Arrays

Índice

- Definición.
- Asignación.
- Arrays multidimensionales.
- Cadenas de caracteres - Strings.
- Funciones para el manejo de caracteres.
- Funciones para el manejo de strings.

Arrays - definición

- Un array es una estructura de datos que representa un conjunto de variables del *mismo tipo* declaradas como una sóla entidad.
- Un array tiene un tipo y tamaño:
  - El tipo especifica el tipo de valor almacenado en cada elemento del array.
  - El tamaño o dimensión especifica cuántos elementos se almacenan en el array. Los índices de un array en C empiezan en 0.
- Sintaxis de definición de arrays:

tipo_dato nombre_array[expresión_entera]; /*unidimensional*/

tipo_dato nomb_array[exp1] [exp2]...[exp*n*]; /*multidimensional*/

Ejemplos de definición de arrays

```
int a[100]; /* Vector de 100 enteros */
char texto[80]; /* Cadena de 80 caracteres */
float vel_x[30]; /* Vector de 30 reales */

float vx[4][4]; /* Matriz de 4x4 reales */
int pto3d_t[10][10][10][10]; /* Matriz de
        4 dimensiones (10 c/dim)de reales */
```

- Es habitual y ***recomendable*** la siguiente construcción:

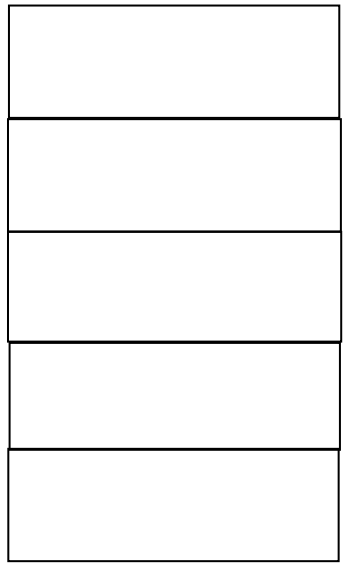
  #define DIMENSION 100

  ```
  ...
  float vel_x[DIMENSION]; /* Vector de reales */
  ```

Arrays

- Cuando se define un array, el compilador separa un conjunto de posiciones contiguas de memoria.
- El *contenido inicial* de un array es indefinido (valores basura).
- Ejm: int a[10];

elemento	*dirección*	*contenido*
a[0]	0F00	
a[1]	0F02	
a[2]	0F04	
…..	…..	
a[9]	0F12	



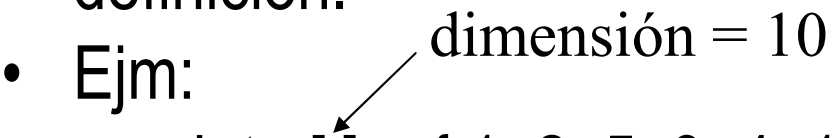
Arrays - asignación

- Se pueden asignar valores a los elementos individuales del array accediendo a cada elemento mediante el *índice* del array.
- Ejm: int a[10];

	elemento	*dirección*	*contenido*
a[0] = 1;	a[0]	0F00	1
a[1] = 5;	a[1]	0F02	5
a[2] = 120;	a[2]	0F04	120
....	.....	.....	
a[9] = a[2] + a[1];	a[9]	0F12	121

Arrays – definición e inicialización

- Los arrays pueden *inicializarse* en el momento de su definición.
- Ejm:




```
int a[ ] = { 1, 2, 5, 3, 4, 10, 8, 9, 7, 10 };
float x[ 6 ] = { 0, 0.32, 1.414, 2.7172, 3.1415, 5.4 };
char color[ 4 ] = { 'R', 'O', 'J', 'O' };
```

- Es un error colocar más elementos que los de la dimensión del array. Si se colocan menos elementos, los demás elementos se inicializan a *cero*.
- Si un array no se dimensiona y se inicializa durante su definición, el compilador le asigna una dimensión según el número de elementos iniciales.

Arrays multidimensionales

- Un array multidimensional se define con corchetes consecutivos. La forma general de su definición es:

  tipo_dato nombre_array[dim1][dim2]. . .[dimN];
- Un array multidimensional se almacena como una secuencia de elementos unidimensionales. Se considera como un array de arrays.
- Si asignan valores en la definición ésta se realiza variando el último índice más rapidamente.
- Ejm:

```
int matriz[ 2 ][ 3 ] = { { 0, 1, 2}, { 3, 4, 5} };
```

Ejemplo 1: frecuencia de ocurrencia dígitos

```
/*****************************************************************\
* Programa: frec_ocurr_digitos.c                                  *
* Descripción: Prog. que cuenta la frecuencia de ocurrencia de digitos  *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>

main()
{
  int c,i;
  int ndigit[10] = { 0};     /* Inicializacion express del array */

  printf ("Escribe numeros y pulsa Enter. Para terminar Ctrl-Z Enter\n");
  while ((c=getchar())!=EOF)
    if (c>='0' && c<='9')  /* Si c es un digito */
       ++ndigit[c-'0'];    /* incrementa el contador del digito c-'0' */

  printf("Frecuencia de ocurrencia de digitos \n");
  for (i=0; i<10 ; ++i)
     printf(" %d = %d\n", i, ndigit[i]);

  return 0;
}
```

Ejemplo 2: adivina números

```
/*****************************************************************\
* Programa: busq_binaria.c                                        *
* Descripción: Prog. que genera N numeros random y permite que el *
*              usuario adivine si esta un numero. Ordena por burbuja *
*              y usa busqueda binaria                             *
\*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10

main()
{
  int lista[N], num, encontrado;
  int i, j, t;
  int izq, der, mitad;

   /* Crea lista de valores random entre 0 y 100.
      rand() devuelve un numero random entre 0 y RAND_MAX */
   srand( (unsigned)time( NULL ) );  /* semilla para random */
   for( i = 0;   i < N;i++ )
      lista[i] = (float)rand()/RAND_MAX*100;
```

Ejemplo 2: adivina números

```
/* Ordenacion por metodo de burbuja para aplicar busqueda binaria  */
   for(i = N-1; i > 0 ; i--)
     for(j = 0 ; j < i ; j++)
      if (lista[j] > lista[j+1])
       { t = lista[j]; lista[j] = lista[j+1]; lista[j+1] = t; }
   printf("He generado una lista de 10 numeros enteros entre 0 y 100.\n \
Adivina los numeros de la lista (Para terminar pulsar tecla no digito)\n");
   while (scanf(" %d", &num)==1)
   {
     izq = 0; der = N-1; encontrado = 0;
     while (izq <= der)
     {
         mitad = (izq + der)/2;
         if (num < lista[mitad])  der = mitad - 1;
         else if (num > lista[mitad])  izq = mitad + 1;
         else {
            encontrado = 1; printf(" %d si esta\n", num); /* elem. en lista */
           break;
          }
     }
     if (encontrado == 0) printf(" %d no esta\n", num);   /* no encontrado */
   }
   return 0;
}
```

Cadenas de caracteres - Strings

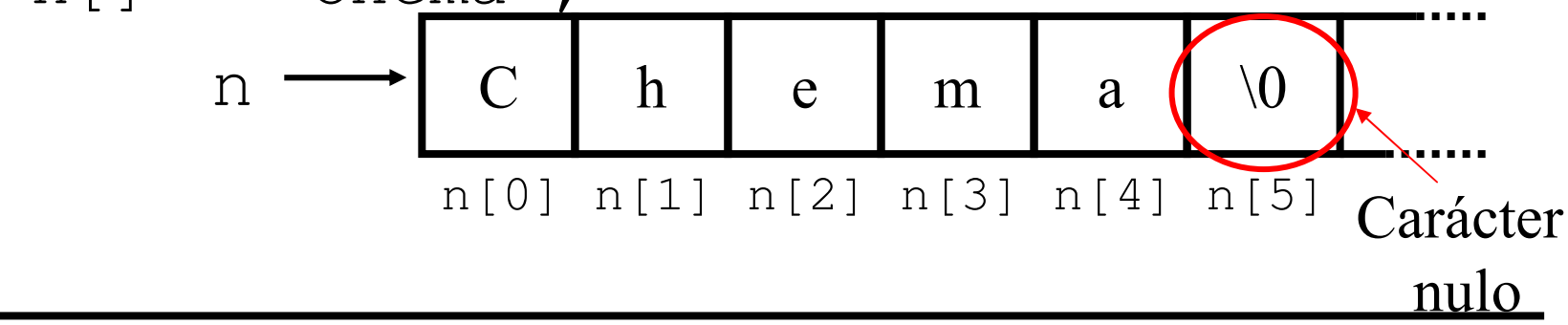
- Un *string* es un tipo especial de array donde cada uno de los elementos es un carácter.
- Ejm: /* declar. de un string de 10 caracteres */

  char cadena[10];
- Cuando se asigna una cadena de caracteres a un string se usa el **caracter nulo '\0'** para indicar el final de la cadena.
- Toda cadena debe usar un elemento de su dimensión para almacenar el caracter nulo, por tanto la dimensión efectiva se reduce en uno.

Strings – Definición e inicialización

- Para inicializar un string durante su definición se usan constantes de tipo cadena.
- Ejm:

```
char cad[6] = {'P','e','p','e',' ','\0'};
char cad[6] = "Pepe ";  /* con dimensión */
char cad[ ] = "Pepe ";  /* dimensión autom */
char n[] = "Chema";
```



Asignación de Strings

- Los strings no pueden asignarse. Por tanto, la asignación de strings por medio del operador (=) sólo es posible en la definición e inicialización del string .
- Ejemplo:

```
char n[50]="Chema";    /* Correcto */
n="Otro nombre"; /* Error: no es posible */
```

- C dispone de una serie de funciones para manipular strings, que se encuentran en la librería <**string.h**>

Lectura de Strings

- Mediante scanf() y gets(). Con sscanf() se lee desde un string.
- Ej:

```
#define MAX_STRING 128
…
char cadena[MAX_STRING];
float x; int n;
…
scanf("%s", cadena);
…
gets(cadena);
…
sscanf(cadena, "%f %d", &x, &n);    /* lee desde string */
```

Funciones para el manejo de caracteres

- Las funciones para el manejo de caracteres se encuentran en la librería **<ctype.h>**

Función	Descripción*	Función	Descripción*
isalnum(c)	(A-Z o a-z) o (0-9)	isprint(c)	Imprimibles incluido ' '
isalpha(c)	(A - Z o a - z)	ispunct(c)	Signos de puntuación
isascii(c)	0 - 127 (0x00-0x7F)	isspace(c)	espacios, (0x09-0x0D,0x20).
iscntrl(c)	(0x7F o 0x00-0x1F)	isupper(c)	(A-Z)
isdigit(c)	(0 - 9)	isxdigit(c)	(0 to 9, A to F, a to f).
isgraph(c)	Imprimibles menos ' '	tolower(c)	convierte c a minúscula
islower(c)	(a - z)	toupper(c)	convierte c a mayúscula

* Las funciones que empiezan por is.. devuelven 1 si c está en el rango especificado

Funciones para el manejo de strings

- Las funciones para el manejo de cadenas de caracteres se encuentran en la librería **<string.h>**

Función	Descripción
strcpy(s1, s2)	copia la cadena s2 en la cadena s1
strcat(s1, s2)	añade la cadena s2 al final de s1
strcmp(s1, s2)	compara s1 y s2. dev: 0 si s1=s2, num.pos. si s1>s2, num.neg si s1<s2.
strlen(s)	devuelve número de caracteres en la cadena s.
strcoll(s1, s2)	idéntico a strcmp numero positivo.
strxfrm(s1, s2, n)	guarda en s1 los n caracteres comparados con s2
strncpy(s1, s2, n)	copia n caracteres de s2 en la cadena s1

Funciones para el manejo de strings

Función	Descripción
strncat(s1, s2, n)	añade n caracteres de s2 al final de s1
strncmp(s1, s2, n)	igual que strcmp pero con n caracteres de s2
strchr(s, c)	busca el primer elemento de s igual a c
strrchr(s, c)	busca el último elemento de s igual a c
strstr(s1, s2)	busca la secuencia s2 en s1
strspn(s1, s2)	busca el primer elemento de s1diferente de s2
strcspn(s1, s2)	busca el 1er elem. en s1 igual a alguno de s2
strpbrk(s1, s2)	busca el 1er elem. en s1 igual a alguno de s2
strtok(s1, s2)	similar a strpbrk, se usa para partir cadenas
strerror(errcode)	devuelve un puntero al mensaje con código errcode

Ejemplo1: frecuencia de ocurrencia letras

```
/*****************************************************************\
* Programa: frecuencia_letras.c                                   *
* Descripción: Prog. que cuenta la frecuencia de ocurrencia de letras    *
\*****************************************************************/
#include <stdio.h>
#include <ctype.h>
#define N_LETRAS 27   /* incluye la ñ */

main()
{
  int c,i, nletras[N_LETRAS] = { 0};

  printf ("Introduce texto y pulsa Enter. Para terminar Ctrl-Z Enter\n");
  while ((c=getchar())!=EOF)
    if (isalpha(c)) {
      if (isupper(c)) c = tolower(c);
           ++nletras[c-'a'];
    }
    else if (c == 164 || c == 165)  ++nletras[26];  /* para la ñ */
  printf("Frecuencia de ocurrencia de letras \n");
  for (i = 'a'; i <= 'z' ; ++i) printf(" %c = %d\n", i, nletras[i-'a']);
  printf(" %c = %d\n", 164, nletras[26]);
  return 0;
}
```

Ejemplo2: uso de strcmp

```
/*****************************************************************\
* Programa: ejem_strcmp.c                                         *
* Descripción: Prog. que usa funciones de comparacion y copia de strings *
\*****************************************************************/
#include <string.h>
#include <stdio.h>

void main( void )
{
   char tmp[20]; int result;
   char string1[] = "El perro marron rapido salta sobre el zorro loco";
   char string2[] = "El PERRO marron rapido salta sobre el zorro loco ";

   /* Case sensitive */
   printf( "Compara cadenas:\n\t%s\n\t%s\n\n", string1, string2 );
   result = strcmp( string1, string2 );
   if( result > 0 )
      strcpy( tmp, "mayor que" );
   else if( result < 0 )
      strcpy( tmp, "menor que" );
   else
      strcpy( tmp, "igual a" );
   printf( "\tstrcmp:   String 1 es %s string 2\n", tmp );
}
```

Estructuras y uniones

Índice

- Definición de estructuras.
- Declaración y definición.
- typedef.
- Definición e inicialización.
- Acceso a los campos.
- Estructuras y arrays como campos de estructuras.
- Uniones, declaración y definición.
- Acceso a los campos de uniones.
- Campos de bits, declaración y definición.

Estructuras de datos básicas de C

El lenguaje C ofrece las siguientes estructuras de datos básicas, en adición a los arrays:

- Estructura o registro (`struct`).
- Unión de campos (`union`).

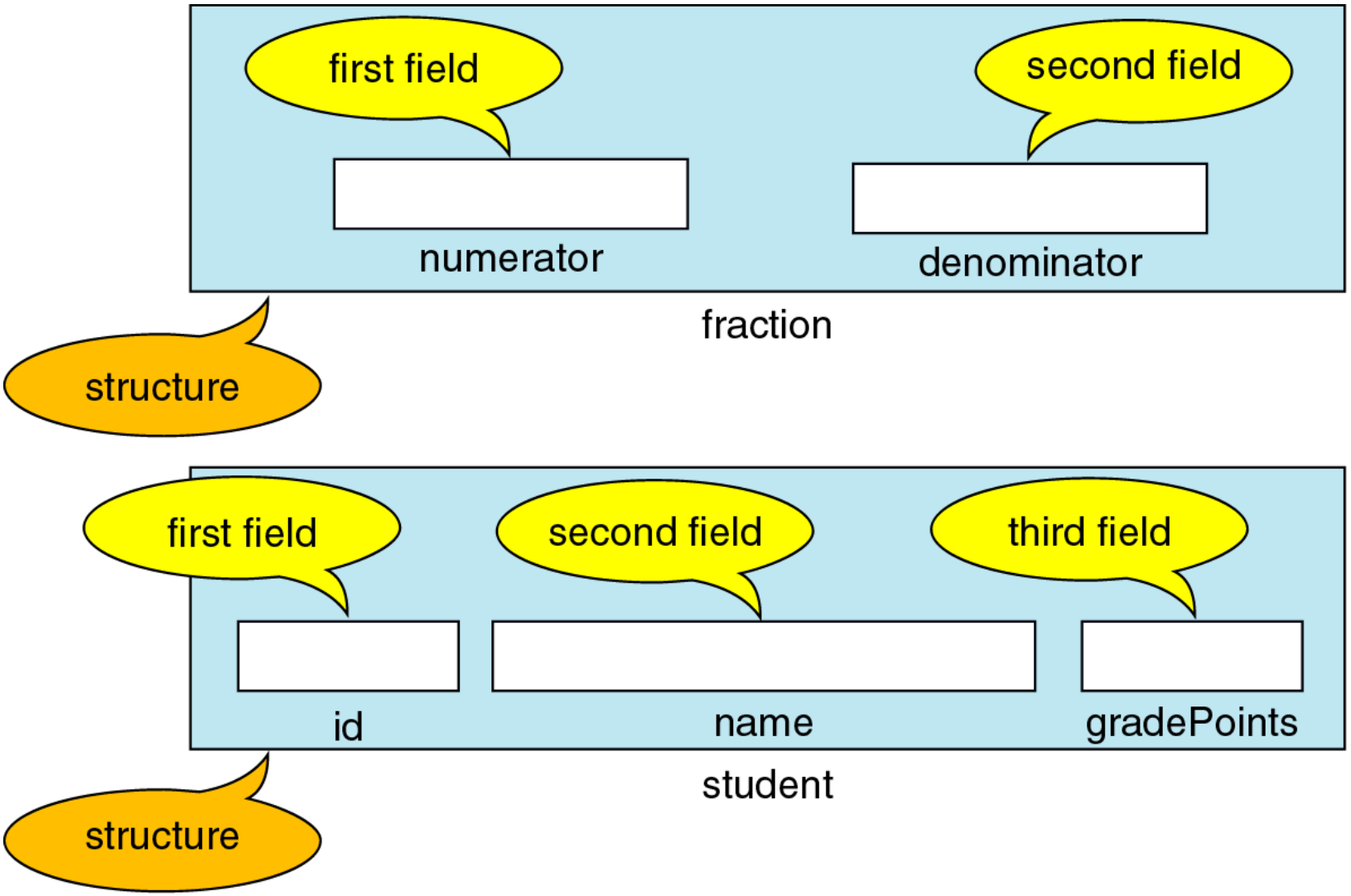
Adicionalmente C ofrece la capacidad de trabajar con campos de bits para reducir la cantidad de memoria utilizada:

- Campos de bits (`struct`).

Estructuras

- Una estructura (`struct`) es una colección de variables relacionadas (campos), que pueden ser de diferentes tipos de datos, bajo un nombre único.
- Ej.: *estudiante* agrupa una serie de datos como son: dni, nombre, dirección, teléfono, sexo, asignaturas matriculadas, notas, etc.
- Normalmente se utilizan para definir *registros* para ser almacenados en ficheros.
- Combinadas con punteros, se pueden crear listas enlazadas, pilas, colas, árboles, etc.

Estructuras



Estructuras – declaración y definición

- Sintaxis de declaración de una estructura:

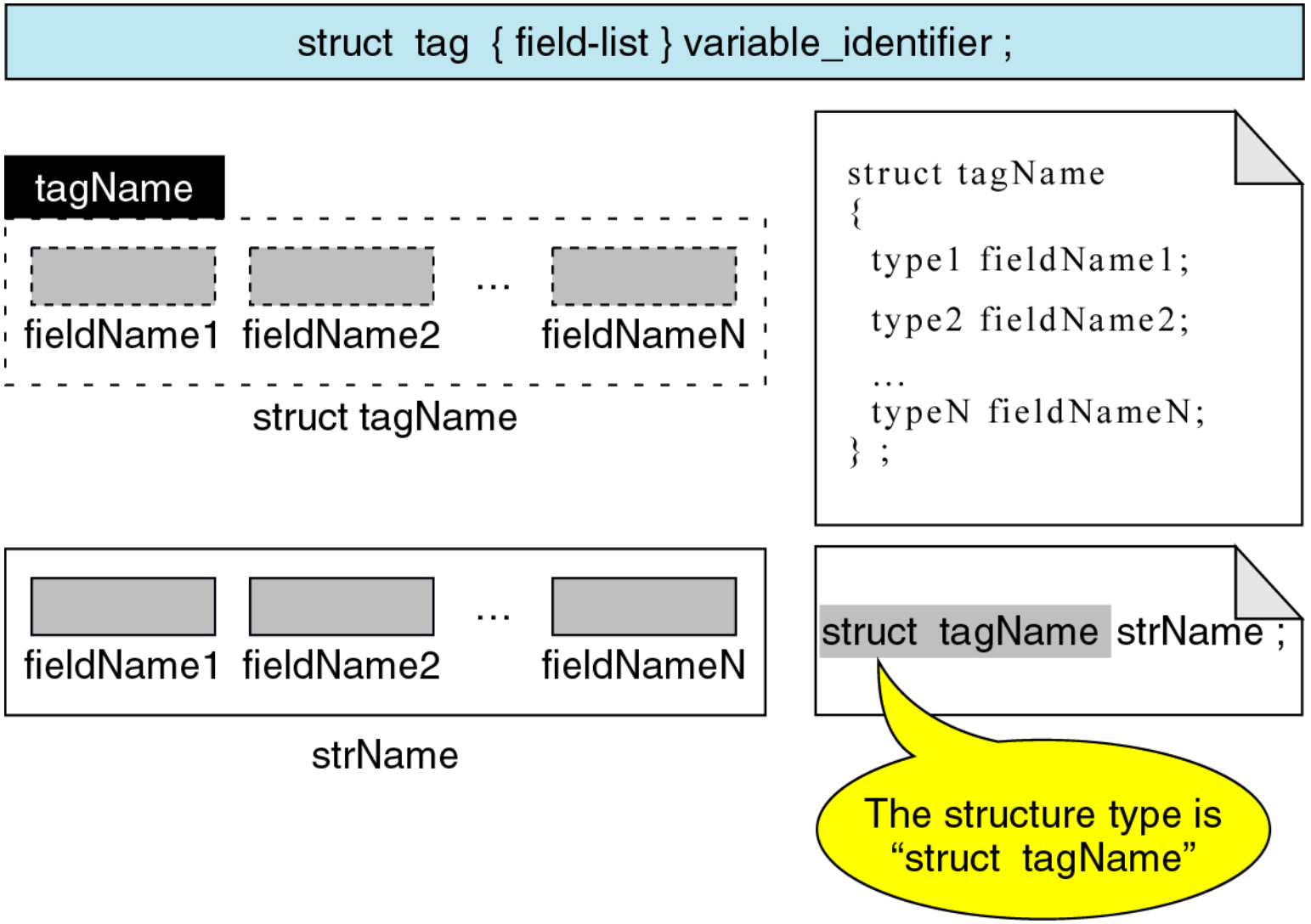
```
struct nombre_de_estructura
{
    tipo_1 campo_1;
    tipo_2 campo_2;
    ...
    tipo_n campo_n;
};
```

La declaración de las estructuras se colocan antes de la función donde se utilizan.

- Con la declaración no se reserva espacio en la memoria, sólo se crea una plantilla con el formato de la estructura. La sintaxis para definir una variable según la declaración anterior es:

```
struct nombre_de_estructura var1, var2,.,varn;
```

Estructuras – declaración y definición



Estructuras – declaración y definición

- Ejemplo de declaración de una estructura:

```
struct paciente
{
    char dni[10];
    char nombre[100];
    char sexo;
    float peso;
    unsigned long n_seg_social;
    char historial[1000];
};
```

- Variables definidas según la estructura anterior:

```
struct paciente pepito, pacientes[200];
```

Estructuras – declaración y definición

- Sintaxis de definición de una variable en base a una estructura sin nombre :

```
struct {
    tipo_1 campo_1;
    tipo_2 campo_2;
    ...
    tipo_n campo_n;
} var1, var2,.,varn;
```

Ej.:

```
struct {
    float real;
    float imag;
} complejo1;
```

typedef

- `typedef` permite dar un nombre arbitrario a un tipo de datos de C. Se usan para definir o “crear” nuevos tipos en base a tipos ya definidos.
- La sintaxis de typedef es parecida a la declaración de una variable excepto que se antepone la palabra reservada typedef.
- Semánticamente el nombre utilizado se convierte en un sinónimo para el tipo de dato que se puede utilizar para definir variables.
- Sintaxis:

```
typedef tipo_existente  tipo_nuevo
```

typedef

```
typedef int BOOLEAN;
typedef int ENTERO;
typedef long int ENTERO_LARGO;
typedef float REAL;
…
/* variables definidas por los nuevos
  tipos*/
ENTERO metros, centimetros;
ENTERO_LARGO distancia;
REAL precision, temperatura;
BOOLEAN b1, b2;
```

Deben definirse antes de declarar cualquier variable del nuevo tipo definido

typedef con struct

Forma 1:

```
struct estudiante {
        char *nombre;
        unsigned short int edad;
        int nota;
};
typedef struct estudiante ESTUDIANTE;
…
ESTUDIANTE curso_prog[NUM_ESTUD];
```

typedef con struct

Forma 2:

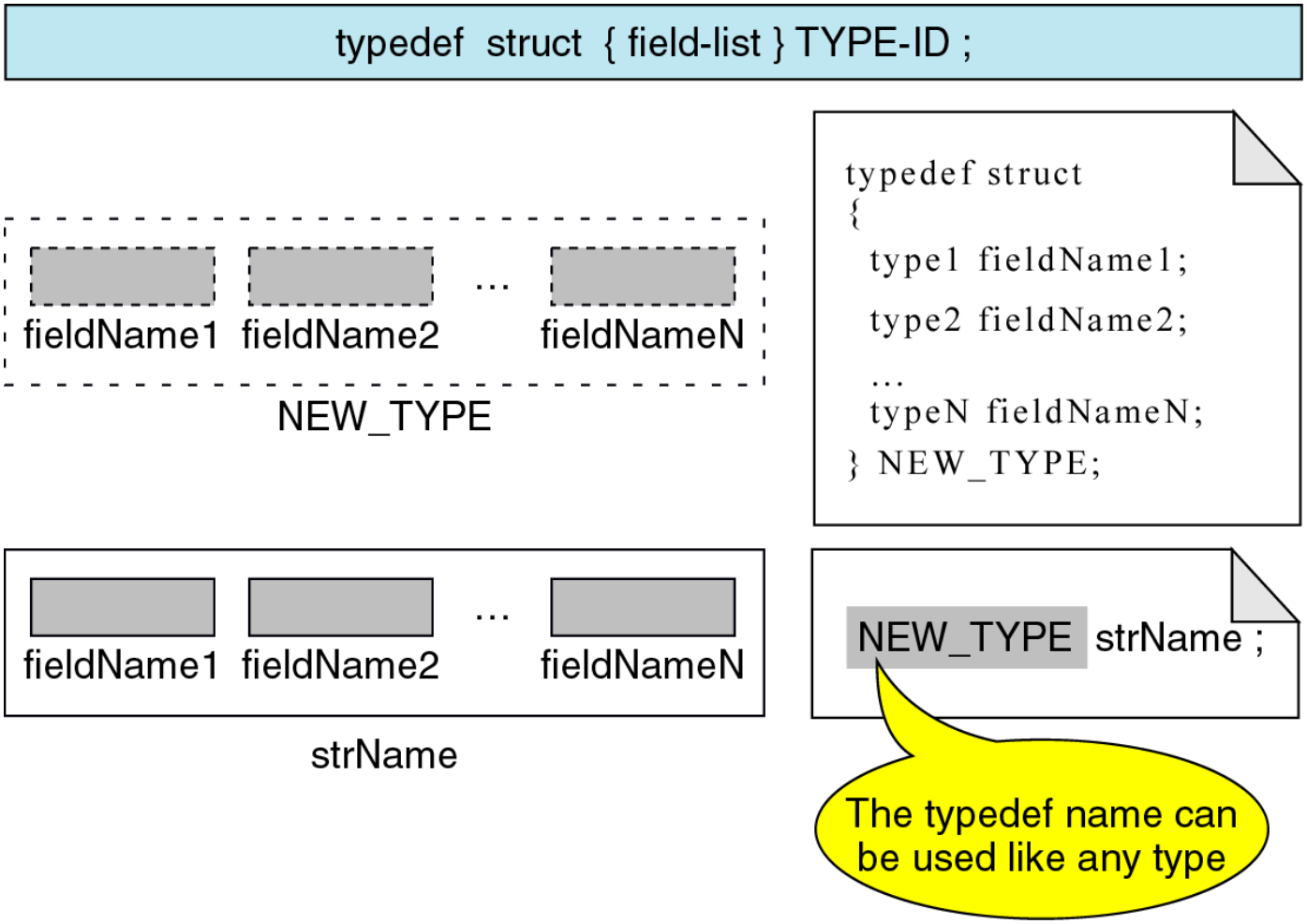
```
typedef struct estudiante {
        char *nombre;
        unsigned short int edad;
        int nota;
} ESTUDIANTE;

…

ESTUDIANTE curso_prog[NUM_ESTUD];
```

typedef con struct

Forma 2:



Resumen de declaraciones struct

```
struct    {
             …
          }  variable_identifier ;
```

structure variable

```
struct  tag
          {
             …
          }  variable_identifier ;

struct  tag  variable_identifier ;
```

tagged structure

```
typedef  struct
          {
             …
          }  TYPE_ID  ;

TYPE_ID   variable_identifier ;
```

type-defined structure

Definición e inicialización de estructuras

- Se puede definir e inicializar una variable declarada como una estructura.

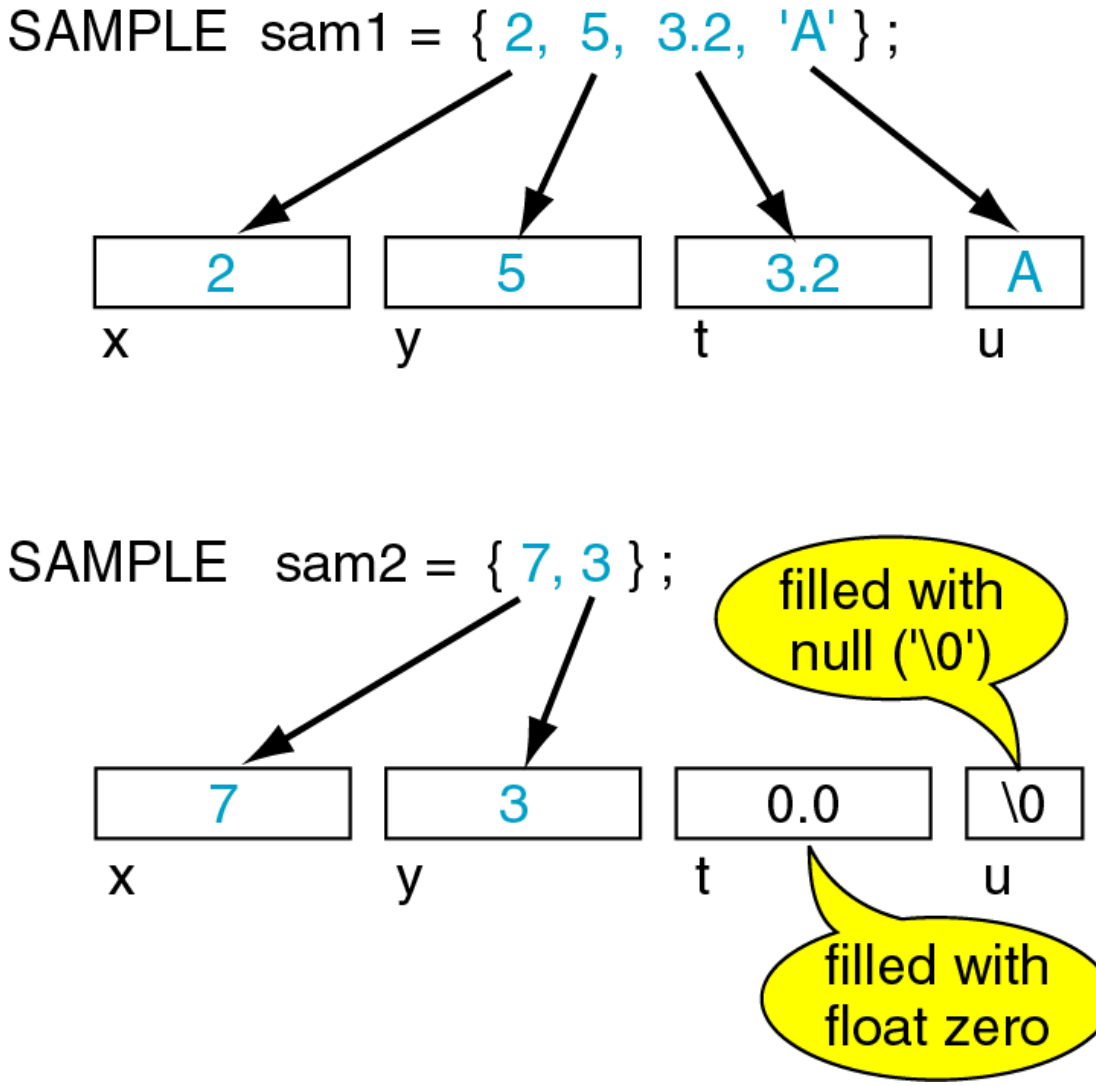
Ej.:

```
struct persona
{
  char nombre[20];
  int  edad;
  float peso;
};

struct persona pepe={“Adan Perez”,31,80};
```

Definición e inicialización de estructuras

```
typedef struct
    {
    int   x;
    int   y;
    float t;
    char  u;
  } SAMPLE;
```



Acceso a los campos de estructuras

- Para acceder a los miembros de una estructura se usa el *operador punto*.
- Sintaxis:

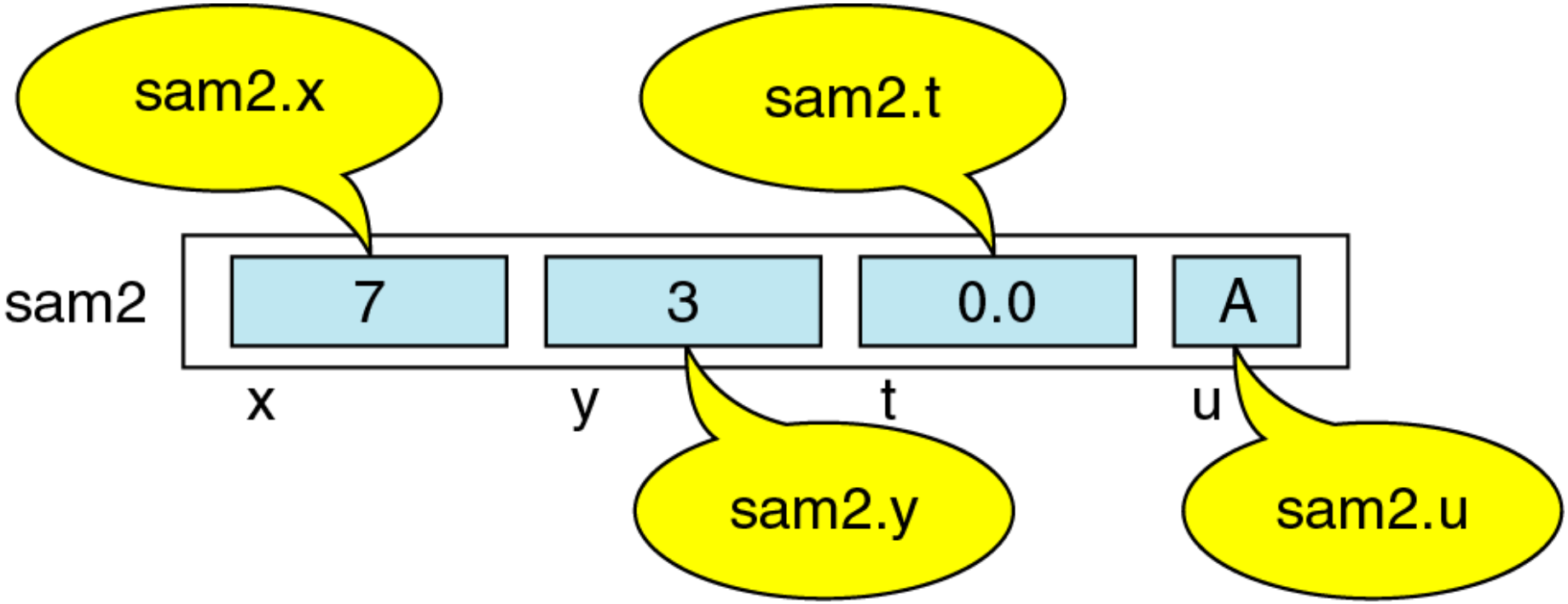
```
nombre_de_estructura.nombre_del_campo
```

Ej.:

```
struct persona
{
   char nombre[20];
   int  edad;
   float peso;
};
…
struct persona pepe;
```

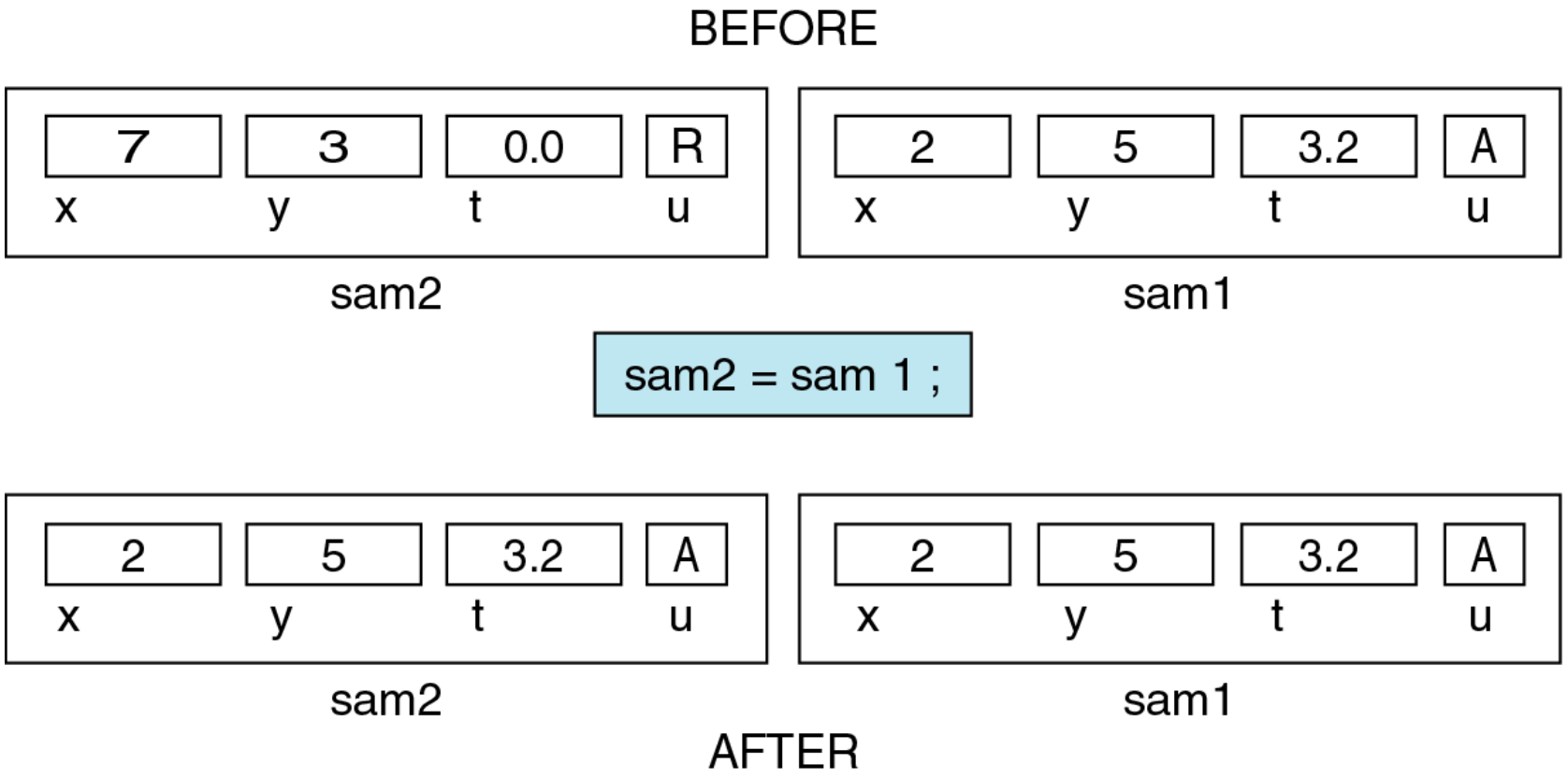
```
…
strcpy(pepe.nombre,"Jon Pi");
pepe.edad = 27;
pepe.peso = 67.5;
```

Acceso a los campos de estructuras



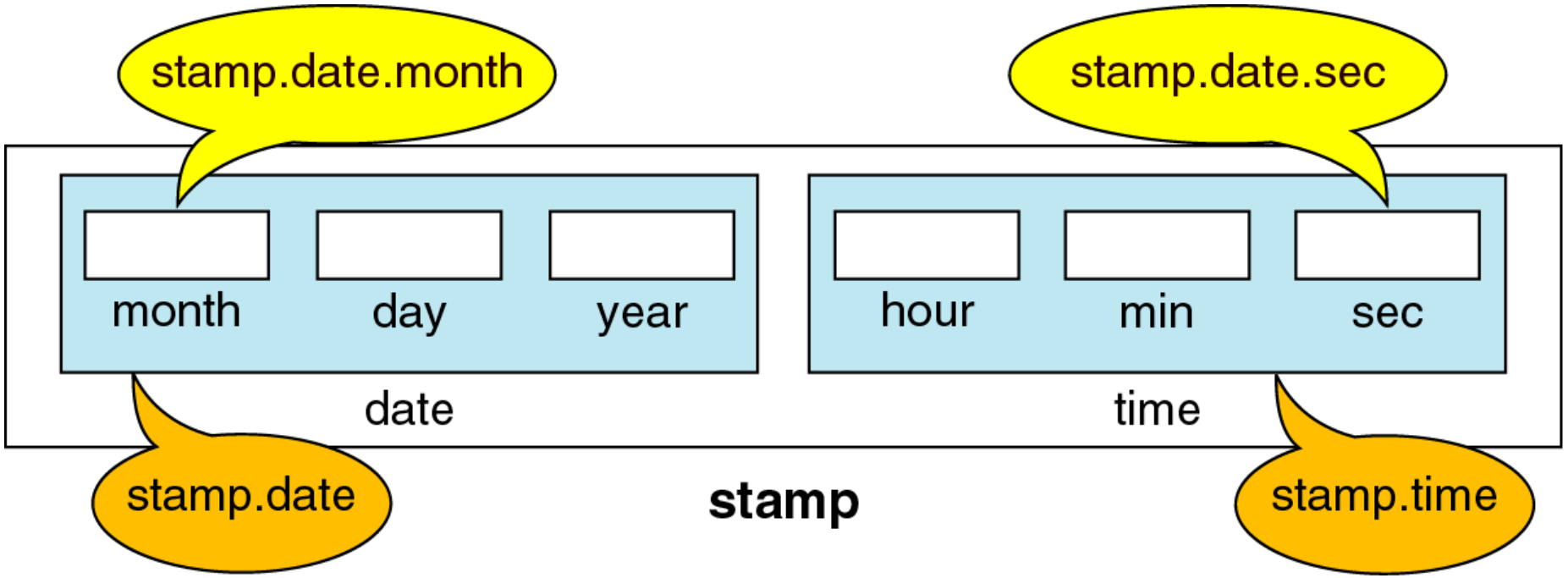
Asignación entre estructuras

- Se puede asignar variables de la misma estructura.

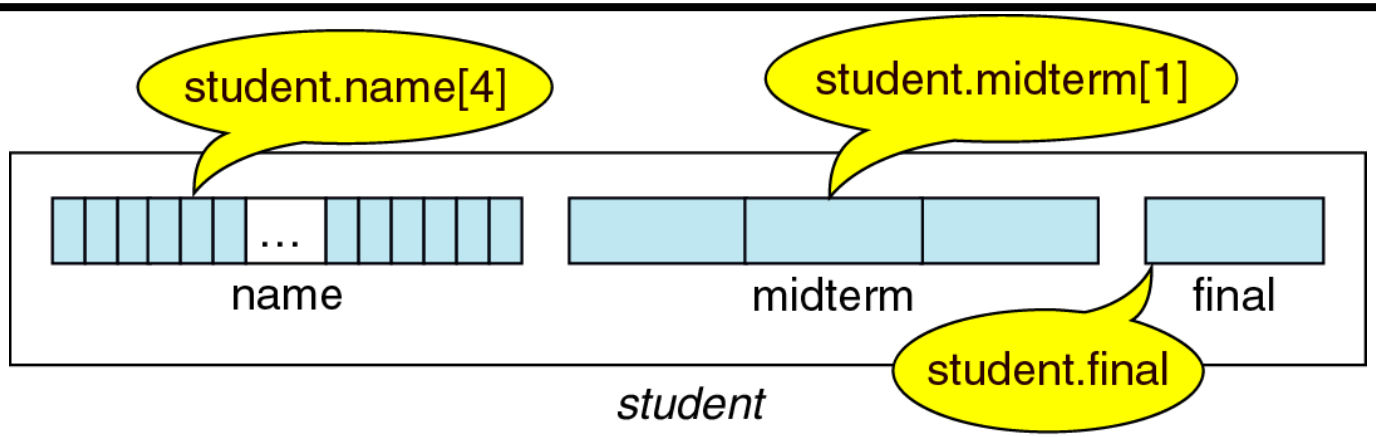


Estructuras como campos de estructuras

- Se puede tener una estructura como campo de otra estructura.



Arrays como campos de estructuras




```
/* Global Declarations */
typedef  struct
  {
   char name[26];
   int  midterm[3];
   int  final int;
  } STUDENT ;
/* Local Definitions */
STUDENT student;
```

Ejemplo: arrays y estructuras como campos

```
typedef struct estudiante {
    /* campos del estudiante: dni, nombre, edad, etc. */
} ESTUDIANTE;

typedef struct universidad {
    char *nombre;
    double puntaje;
    ESTUDIANTE estud[NUM_ESTUD];
} UNIVERSIDAD;


int main()
{
  UNIVERSIDAD Cantabria;

  Cantabria.puntaje = 9.3;
  Cantabria.estud[2541].edad = 20;
}
```

Ejemplo: suma complejos

```
/*********************************************************************\
* Programa: suma_complejo1.c                                          *
* Descripción: Prog. que usa estruct. para representar y sumar complejos *
\*********************************************************************/
#include <stdio.h>

typedef struct complejo {
  double real;
  double imag;
} COMPLEJO;

int main()
{
 COMPLEJO comp1, comp2, result;

 printf("Introduce la parte real del primer complejo: "); scanf(" %lf", &comp1.real);
 printf("Introduce la parte imaginaria del primer complejo: ");
 scanf(" %lf", &comp1.imag);

 printf("Introduce la parte real del segundo complejo: "); scanf(" %lf", &comp2.real);
 printf("Introduce la parte imaginaria del segundo complejo: ");
 scanf(" %lf", &comp2.imag);

 result.real = comp1.real + comp2.real; result.imag = comp1.imag + comp2.imag;
 printf("\nEl resultado de la suma es: %f + %f j\n",result.real,result.imag);

 return 0;
}
```

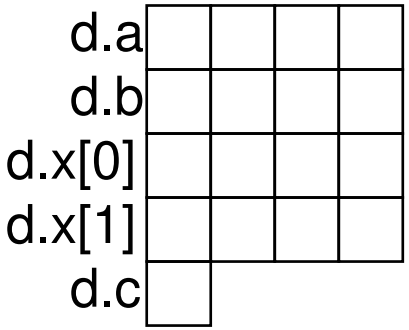
Uniones

- Una union (`union`) es similar a una estructura en que contiene miembros que pueden ser de diferentes tipos de datos, pero que *comparten* el mismo área de almacenamiento en memoria. En las estructuras cada miembro tiene asignada su propia área de almacenamiento.
- Las uniones se usan para ahorrar memoria y son útiles en aplicaciones que involucran múltiples miembros donde no se necesita asignar valores a todos los miembros a la vez o hay condiciones entre los campos.

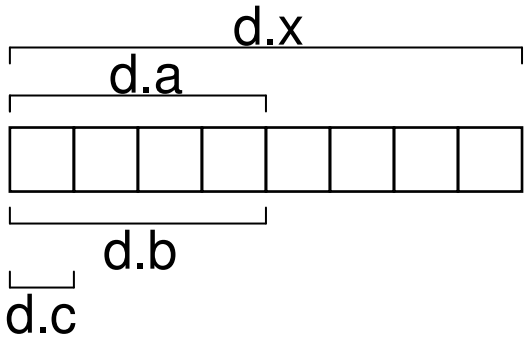
Union

Una `union` es similar a un `struct`, pero todos los campos comparten la misma memoria.

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




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



Uniones – declaración y definición

- Sintaxis de declaración de una union:

```
union [nombre_de_union]
{
    tipo_1 campo_1;
    tipo_2 campo_2;
    ...
    tipo_n campo_n;
};
```

La declaración de las uniones se colocan antes del elemento que lo utiliza.

- Con la declaración no se reserva espacio en la memoria, sólo se crea una plantilla con el formato de la unión. La sintaxis para definir una variable según la declaración anterior es:

```
union nombre_de_union var1, var2,.,varn;
```

Uso de `union`

Los `union` se usan para diferentes representaciones de los datos o para información condicionada:

```
union id
{
  int  talla;
  char color[12];
} camisa;
```

```
struct empleado
{
  char nombre[40];
  int  tipo_contrato;
  union
  {
    int  nomina;
    int  pts_hora;
  } sueldo;
} p;
```

Acceso a los campos de uniones

- Para acceder a los miembros de una union se usa el *operador punto*.
- Sintaxis:

```
nombre_de_union.nombre_del_campo
```

Ej.:

```
union id
{
   char color;
   int  talla;
};
…
union id pepe;
```

```
…
pepe.color = 'R';
…
pepe.talla = 67;
```

Ejemplo: manejo de union

```
/*****************************************************************\
* Programa: ej_union.c                                            *
* Descripción: Prog. que usa union                                *
\*****************************************************************/
#include <stdio.h>

int main()
{
  union id {
    char color[12];
    int talla;
  };
  struct ropa {
    char fabricante[20];
    float coste;
    union id descripcion;
  };

  struct ropa camisa = {"Americana", 25.0, "blanca"};
  printf(" %d\n", sizeof(union id));
  printf("%s %5.2f  ", camisa.fabricante, camisa.coste);
  printf("%s %d \n", camisa.descripcion.color, camisa.descripcion.talla);
  camisa.descripcion.talla = 12;
  printf("%s %5.2f  ", camisa.fabricante, camisa.coste);
  printf("%s %d \n", camisa.descripcion.color, camisa.descripcion.talla);

 return 0;
}
```

Campos de bits

- Cuando se requiere trabajar con elementos que consten de pocos bits se puede usar los campos de bits.
- Varios datos de este tipo se puede empaquetar en una sola palabra de memoria.
- La palabra se subdivide en capos de bits individuales.
- Los campos de bits se definen como miembros de una estructura.
- Los miembros de los campos de bits se acceden como cualquier miembro de una estructura.

Campos de bits – declaración y definición

- Sintaxis de declaración de un campo de bits:

```
struct [nombre_de_cbits]
{
    tipo_1 campo_1 : expr-cte;
    tipo_2 campo_2;
    ...
    tipo_n campo_n;
};
```

- Los tipos aceptables son: unsigned int, signed int, or int. La expr-cte debe ser un valor no negativo.
- La sintaxis para definir una variable según la declaración anterior es:

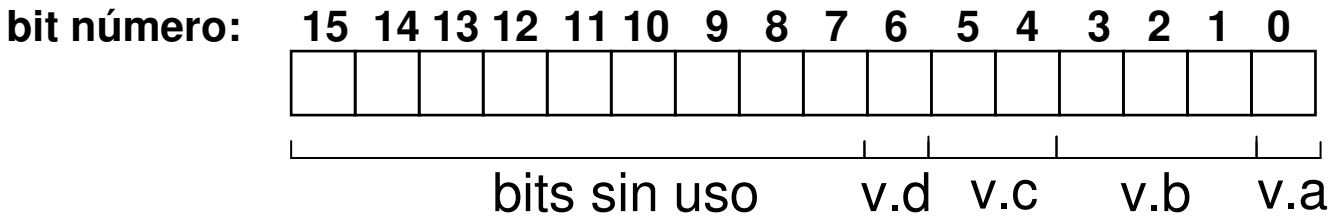
```
union nombre_de_cbits var1, var2,.,varn;
```

Campos de bits

Ejemplo:

```
struct muestra
{
  unsigned short a : 1;
  unsigned short b : 3;
  unsigned short c : 2;
  unsigned short d : 1;

};
struct muestra v;
```



Ejemplo: manejo de campos de bits

```
/*****************************************************************\
* Programa: ej_cbits.c                                            *
* Descripción: Prog. que usa un campo de bits                     *
* Autor: Pedro Corcuera                                           *
* Revisión: 1.0 2/02/2008                                         *
\*****************************************************************/
#include <stdio.h>

int main()
{
  struct cbits {
     unsigned short a : 5;
     unsigned short b : 3;
     unsigned short c : 1;
     unsigned short d : 4;
  };

  struct cbits v = {24, 5, 1, 14};

  printf("v.a = %d   v.b = %d   v.c = %d   v.d = %d\n", v.a, v.b, v.c, v.d);
  printf("v requiere %d bytes \n", sizeof(v));

 return 0;
}
```