ESTRUCTURAS

- **Estructuras** (también llamadas **registros**): permite agrupar con un mismo nombre (**identificador**) datos de **igual** o **distinto tipo**. Las variables declaradas dentro de la estructura son sus **miembros** (**campos**). Una vez declarada se utiliza como los datos simples. Se pueden definir arreglos de estructuras o incluso utilizarlas en forma anidada (declarar un miembro que sea a su vez la misma estructura).

  Ejemplo: se desea almacenar la información de personas, con su año de nacimiento (int), nombre completo (char []), sueldo/promedio (float) y nacionalidad ('A'/'E') (char).

- La **declaración** de un tipo **struct** se puede realizar de tres formas:
  - Sintaxis 1: se declara la estructura y se define la variable al mismo tiempo.
  - Sintaxis 2: se declara primero la estructura (generalmente en un *header*) y después se pueden definir las variables sin tener que repetir la estructura.
  - Sintaxis 3: se declara un nuevo tipo de datos mediante **typedef**, que luego se puede utilizar como cualquier otro tipo definido de C (recomendado).

ESTRUCTURAS – FORMA 1

- La **declaración** se realiza indicando el tipo de dato y nombre de cada campo que contiene, y a continuación se definen las variables que se utilizan.

```
struct  nombre_estructura
{
        tipo_1  elemento_1;
        tipo_2  elemento_2;
        ⋮
        tipo_n  elemento_n;
} var1, var2 … varN;            // se genera la reserva de memoria
```

Ejemplo:

```
struct  persona
{
        int   nacimiento;
        char  nombre[30];
        float  cantidad;
        char   nacion;
} alumno,  empleado, … contacto;
```

ESTRUCTURAS – FORMA 2

- Similar al caso anterior, pero se realizan dos pasos:

1. Declaración de la estructura (en general en un archivo de cabecera)

```
struct nombre_struc                    // no se reserva memoria
{
    tipo_1  elemento_1;
    tipo_2  elemento_2;
    ⋮
    tipo_n  elemento_n;
} ;
```

2. Declarar una (o más) variables del tipo definido por la estructura

```
struct nombre_struc  variable1,      // se asigna memoria para cada
                     variableN;      // variable de tipo nombre_struc
```

Ejemplo:

```
struct  persona  alumno,  empleado, contacto;
```

ESTRUCTURAS – FORMA 3

- C permite poner nombre a los tipos de datos definidos por el usuario mediante el operador ***typedef***.

- Declaración del nuevo tipo:

```
typedef struct  [nombre_struc]      // el nombre es opcional
{
     tipo_1  elemento_1;
     tipo_2  elemento_2;
     ⋮
     tipo_n  elemento_n;
} nuevo_tipo;
```

declarada previamente

```
/****  tambien se puede typedef estruct prototipo_anterior nuevo_tipo  ****/
```

- Declaración de las variables del tipo definido por el usuario:

```
nuevo_tipo  variable1,          // se asigna memoria para cada
            variableN;          // variable de nuevo_tipo (nombre_struc)
```

Ejemplo: tipo_persona alumno, empleado, contacto;

ACCESO A LOS DATOS

- Con las estructuras se puede trabajar a dos niveles, con la estructura completa o con sus **campos** en forma independiente.
- El **campo** de una estructura es una variable de un tipo determinada y se trata como tal. Se accede a su valor mediente el **operador punto** '.':

  <variable de tipo estructura>.<nombre del campo>

- La asignación de valores de puede hacer de distintas formas;

  1. En el momento de la declaración (inicialización).
  2. Utilizando sentencias de asignación para cada elemento.
  3. Mediante operaciones de lectura de cada uno de los campos.
  4. Por copia de una estructura idéntica.

- El tamaño de una estructura siempre es mayor o igual a la suma de los tamaños de sus miembros (se utilizan bytes para alinear los datos).

ASIGNACION DE VALORES

1. En el momento de la declaración (inicialización):

```
struct   persona
{
    int   nacimiento;
    char  nombre[30];
    float  cantidad;
    char   nacion;
};
```

```
typedef   struct
{
    int   nacimiento;
    char  nombre[30];
    float  cantidad;
    char   nacion;
} persona;
```

```
struct  persona empleado = {1965,  "Juan Jacinto Jimenez", 12345.67, 'A'};
```

```
persona empleado = {1965,  "Juan Jacinto Jimenez", 12345.67, 'A'};
```

2. Utilizando sentencias de asignación para cada elemento:

```
empleado.nacimiento = 1965;
empleado.nombre = "Juan Jacinto Jimenez";   // no se puede realizar!!
empleado.cantidad = 12345.67;
empleado.nacion = 'A';
```

ASIGNACION DE VALORES

3. Mediante operaciones de lectura a cada uno de sus campos:

```
scanf("%d",  &empleado.nacimiento);
scanf("%s",  empleado.nombre);
scanf("%f",  &empleado.cantidad);
scanf("%c",  &empleado.nacion);
```

4. Por copia de una estructura idéntica:

```
struct persona alumno, empleado;
empleado.nacimiento = 1965;
empleado.nombre = "Juan Jacinto Jimenez";   // no se puede realizar!!
empleado.cantidad = 12345.67;
empleado.nacion = 'A';
alumno = empleado;                          // se copian todos los campos
```

Ejemplo acceso a los datos

- Escribir un programa que determine si dos puntos son iguales.

```
void main (void)
{
    struct coordenadas
    {
        float x;
        float y;
    } puntoA,  puntoB;

    printf("Coordenada x del punto A: "); scanf("%f", &puntoA.x);
    printf("Coordenada y del punto A: "); scanf("%f", &puntoA.y);
    printf("Coordenada x del punto B: "); scanf("%f", &puntoB.x);
    printf("Coordenada y del punto B: "); scanf("%f", &puntoB.y);

    if( (puntoA.x == puntoB.x) && (puntoA.y == puntoB.y) )
        printf("A y B son iguales \n");
    else
        printf("A y B son distintos \n");
}
```

ESTRUCTURAS ANIDADAS

- Un miembro de una estructura puede ser de tipo básico (int, float, char, …), pero también puede ser un array (vector o matriz), un puntero u otra estructura.
- Por ejemplo para tener un tipo de dato que represente un **triángulo**:

```
struct  punto
{
    int  coorx;
    int  coory;
};
```

```
typedef   struct
{
    int  color_linea;
    int  color_fondo;
    struct punto verticeA;
    struct punto verticeB;
    struct punto verticeC;
    char  tipo[20];
} triangulo;
```

- Se pueden declarar las variables del nuevo tipo:

```
triangulo figura1 = { 1, 15, {10, 10}, {10, 100}, {100, 100},  "rectangulo"};
```

EST. ANIDADAS: ACCESO A LOS DATOS

- Para accede a un campo que se encuentra anidado se utiliza el operador punto '.' (operador miembro) tantas veces como sea necesario para alcanzarlo.

Ejemplo: se quieren modificar los valores introducidos en la declaración de la variable figura1:

```
triangulo figura1 = { 1, 15, {10, 10}, {10, 100}, {100, 100}, "rectangulo"};
figura1.color_linea = 4;
figura1.color_fondo = 14;
figura1.verticeA.coorx = 20;
figura1.verticeA.coory = 20;
figura1.verticeB.coorx+= 50;
figura1.verticeB.coory+= 100;
figura1.verticeC.coorx*= 2;
figura1.verticeC.coory*= 2;
strcpy(figura1.tipo, "escaleno");
```

Ej. 3: triangulo

ESTRUCTURAS ANIDADAS: Ejemplo 2


```
struct  fecha
{
    int  dia;
    int  mes;
    int  anio;
};

struct  telefono
{
    int  pais;
    int  ciudad;
    int  numero;
};
```

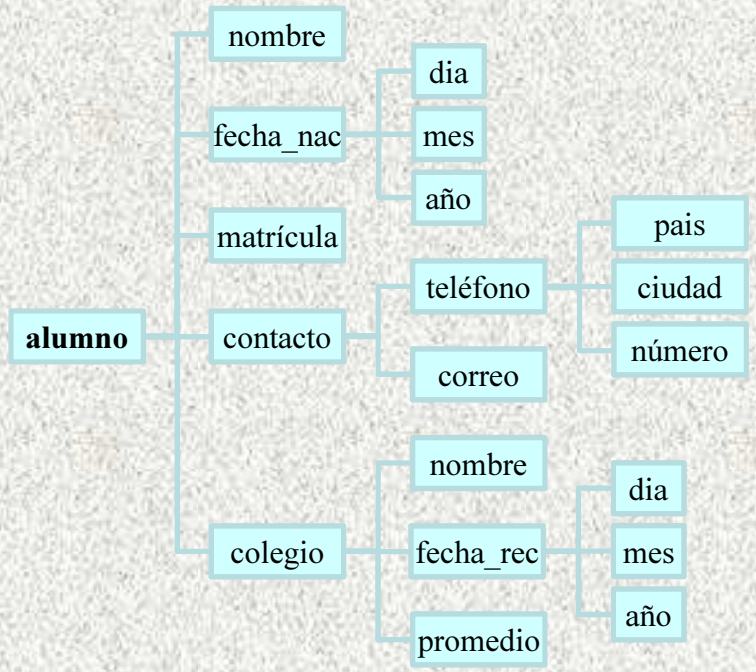
```
struct  contacto
{
    struct telefono tel;
    char correo[20];
};
```

```
struct  colegio
{
    char  nombre[20];
    struct fecha recibido;
    float  promedio;
} ;
```

```
struct  alumno
{
    char nombre[20];
    struct fecha fecha_nac;
    int  matricula;
    struct contacto info;
    struct colegio colegio;
} ;
```

EST. ANIDADAS: Ejemplo 2 – datos

```
#include “prototipos.h”
int main()
{
    struct alumno Paula;
    copia_string(Paula.nombre,"Paula Perez");
    Paula.matricula = 2002;
    Paula.fecha_nac.dia = 22;
    Paula.fecha_nac.mes = 05;
    Paula.fecha_nac.anio = 1993;
    Paula.info.tel.pais = 54;
    Paula.info.tel.ciudad = 223;
    Paula.info.tel.numero = 4816600;
    copia_string(Paula.info.correo,"paulap@fi.mdp.edu.ar");
    copia_string(Paula.colegio.nombre,"Comercial No 2");
    Paula.colegio.recibido.dia = 30;
    Paula.colegio.recibido.mes = 11;
    Paula.colegio.recibido.anio = 2001;
    Paula.colegio.promedio = 9.88;
    return 0;
}
```



Ej. 4: alumno

ARREGLO DE ESTRUCTURAS

- Los arrays de estructuras son vectores o matrices en los que cada uno de sus elementos es una estructura.
- Declaración:

1. struct nombre_struc nombre_array [dimensiones]
2. typedef struct { ……} nuevo_tipo;

   nuevo_tipo nombre_array [dimensiones]

- Se pueden inicializar también en el momento de la declaración.

1. struct punto vertices[5];

   struct punto vertices[3] = { 0,1,10, 20, 30,12};

   struct punto vertices[3] = { {0,1}, {10, 20}, {30,12} };
2. typedef struct alumno cursante;

   cursante cuatri1[15];

   cursante cuatri1[15] = { {“Jose Alvarez”, 10565, 22, 1, 1995, …}, … { … 6.25} };

ARREGLO DE ESTRUCTURAS – Ejemplo

- Almacenar la información (nombre, matrícula y materias rendidas) de los alumnos que cursan Programación Estructurada.

```
typedef  struct
{
    int dia;
    int mes;
    int anio;
} fecha;
```

```
typedef  struct
{
    char  nombre[20];
    fecha aprobado;
    float  promedio;
} catedras;
```

```
typedef  struct
{
    char nombre[20];
    int  matricula;
    catedras materia[30];
} datos_alumno;
```

```
void main (void)
{
   datos_alumno  alumnos[12];

   for(int cnt=0; cnt<12; cnt++)
      ingresar_datos;
     ⋮
   alumnos[5].matricula = 10528;                    // acceso a un campo en particular
   alumnos[3].materia[2].aprobado.anio = 2014;      // acceso a un campo en particular
     ⋮
}
```

Ej. 5: array_struct

ESTRUCTURAS COMO PARAMETROS

- Por defecto el paso de una estructura a una función se realiza **por valor**.
- Las modificaciones del valor de un campo de la estructura durante la ejecución de la función no son visibles fuera del alcance de esa función.
- En general las estructuras suelen ser datos de tipos complejos (por la variedad de tipos), lo que se dificulta el tratar de cambiar un valor particular mediante una función.
- Solución: pasar la estructura **por referencia** ➔ **PUNTEROS**.

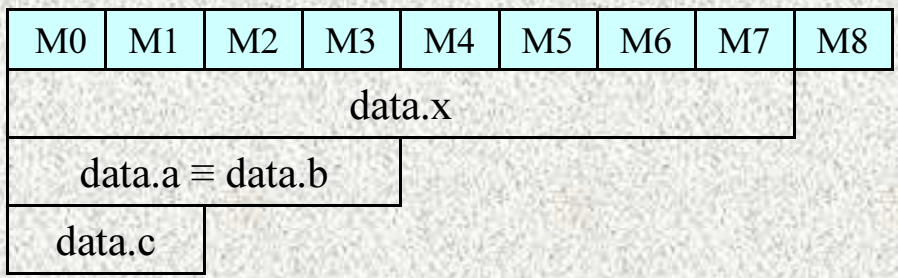
Ej. 6: struct_param

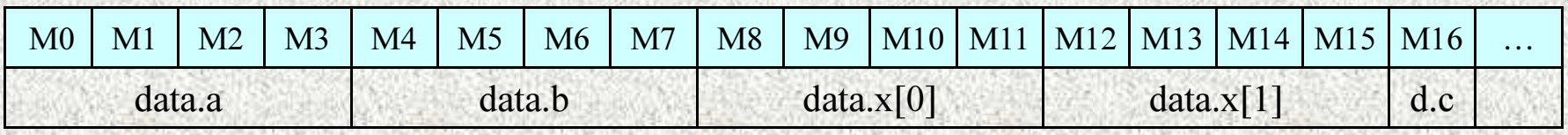
UNIONES

- Una **union** es similar a una **struct**, puede contener elementos de diferentes tipos y tamaños. La diferencia radica en que todos los campos comparten la **misma memoria**, si se modifica el valor de uno de ellos se pierden los valores del resto.
- El tamaño (**sizeof**) de una union es igual al tamaño del mayor de sus miembros.

```
struct datos
{
    int  a;
    int  b;
    int x[2];
    char c;
} data;
```

```
union datos
{
    int  a;
    int  b;
    int x[2];
    char c;
} data;
```





UNIONES – USOS

- Un ejemplo típico del uso de **union** se presenta en las planillas de cálculo donde cada celda puede contener un número, una fecha, texto ó fórmulas dependiendo de la aplicación. Para almacenar todas las celdas se puede utilizar una matriz (con dimensiones iguales a la de la hoja) de uniones.

- Se pueden utilizar para explorar como se almacena en memoria un valor:

```
typedef  union                          // se puede definir un nuevo tipo de dato
{
     float  fvalorint;
     char  mem[4];
} explora;

explora memoria;                        // defino una variable de tipo explora

memoria.fvalor = 2.3;

printf("\n\nEl valor %f se almacena en memoria = %x %x %x %x\n\n“,
                    memoria.fvalor, memoria.mem[0], memoria.mem[1],
                    memoria.mem[2], memoria.mem[3]);
```

UNIONES - INICIALIZACION

- Debido a que se almacena sólo un valor, las reglas para inicializar las uniones son diferentes que en el caso de las estructuras. Existen tres alternativas:

1. Copiar los elementos desde otra estructura del mismo tipo.
2. Asignarle un valor al **primer elemento** de la union.
3. En C99 se puede inicializar cualquier elemento indicándolo explícitamente.

```
union hold
{
    int digit;
    float bigfl;
    char letter;
};

union hold valA;
valA.letter = 'R';

union hold valB = valA;              // inicializa una union con otra
union hold valC = {88};              // inicializa el miembro digit
union hold valD = {.bigfl = 118.2};  // inicializa el miembro bigfl
```

UNIONES – ACCESO a los DATOS

```
union hold
{
    int digit;
    float bigfl;
    char letter;
} fit;

fit.digit = 735;          // 2 bytes  = df  02 00 00
fit.letter = 'A';         // 1 byte   = 41 02 00 00   - 'A' = 65 = 0x41.
printf("%d", fit.letra)   // imprime  'A'  porque toma un sólo byte , ignora el 02
printf("%d", fit.digit)   // imprime  577 = 0x241
fit.bigfl = 2.3;          // 4 bytes  = 33 33 13 40
fit.letter+= 1;           // se obtiene '4' = 0x34 en vez de 'B'
```

- Se debe considerar que el contenido de la union puede no ser el deseado.

Ej. 7: union

TIPO DE DATOS ENUM

- Mediante el uso de la palabra clave **enum** se crea un nuevo tipo de dato que permite declarar nombres simbólicos para representar constantes de tipo **int** y asignarles valores específicos. El objetivo es aumentar la legibilidad del programa.

- Para utilizar variables del nuevo tipo se realizan dos pasos:

1. Declaración de la estructura (en general en un archivo de cabecera). Los identificadores (nombre_elemento) son los posibles valores de la variable.

```
enum  nombre_tipo                    // no se reserva memoria
{
      nombre_elemento_1;             // corresponde a un valor = 0
      nombre_elemento_2;             // corresponde a un valor = 1
      ⋮
      nombre_elemento_n;             // corresponde a un valor = n
} ;
```

2. Declarar una (o más) variables del tipo definido en la enumeración

```
enum nombre_tipo  variable;          // se asigna memoria equivalente a un int
```

ENUM – DECLARACION

Ejemplo:

1. **enum** espectro
   {
   rojo,
   verde,
   azul
   };
2. **enum** espectro color;

- Se puede declarar el nuevo tipo y la variable simultáneamente:

```
enum espectro
{
    rojo,
    verde,
    azul
} color;
```

ENUM – ACCESO a los DATOS

```
int main()
{
    enum espectro {rojo, naranja, amarillo, verde, azul, violeta};
    enum espectro color;

    printf("El tamaño de color es = %d\n", sizeof(color));
    printf("El valor de amarillo es = %d\n", amarillo);
    return 0;
}
```

- Se pueden usar los nombres simbólicos como una variable int :

```
int  cfondo;
cfondo = azul;


if (cfondo == violeta)
    . . .;


for (cfondo =naranja; cfondo<violeta; cfondo++)
    . . . ;
```

ENUM – VALORES

- Por defecto los valores asignados a los nombres simbólicos son 0, 1, 2, …
- Se pueden asignar valores enteros a cada nombre simbólico (constantes) de acuerdo a la necesidad del usuario.

  enum niveles { low = 100, medium = 500, high = 900};
- Si se asigna un valor a una constante pero no a las siguientes, éstas serán numeradas consecutivamente a partir del último valor.

  enum precios {nulo, cien = 100, cien1, cien2, dosc = 200, dosc1};

  Al imprimir los valores se obtiene: nulo = 0
  cien = 100
  cien1 = 101
  cien2 = 102
  dosc = 200
  dosc1 = 201

Ej. 8: enum