Estructura de Datos

MEMORIA ESTÁTICA
MEMORIA DINÁMICA
TIPO PUNTERO
DECLARACIÓN DE PUNTEROS
GESTIÓN DE MEMORIA DINÁMICA
RESUMEN
EJEMPLO

Datos estáticos

Su tamaño y forma es constante durante la ejecución de un programa y por tanto se determinan en tiempo de compilación.

El ejemplo típico son los arrays.

Tienen el problema de que hay que dimensionar la estructura de antemano, lo que puede conllevar desperdicio o falta de memoria.

Datos dinámicos

Su tamaño y forma es variable (o puede serlo) a lo largo de un programa, por lo que se crean y destruyen en tiempo de ejecución.

Esto permite dimensionar la estructura de datos de una forma precisa: se va asignando memoria en tiempo de ejecución según se va necesitando.

Tipo puntero

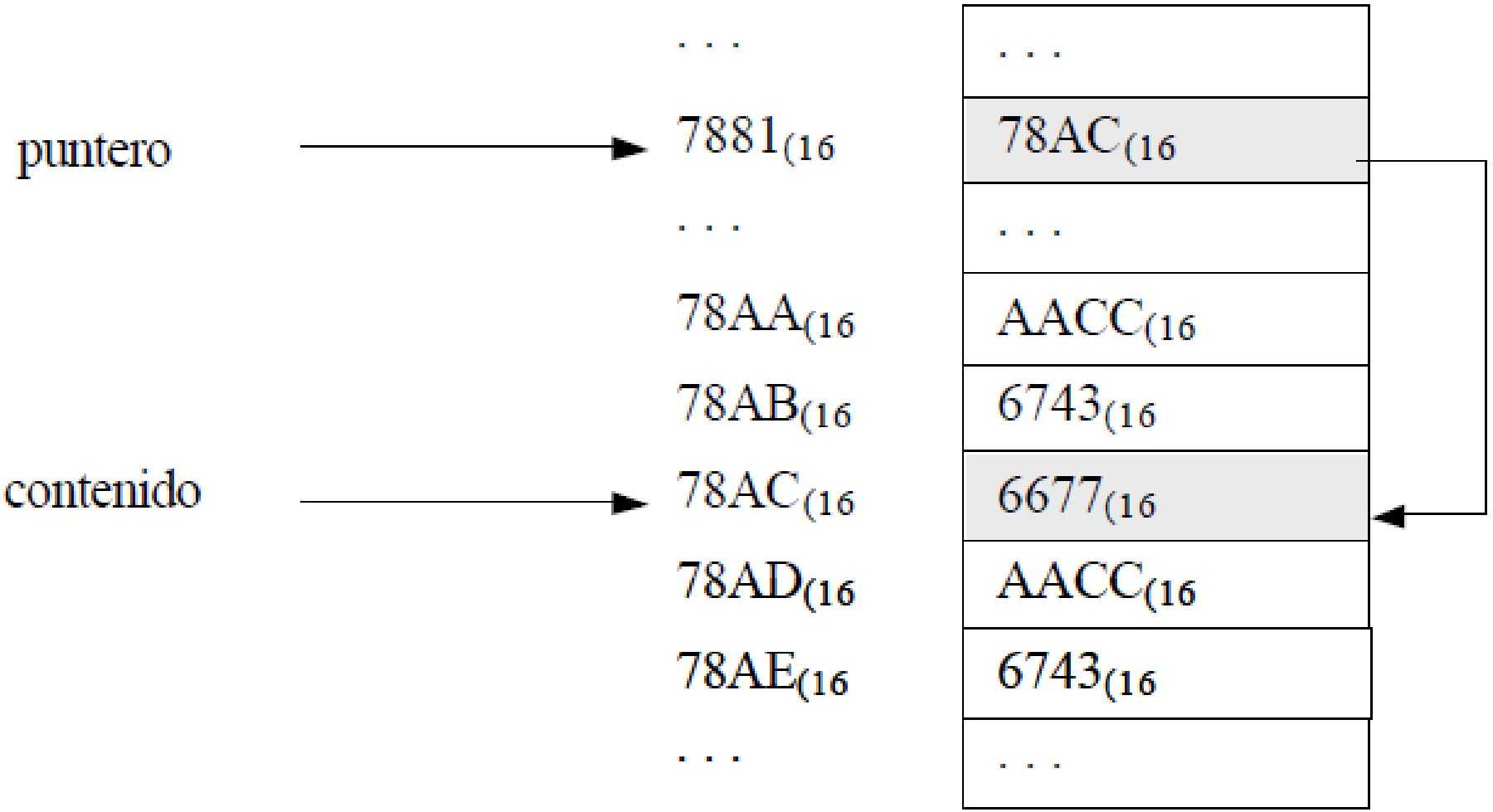
Las variables de tipo puntero son las que nos permiten referenciar datos dinámicos.

Tenemos que diferenciar claramente entre:

1. La variable referencia o apuntadora, de tipo puntero.
2. La variable anónima referenciada o apuntada, de cualquier tipo, tipo que estará asociado siempre al puntero.

Físicamente, un puntero no es mas que una dirección de memoria.

En el siguiente ejemplo se muestra el contenido de la memoria con un puntero que apunta a la dirección $78AC_{(16}$, la cual contiene $6677_{(16}$:



Declaración de Punteros

Definiremos un tipo puntero con el carácter asterisco (*) y especificando siempre el tipo de la variable referenciada.

Ejemplo:

```
Typedef   int   *PtrInt;          // puntero a enteros

PtrInt p;          // puntero a enteros
```

O bien directamente:

```
int *p;          // puntero a enteros
```

Cuando **p** este apuntando a un entero de valor -13, gráficamente lo representaremos así:

Para acceder a la variable apuntada hay que hacerlo a través de la variable puntero, ya que aquella no tiene nombre (por eso es anónima).

La forma de denotarla es *p.

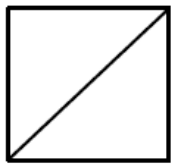
En el ejemplo *p = -13
p = dirección de memoria de la celda con el valor -13, dirección que no necesitamos tratar directamente.

Ejemplo de punteros a estructuras

```
struct TipoRegistro {
    int num;
    char car;
 };

 typedef TipoRegistro
*TipoPuntero;
 TipoPuntero p;
```

Para inicializar una variable de tipo puntero, le asignaremos la macro NULL (p = NULL)



Así:

p es la dirección de un registro con dos campos (tipo puntero)

***p** es un registro con dos campos (tipo registro)

(*p).num es una variable simple (tipo entero)

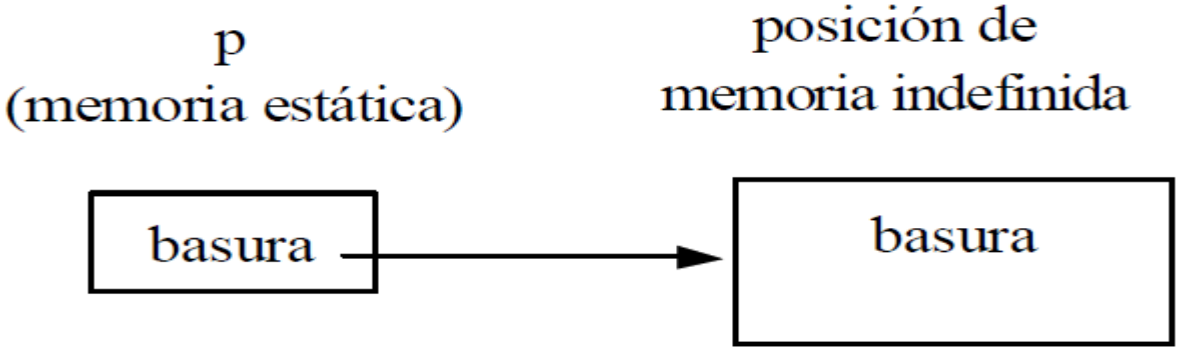
p->num es una variable simple (tipo entero)

(*p).car es una variable simple (tipo carácter)

p->car es una variable simple (tipo carácter)

Gestión de la memoria dinámica

Cuando declaramos una variable de tipo puntero, por ejemplo **int *p**; estamos creando la variable p, y se le reservara memoria -estatica- en tiempo de compilación; pero la variable referenciada o anónima no se crea. En este momento tenemos:



La variable anónima debemos crearla después mediante una llamada a un procedimiento de asignación de memoria -dinámica- predefinido.

¿Cómo se reserva memoria dinámicamente?

El lenguaje C dispone de una serie de librerías de funciones estándar. El fichero de cabeceras stdlib.h contiene las declaraciones de dos funciones que nos permiten reservar memoria, así como otra función que nos permite liberarla.

Reserva de memoria Las dos funciones que nos permiten reservar memoria son:

malloc (cantidad_de_memoria);

calloc (número_de_elementos, tamaño_de_cada_elemento);

Estas dos funciones reservan la memoria especificada y nos devuelven un puntero a la zona en cuestión. Si no se ha podido reservar el tamaño de la memoria especificado devuelve un puntero con el valor 0 o NULL.
Antes de indicar como deben utilizarse las susodichas funciones tenemos que comentar el operador sizeof. Este operador es imprescindible a la hora de realizar programas portables, es decir, programas que puedan ejecutarse en cualquier máquina que disponga de un compilador de C.

El operador sizeof(tipo_de_dato), nos devuelve el tamaño que ocupa en memoria un cierto tipo de dato, de esta manera, podemos escribir programas independientes del tamaño de los datos y de la longitud de palabra de la máquina. En resumen si no utilizamos este operador en conjunción con las conversiones de tipo cast probablemente nuestro programa sólo funciones en el ordenador sobre el que lo hemos programado.

Con todo lo mencionado anteriormente veamos un ejemplo de un programa que reserva dinámicamente memoria para algún dato.

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

main()
{
int     *p_int;
float   *mat;

p_int = (int *)  malloc(sizeof(int));
mat   = (float *)calloc(20,sizeof(float));

if ((p_int==NULL)||(mat==NULL))
        {
                printf ("\nNo hay memoria");
                exit(1);
        }

/* Aquí irían las operaciones sobre los datos */
/* Aquí iría el código que libera la memoria */

}
```

Este programa declara dos variables que son punteros a un entero y a un float. A estos punteros se le asigna una zona de memoria, para el primero se reserva memoria para almacenar una variable entera y en el segundo se crea una matriz de veinte elementos cada uno de ellos un float. Obsérvese el uso de los operadores cast para modificar el tipo del puntero devuelto por malloc y calloc, así como la utilización del operador sizeof.

Para termina un breve comentario sobre las funciones anteriormente descritas. Básicamente da lo mismo utilizar malloc y calloc para reservar memoria es equivalente:

```
mat = (float *)calloc (20,sizeof(float));
mat = (float *)malloc (20*sizeof(float));
```

Liberación de la memoria. La función que nos permite liberar la memoria asignada con malloc y calloc es free(puntero), donde puntero es el puntero devuelto por malloc o calloc. En nuestro ejemplo anterior, podemos ahora escribir el código etiquetado como : /* Ahora iría el código que libera la memoria */ free (p_int); free(mat);

Ventajas de la asignación dinámica

Vamos a exponer un ejemplos en el que se aprecie claramente la utilidad de la asignación dinámica de memoria. Supongamos que deseamos programar una serie de funciones para trabajar con matrices. Una primera solución sería definir una estructura de datos matriz, compuesta por una matriz y sus dimensiones puesto que nos interesa que nuestras funciones trabajen con matrices de cualquier tamaño. Por tanto la matriz dentro de la estructura tendrá el tamaño máximo de todas las matrices con las que queramos trabajar y como tenemos almacenadas las dimensiones trabajaremos con una matriz de cualquier tamaño pero tendremos reservada memoria para una matriz de tamaño máximo.

Estamos desperdiciando memoria. Una definición de este tipo sería:

```
typedef struct {
    float mat[1000][1000];
    int ancho,alto;
 } MATRIZ;
```

Sin embargo podemos asignar memoria dinámicamente a la matriz y reservar sólo el tamaño que nos hace falta. La estructura sería ésta.

```
struct mat {
float *datos;
int ancho,alto; };

typedef struct mat *MATRIZ;
```

El tipo MATRIZ ahora debe ser un puntero puesto que tenemos que reservar memoria para la estructura que contiene la matriz en sí y las dimensiones. Una función que nos crease una matriz sería algo así:

```
MATRIZ inic_matriz (int x,int y)
 {
MATRIZ temp;
 temp = (MATRIZ) malloc (sizeof(struct mat)); t
emp->ancho = x;
temp->alto = y;
 temp->datos = (float *) malloc (sizeof(float)*x*y);
return temp;
}
```

En Resumen

Un apuntador es una ***variable*** que contiene la dirección en memoria de otra variable.

Se pueden tener apuntadores a cualquier tipo de variable.

El operador ***unario*** o ***monádico*** **&** devuelve la dirección de memoria de una variable.

El operador de ***indirección*** o ***dereferencia*** * devuelve el ``contenido de un objeto apuntado por un apuntador''.

Para declarar un apuntador para una variable entera hacer:

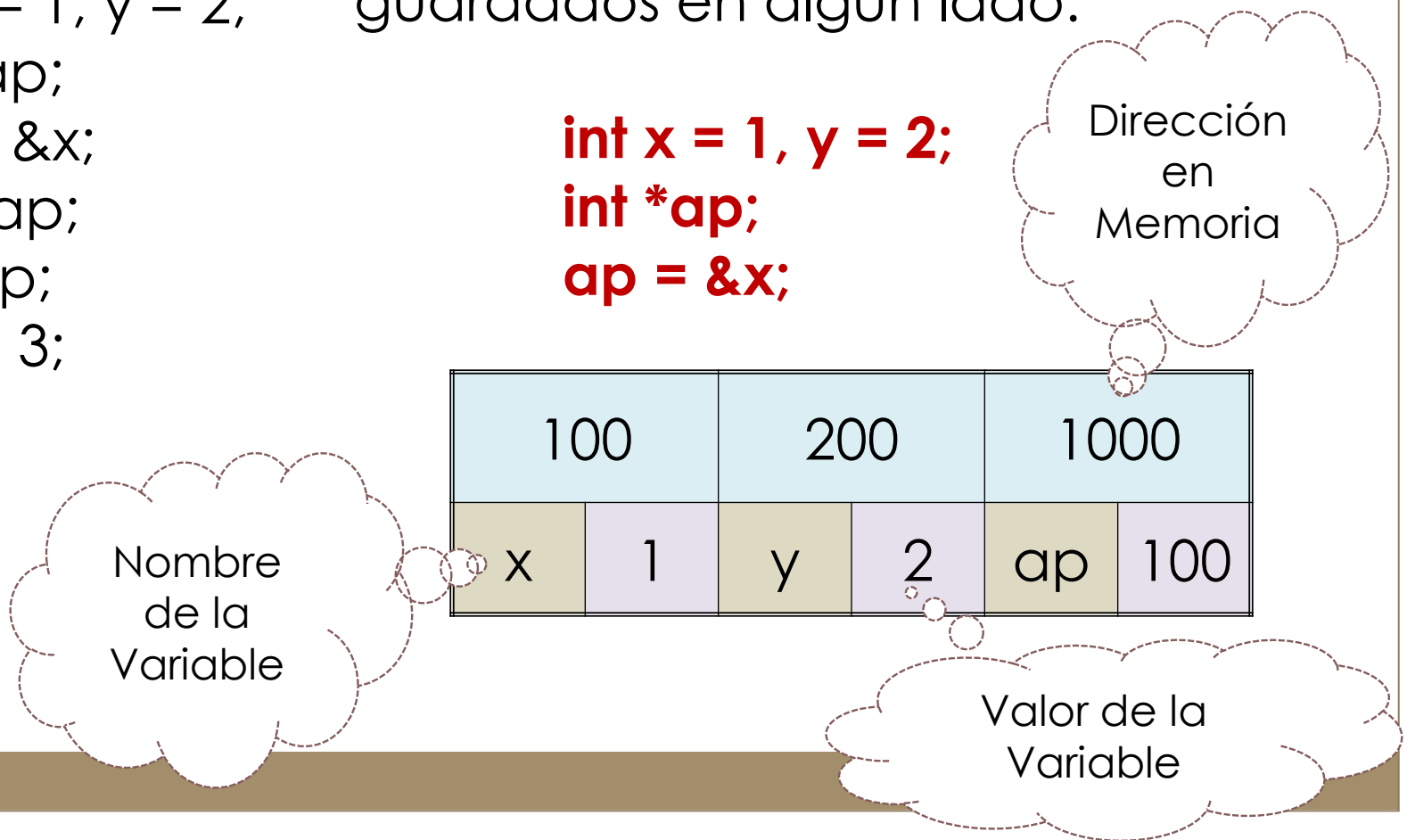
```
int *apuntador;
```

Ejemplo

```
main() {
    int x = 1, y = 2;
    int *ap;
    ap = &x;
    y = *ap;
    x = ap;
   *ap = 3;
}
```

Nota: un apuntador es una variable, por lo tanto, sus valores necesitan ser guardados en algún lado.

int x = 1, y = 2;
int *ap;
ap = &x;



Las variables ***x*** e ***y*** son declaradas e inicializadas con ***1*** y ***2*** respectivamente, ***ap*** es declarado como un apuntador a entero y se le asigna la dirección de ***x*** (***&x***). Por lo que ***ap*** se carga con el valor ***100***.

y = *ap;

100		200		1000	
x	1	y	1	ap	100

Después ***y*** obtiene el contenido de ***ap***. En el ejemplo ***ap*** apunta a la localidad de memoria 100 -- la localidad de ***x***. Por lo tanto, ***y*** obtiene el valor de ***x*** -- el cual es 1.

x = ap;

100		200		1000	
x	100	y	1	ap	100

El valor de ap en ese momento es 100, entonces asigna a x el valor de ap

***ap= 3;**

100		200		1000	
x	3	y	1	ap	100

Finalmente se asigna un valor al contenido de un apuntador (****ap***).