almacenarla en una estructura de datos auxiliar.
- El acceso a los campos de una "union" se realiza igual que en una estructura

```
union user_id {
     struct NIF {
          unsigned int number;
          char ch;
     } nif;

     struct CIF {
          unsigned int number;
          char ch;
     } cif;

     char Passport[8];

     struct NIE {
          char prefix;
          unsigned int number;
          char suffix;
     } nie;
     char username[16];
};
```

Estructuras vs Uniones

- La forma de almacenamiento es diferente
  - Una estructura (**"struct"**) permite almacenar variables relacionadas juntas y en posiciones contiguas de memoria
  - Las uniones (**"union"**) almacenan miembros múltiples en un paquete, pero en lugar de situar sus miembros unos detrás de otros, todos los miembros se solapan entre sí en la misma posición

Unión. Declaración

- El formato de la declaración especifica un nuevo tipo de dato:

```
union <nombre de la union>
{
<tipo de datos miembro1> <nombre miembro1>;
<tipo de datos miembro2> <nombre miembro2>;
…
<tipo de datos miembroN> <nombre miembroN>;
};
```

- La cantidad de memoria reservada para la unión es igual a la anchura de la variable más grande.
- En el tipo unión, todos los miembros ocupan la misma posición de memoria, la del miembro de mayor tamaño

Ejemplo 1 - Uniones

- Crear las siguientes dos estructuras para cargar de usuario pedidos por terminal:

```
union datos_u{
 int edad;
 float peso;
 char nombre[10};
} ;
```

```
struct datos_t{
  int edad;
  float peso;
  char nombre[10];
};
```

  - Crear un programa que pida por pantalla los datos dos veces. La primera vez usando "datos_t", y la segunda vez usando datos_u.
  - Mostrar los resultados leídos de ambas estructuras ¿Qué observas?
  - ¿Cuanto ocupa la estructura datos_t? ¿Y datos_u? ¿Qué observas?

Ejemplo 2 - Uniones

- Las uniones son útiles para ahorrar memoria en caso de no necesitar todos los registros de una estructura:

```
#define INT   1
#define FLOAT 2
#define CHAR   3

struct lectura_datos
{
   int tipo; //INT, FLOAT o CHAR
   int i;
   float f;
   char str[4];
};
```

- Usando la estructura anterior, desarrollar un programa que pregunte al usuario el tipo de dato que introducirá (integer, float o char[3]), y que luego lea los datos introducidos por el usuario, almacenándolos en esa estructura.
  - Crear una función que pida los datos y rellene esa estructura.
  - Crear una función que muestre el contenido de la estructura "lectura_datos" por pantalla, en función del tipo de dato que ha almacenado (solo muestra integer si el tipo es "INT", por ejemplo).

- Modificar la estructura Para que las variables "I", "f" y "str" ocupen la misma zona de memoria.

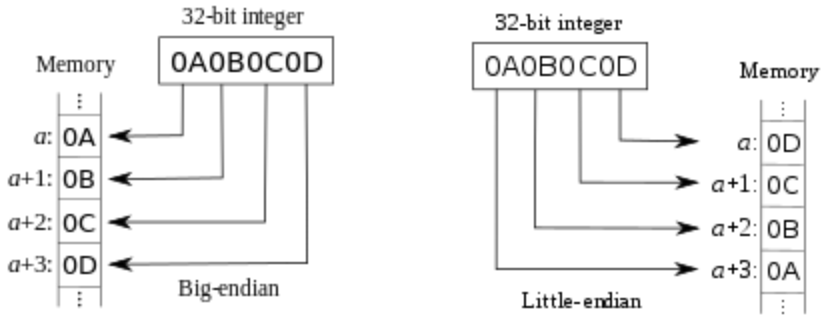
Ejemplo 3 - Uniones

- Cómo implementar una función de "byteswap" o saber si nuestro computador es una arquitectura Big-Endian o Little-Endian.
- Vamos a ver un ejemplo de como resolver un ejemplo de este estilo utilizando UNIONES y para ello introduciremos antes el concepto de que son los formatos "BigEndian" y "LittleEndian".
- Son formatos que hacen referencia a como se almacenan los datos en la memoria del ordenador. "**BigEndian**" es aquel formato que ordena los bytes desde el más significativo al menos significativo desde las direcciones de memoria más bajas a las más altas, mientras "**LittleEndian**" lo hace desde el menos significativo al más significativo.
- BigEndian es el utilizado por ejemplo en arquitecturas PowerPC (IBM, Apple) mientras que LittleEndian es adptado en la mayoría de los procesadores Intel y ARM.

Ejemplo 3 - Uso de uniones

Función byteswap:

- Una función de "byteswap" permite intercambiar el orden de los bytes con que se representan los datos básicos en nuestro ordenador. Útil cuando se trabaja con diferentes tipos de arquitecturas (BigEndian/LittleEndian)



Utilizando una "union" de este estilo podríamos implementar una función de byteswap:

```
union bswap32{
        int data;
        char bytes[4];
};
```

Ejemplo 3 – Uso de uniones

- Ej3.1 : Averiguar si la máquina en la que trabajamos es "BigEndian" o "LittleEndian"
  - Crear una variable de tipo "bswap32", y guardar un "1" en el miembro de la unión data.
  - Si bytes[0]==1 -->LittleEndian
    - Else -->BigEndian
- Ej3.2: Realizar una función que intercambie los bytes para pasar de "BigEndian" a "LittleEndian" y viceversa.

Typedef

- Un **typedef** permite crear un alias o sinónimo de un tipo de dato definido por el usuario o de un tipo ya existente.

- La palabra reservada **typedef** nos permite crear nuevos tipos de datos dentro del lenguaje C. De esa manera extendemos el lenguaje del que partimos, creando nuevas palabras reservadas para nuestro programa.

```
typedef int tipoEntero; // Usaremos la nueva palabra reservada "tipoEntero" para declarar integers
```

- También nos permite crear "atajos" a tipos de datos compuestos.

```
typedef unsigned int tipoEnteroSinSigno; // Usaremos la nueva palabra reservada
                                         //  "tipoEnteroSinSigno" para declarar  integers sin signo
```

- Su función es similar a la de las macros de tipo "#define", ambas sustituyen una palabra reservada por un valor. Su principal diferencia reside en que las macros son "sólo" reglas de sustitución, mientras que los tipos "typedef" heredan todas las características de tipado de variables.

Typedef

- Por último, uno de los usos más comunes es el de "simplificar o agilizar" el uso de los tipos estructurados de datos. No sólo nos ahorramos la palabra "struct", sino que creamos un tipo nuevo de dato con las propiedades de tipado de C:

```
struct datos_t{
 int edad;
 float peso;
 char nombre[10];
};
```

```
struct datos_t variable;
```

Nombre de la estructura

```
typedef struct datos_t {
  int edad;
  float peso;
  char nombre[10];
}datos_t;
```

el tipo creado con typedef

```
datos_t variable;
```

- El nombre de la estructura y el tipo no tienen porqué ser iguales
- Y al hacer el typedef el nombre de la estructura se puede omitir.

Ejemplo 1 - Typedef

- Se pide realizar una cabecera que contenga los siguientes tipos de datos, declarados con typedef:

  Int64 : Es un tipo de datos “long long int”
  Int32: tipo de datos “int”
  Int16: tipo de datos “short”
  Byte: tipo de datos “char”

- Declarar una estructura: arbol_t, que contendrá los siguientes campos:
  - numeroHojas: Tipo de datos int64
  - numeroRamas: tipo de datos integer
  - numeroRaices: tipo de datos short
  - nombreArbol: Array de 10 bytes

- Se pide implementar las siguientes funciones:

Ejemplo 1 - Typedef

- **arbol_t crearArbol();**

  Esta función pedirá los datos al usuario necesarios para rellenar una variable de tipo arbol_t, y la devolverá como resultado.

- **void mostrarArbol(arbol_t arbol);**

  Esta función mostrará por terminal cada una de los campos de la estructura arbol_t. Por cada campo, deberá mostrar una línea con el nombre del campo y el valor que tiene.

- El main del programa ofrecerá la siguiente funcionalidad:

  - El programa mostrará al inicio un menú con tres opciones: “Crear árbol”, “Mostrar árbol” y “Salir”. Cada una de esas opciones invocará a la funciones decaradas anteriormente, y en caso de elegir “salir” se terminará el programa.

Enumeraciones

- A veces es necesario utilizar valores de constantes cuyo valor realmente no importa y lo que importa son los casos que pueden denotar. Por ejemplo, constantes que denoten el estado civil: soltero, casado, viudo,… En este caso lo importante no es el valor de la constante sino que el estado sea uno de los valores válidos.

- Las **enumeraciones** son un tipo de C que nos permiten asignar automáticamente valores enteros a una colección de nombres. Por tanto, una enumeración es un tipo definido por el usuario con constantes de nombre de tipo entero.

- Las enumeraciones sirven para extender el lenguaje C de manera que podamos "contar", "ordenar" y "seleccionar variables que representan conceptos complejos.

Enumeraciones

- En la declaración de un enum se escribe una lista de valores que internamente se asocian con las constantes enteras. La asociación que por defecto se hace es de forma ordenada (0,1,2,…), es decir, enumerador1=0, enumerador2=1.....

```
enum
{
    enumerador1, enumerador2, …, enumeradorN
};
```

- A la enumeración se le puede asignar un nombre, que se comportará como un nuevo tipo de dato que solo podrá contener los valores especificados en la enumeración

```
enum nombre
{
    enumerador1, enumerador2, …, enumeradorN
};
```

Enumeraciones

- En la declaración del tipo enum puede asociarse a los identificadores valores constantes diferentes en vez de la asociación que por defecto se hace (0,1,2,..). Para ello se utiliza el formato:

```
enum nombre
{
   enumerador1 = expresión_constante1,
   enumerador2 = expresión_constante2,
   …,
   enumeradorN = expresión_contanteN
};
```

- Los identificadores del enumerado (enumerador1, enumerador2, …, enumeradorN) no se pueden introducir por teclado ni se pueden escribir directamente en pantalla.

Enumeraciones

- Enumerado de estaciones del año

  enum {primavera, verano, otonno, invierno}

  Cambiando la asignación por defecto de los valores enteros de las constantes:

```
enum estaciones
{
    primavera = 1,
    verano = 2,
    otonno = 3,
    invierno = 4
};
```

Enumeraciones

- Las enumeraciones facilitan el manejo de los datos ya que es más fácil recordar un nombre que un valor entero
- Las declaraciones pueden hacerse dentro o fuera del main
- Es habitual encontrar las enumeraciones como herramienta para generar menús. De esta forma se asigna a cada identificador (que son las opciones para el usuario) un número entero.
- Las enumeraciones admiten como máximo 256 elementos

Ejemplo 1 - Enumeraciones

- Queremos definir una estructura “hoja”, que va a tener su tamaño, grosor y color. Los colores posibles pueden ser “amarillo”, “verde” o “marrón”. ¿Como podemos modelarlo?

```
typedef struct hoja{
        float tamanio;
        float grosor;
        ¿?¿? Color;
} hoja_t;
```

- Necesitamos tipos de variables que definan los colores que el usuario puede elegir, usamos “enums”:

```
typedef enum
        {amarillo,verde,marron} color_e;
```

- Ahora nuestra estructura puede usar un nuevo tipo de variable “color_e”:

```
typedef struct hoja{
        float tamanio;
        float grosor;
        color_e color;
}hoja_t;
```

Ejemplo 1 - Enumeraciones

- Las variables de tipo "**color_e**" podrán tomar alguno de los valores predefinidos en el enumerado. Cada una de las palabras usadas al declarar el nuevo tipo pasan a ser palabras reservadas: No se podrán usar en variables:

```
hoja_t        nuevaHoja;

nuevaHoja.tamanio  = 0.5;
nuevaHoja.grosor=  0.3;
nuevaHoja.color  =verde;
```

- Internamente, los tipos enum se representan como enteros. Recordar que el valor de cada uno de los enumerados usados está dado por el orden de aparición en el tipo, empezando por "0".

Ejemplo 1 - Enumeraciones

- Es decir, se cumple que:

```
typedef enum{
        amarillo,verde,marron}  color_e;
```

amarillo=0 verde=1 marron=2

- Por lo tanto, dado que son valores "integer", las siguientes asignaciones son válidas:

```
nuevaHoja.color=0   // --> equivale al amarillo
```

- Podemos forzar su valor de inicialización al declarar el tipo:

```
typedef enum{
        amarillo=1,verde=3,marron=6}  color_e;
```

Ejemplo 1 - Enumeraciones

- Se pide, partiendo del código realizado en el Ejemplo de typedef (en el que se ha creado la estructura de tipo arbol_t):
  - Crear un tipo de datos enumerado “tipoArbol_e” que permita describir los siguientes árboles:
    - pino, haya, roble, eucalipto
  - Modificar la estructura “arbol_t” para que el campo “nombreArbol” sea de tipo “tipoArbol_e”.
  - Modificar el programa para que, al rellenar la estructura de tipo arbol_t, se pidan los tres tipos de árboles por pantalla, e inicializar el campo “nombreArbol” acorde con el dato introducido por el usuario.
  - Igualmente, modificar el método “mostrarArbol” para que muestre correctamente el campo “nombreArbol” de tipo “tipoArbol_e” por pantalla.