

Punteros. Introducción



Eva Lucrecia Gibaja Galindo

Dpto. Informática y Análisis Numérico

Introducción

- Cuando una variable se declara, se asocian 3 atributos con la misma:

- Nombre
- Tipo
- Dirección

```
int numero=6;
```

1011		numero
1007	6	
1003		
1002		
1001		

- Al valor de la variable se accede por su nombre, ¿Cómo se accede a su dirección?

Tipo de dato puntero

- Un puntero es un tipo de dato que contiene la dirección de memoria de un dato

■ Declaración

Tipo de variable * Nombre de variable

- Tipo de variable: Cualquier tipo, predefinido o creado
- Cuando se declara un puntero se reserva memoria para albergar la dirección de memoria de un dato, no el dato en si

Punteros. Declaración

■ Ejemplo: `char c='a';`
`char* ptrc;`
`int* ptri;`

1011	...	
1007	?	ptri
1003	?	ptrc
1002	'a'	c
1001	...	

■ Declara:

- *c* como una variable de tipo carácter cuyo valor es 'a'
- *ptri* como una variable de tipo puntero que puede contener direcciones de memoria de objetos de tipo *int*
- *ptrc* como una variable puntero que puede contener direcciones de memoria de objetos de tipo *char*

Operadores

■ Operador de dirección: **&<variable>**

- Devuelve la dirección de memoria donde empieza la variable *<variable>*
- Se utiliza habitualmente para asignar valores a datos de tipo puntero

```
int i, *ptri; ptri=&i;
```

- *i* es una variable de tipo entero, por lo que la expresión *&i* es la dirección de memoria donde comienza un entero y, por tanto, puede ser asignada al puntero *ptri*
- Se dice que *ptri* apunta o referencia a *i*
- Se puede aplicar sobre datos de cualquier tipo (incluso sobre punteros)

Operadores

■ Operador de contenido (indirección): ***<puntero>**

- Devuelve el contenido del objeto referenciado por *<puntero>*
- Esta operación se usa para acceder al objeto referenciado o apuntado por el puntero

```
char c, *ptrc;
```

```
ptrc=&c;
```

```
*ptrc='A' //Equivalente a c='A'
```

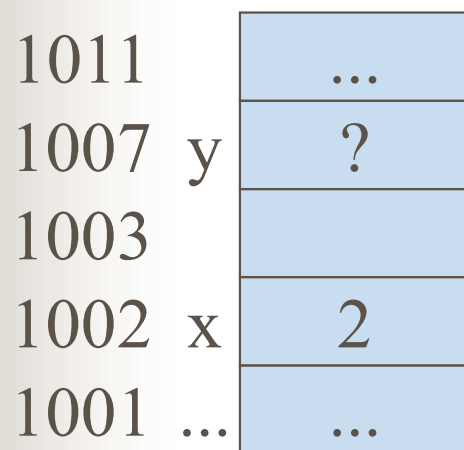
- *ptrc* es un puntero a carácter que contiene la dirección de *c*, por tanto, la expresión **ptrc* es el objeto apuntado por el puntero, es decir, *c*

- Solo se puede aplicar sobre datos de tipo puntero

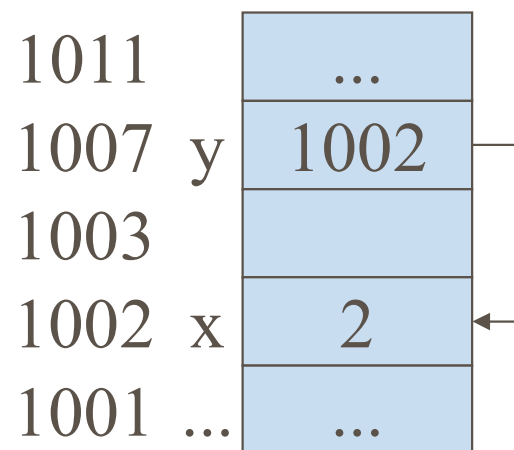
Operadores

■ Ejemplo:

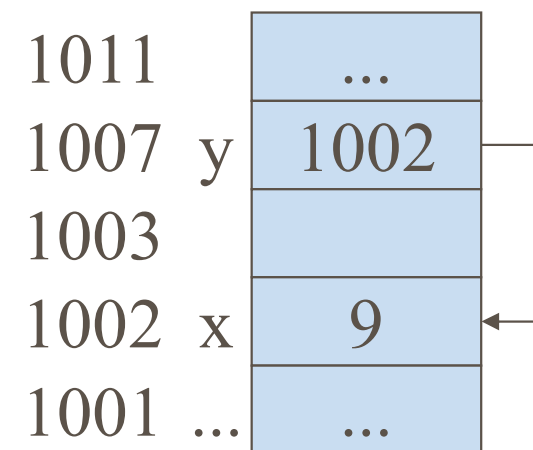
```
[1] int x=2; /* variable de tipo entero*/
[2] int * y; /*variable puntero*/
[3] y=&x; /* y contiene la dirección de x */
[4] x=(*y)+7; /* equivale a x = x+7; */
```



[1] y [2]



[3]



[4]

Ejemplo

```
void main()
```

{

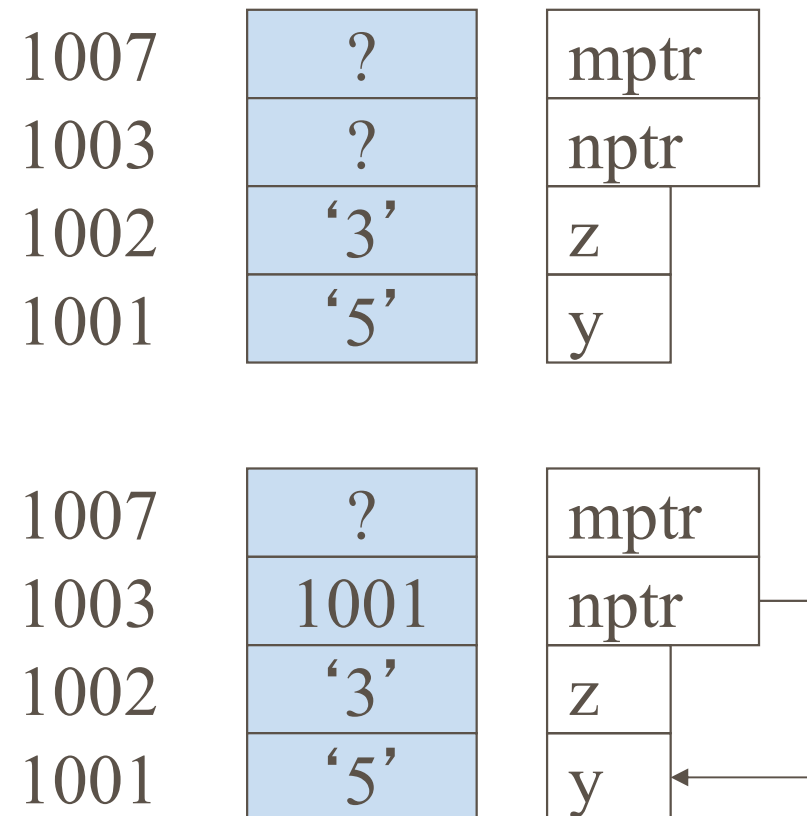
```
char y = '5';
```

```
char z = '3';
```

```
char* nptr;
```

```
char* mptr;
```

```
nptr=&y;
```

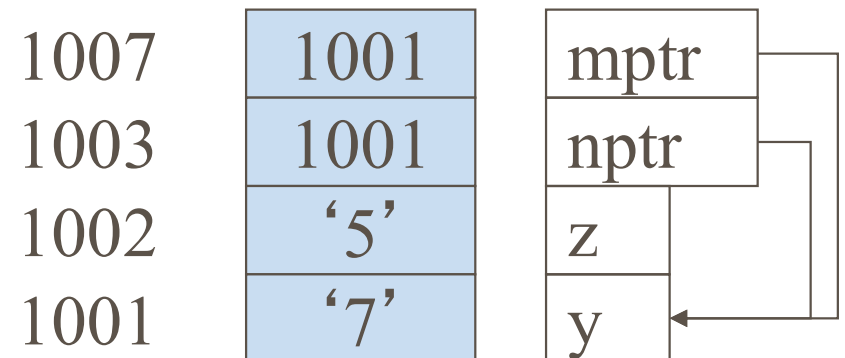
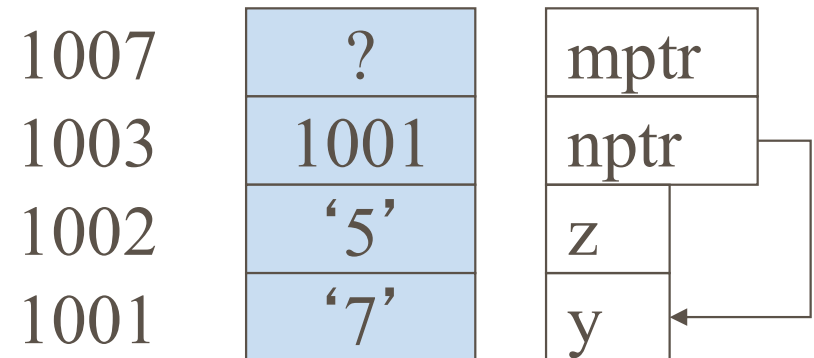
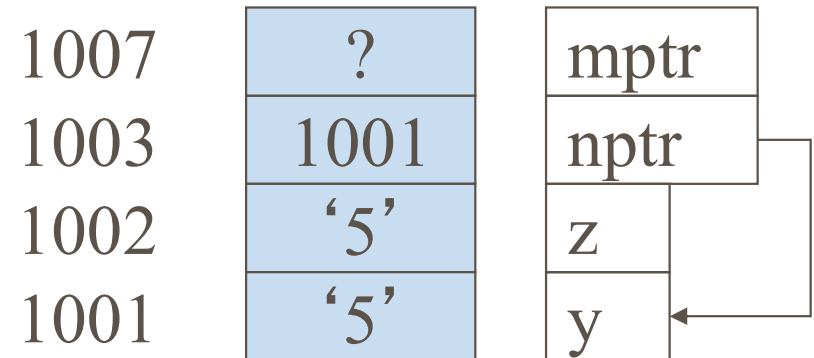


Ejemplo

```
z=*nptr; //z=y;
```

```
*nptr='7'; //y='7';
```

```
mptr = nptr;
```



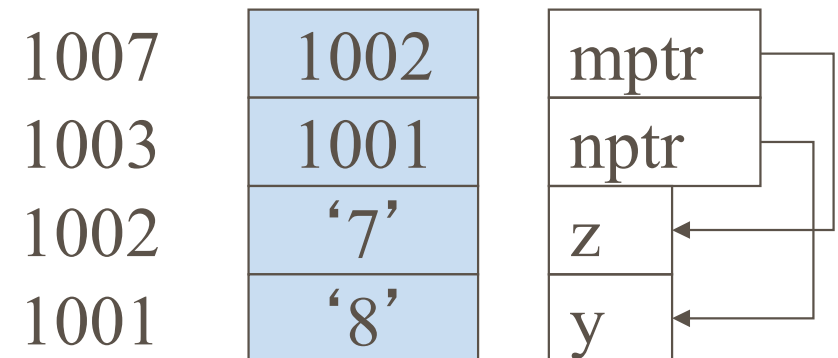
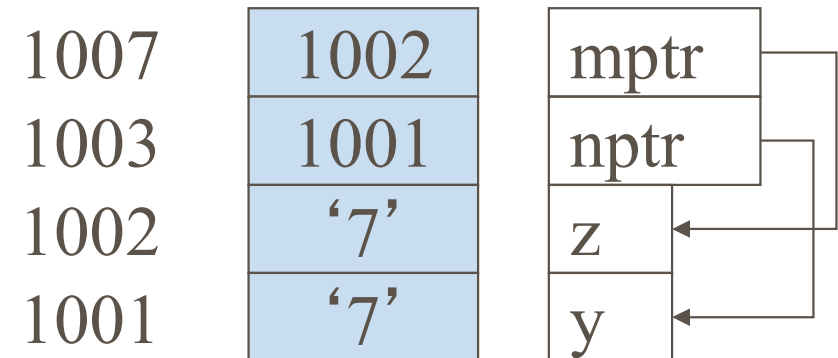
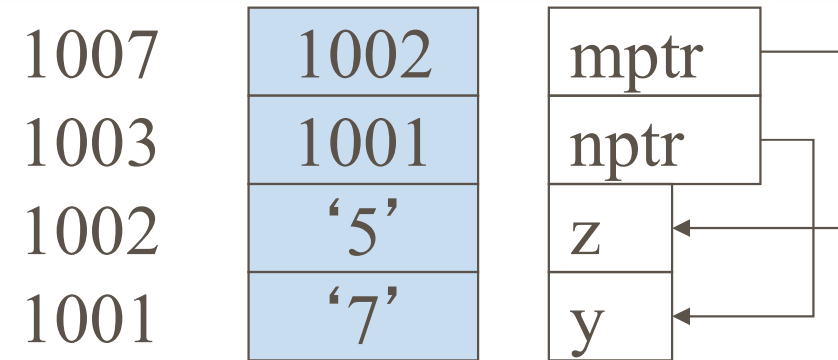
Ejemplo

```
mpt r=&z;
```

```
*mpt r = *nptr; //z=y;
```

```
y= (*mpt r) +1; //y=z+1;
```

```
}
```



Punteros y verificación de tipos

- Los punteros se enlazan a tipos de datos específicos, de modo que C verificará si se asigna la dirección de un tipo de dato al tipo correcto de puntero
- Así, por ejemplo, si se define un puntero a `float`, no se le puede asignar la dirección de un carácter o un entero

```
float* pf;
```

```
char c;
```

```
pf=&c; //Dará un error
```

- El tamaño de memoria reservado para albergar un puntero es el mismo (usualmente 32 bits) independientemente del tipo de dato al que apunte (todos almacenan una dirección de memoria)

Errores comunes I

■ Uso de punteros no inicializados

```
main() {
    int y = 5, *nptr;
    *nptr=5; //Error
}
```

■ Asignar punteros de distinto tipo

■ Asignación de valores al puntero y no a la variable que apunta

```
main() {
    int y = 5, *nptr = &y;
    nptr=9; //Error
}
```

```
main() {
    int a = 10, *ptri;
    float b = 5.0, *ptrf;

    ptri = &a;
    ptrf = &b;
    ptrf=ptri; //Error
}
```


Errores comunes I

- **Confundir el * al declarar una variable de tipo puntero con el * cuando funciona como operador**

```
main() {  
    /*Declaración de variable. El * funciona como indicador de  
    que ptr es puntero a entero*/  
    int y = 5, *ptr=&y;  
  
    /*Sentencia ejecutable. La expresión *ptr=&y daría un  
    error*/  
    ptr=&y;  
  
    /*Sentencia ejecutable. El * funciona como operador y  
    devuelve el dato al que apunta ptr (y)*/  
    *ptr = 9;  
}
```

Escribir el valor de un puntero

■ Utilizar *printf* con uno de los siguientes códigos:

- **%lu**: vista como entero largo
- **%X**: vista en hexadecimal en mayúsculas
- **%p**: vista en hexadecimal en minúsculas
- **%x**: vista en hexadecimal en minúsculas

```
int a=5;
printf("\n%lu:%lu %X:%X %x:%x %p:%p\n", &a, &a, &a, &a);
```



```
%lu:14745564 %X:E0FFDC %x:e0ffdc %p:e0ffdc
```

Punteros NULL y void

■ Existen dos tipos de punteros especiales:

■ NULL

- Puntero nulo. No apunta a ninguna parte en particular, no direcciona ningún dato válido en memoria
- Proporciona un medio de conocer cuando una variable puntero no direcciona un dato válido
- Definida en *stddef.h*, *stdio.h*, *stdlib.h* y *string.h*.
- También se puede definir en la parte superior del programa
 - `#define NULL 0`

■ void

- Un puntero *void* apunta a cualquier tipo de dato
- Declaración: `void* ptr;`

Punteros a punteros



Eva Lucrecia Gibaja Galindo

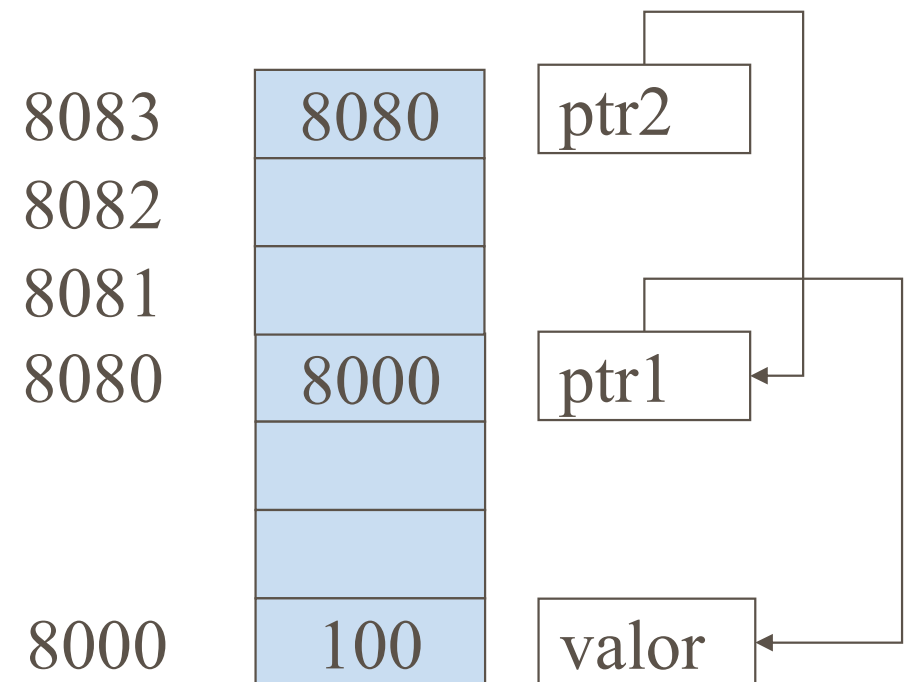
Dpto. Informática y Análisis Numérico

Punteros a punteros

- Un puntero puede apuntar a otra variable puntero
- Declaración:

■ *Tipo de variable ** Nombre de variable*

```
int valor=100;
int * ptr1=&valor;
int** ptr2=&ptr1;
```

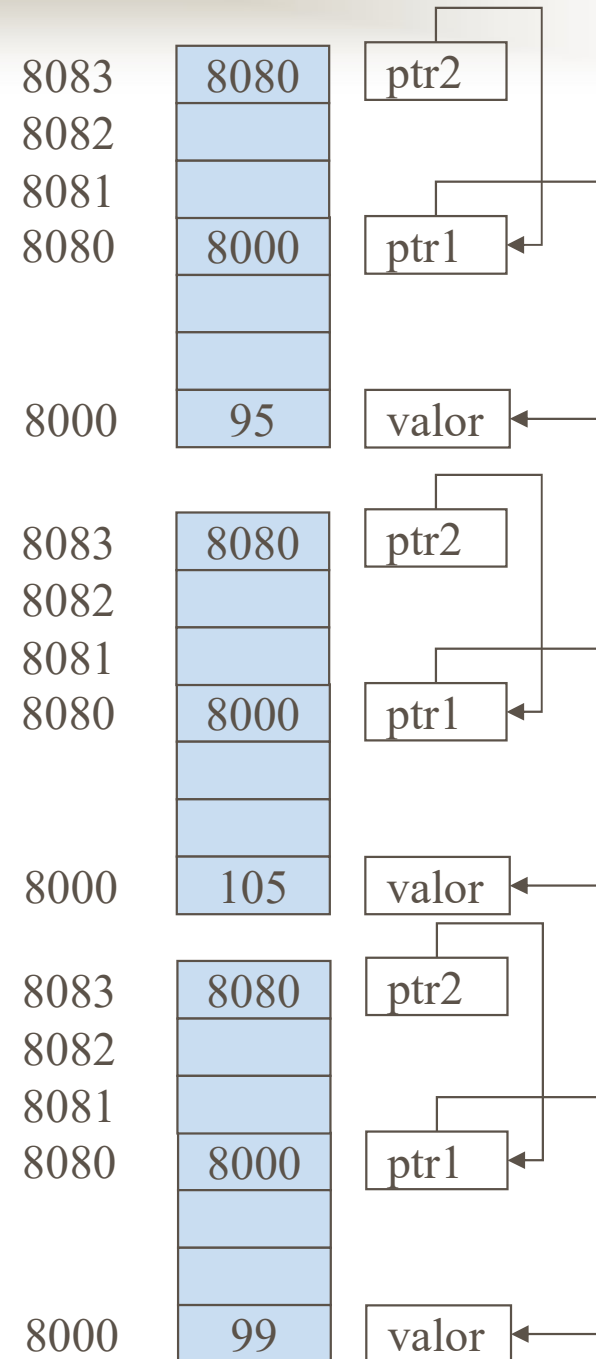


Punteros a punteros

```
valor=95;
```

```
*ptr1=105; //asigna 105 a valor
```

```
**ptr2=99; //asigna 99 a valor
```



Punteros a estructuras



Eva Lucrecia Gibaja Galindo

Dpto. Informática y Análisis Numérico

Operadores de estructuras

- En muchos casos, se utilizan punteros a estructuras

```
struct punto a;
struct punto* p;
p=&a;
printf("%d, %d", (*p).x, (*p).y);
```

```
struct punto
{
    int x;
    int y;
};
```

- Los paréntesis son necesarios en $(*p).x$ debido a que la precedencia del operador “.” es mayor que la de “*”
- La expresión $*p.x$ significa $*(p.x)$, lo cual es ilegal ya que x no es un puntero

Operadores de estructuras

- Los punteros a estructuras son tan frecuentes que se ha proporcionado una notación alternativa como abreviación
- Si p es un puntero a estructura, entonces “ $p \rightarrow \text{miembro de estructura}$ ” se refiere al miembro en particular

```
struct punto a;
a.x=2; a.y=4;
struct punto* p;
p=&a;
printf(“%d, %d”, p->x, p->y);
// Equivale a printf(“%d, %d”, (*p).x, (*p).y);
```

Estructuras autoredefinidas

- Son estructuras que se definen de forma recursiva

```
struct nodoLista
{
    int n;
    struct nodoLista* siguiente;
};
```

Las veremos en el tema de
listas, pilas y colas



Punteros. Paso de parámetros



Eva Lucrecia Gibaja Galindo

Dpto. Informática y Análisis Numérico

Paso de parámetros

- Los módulos se comunican a través de los parámetros:
 - Paso de parámetros por valor o por copia
 - La función recibe una copia de los valores de los parámetros
 - Si el correspondiente parámetro se modifica dentro del módulo, esta modificación no se verá fuera del ámbito del módulo
 - Paso de parámetros por referencia o por variable
 - La función recibe la dirección de memoria del valor del parámetro
 - Si el parámetro se modifica dentro del módulo, esta modificación se verá fuera del ámbito del módulo

ej. paso por valor

```
int cuadrado(int x)
{
    x = x*x;
    return x;
}

void main()
{
    int a, b=4;
    a = cuadrado(b);
    //b = 4
    //a =16;
}
```


Paso de parámetros

■ Tipos de parámetros

- *Parámetros formales.* Son los identificadores definidos en la declaración de un módulo (cabecera)
- *Parámetros reales o actuales.* Son las expresiones pasadas como argumentos en la llamada a un módulo

```
float media3 (float x1, float x2, float x3)
```

```
{return (x1+x2+x3)/3.0;}
```

```
int main()
```

```
{ float med, x;
```

```
.....
```

```
med = media3(1, 2, x);
```

```
}
```



parámetros formales (prototipo)



parámetros actuales (llamada)

Paso de parámetros por referencia

En C TODOS los parámetros se pasan por VALOR

- El paso por referencia (o por variable) se simula mediante punteros:
 - 1 **En la cabecera:** añadir delante del identificador del parámetro formal un *****
 - `void modulo(int* parametroPorReferencia);`
 - Esto indica al compilador que el parámetro actual puede cambiar su valor en función de las operaciones que se realicen dentro del módulo
 - 2 **Dentro del módulo:** el parámetro se trata como un dato más y en caso necesario se utilizará el operador de contenido *****
 - 3 **En la llamada:** el parámetro actual consiste en la dirección de la variable, generalmente obtenida mediante el operador de dirección **&** (excepto si la variable es de tipo puntero, en cuyo caso sólo se pasa el nombre de la misma)
 - `int variable, int* ptr=&variable;`
 - `modulo(&variable);`
 - `modulo(ptr);` //sin & porque ptr es un puntero
- Las constantes, literales y expresiones no pueden ser parámetros actuales pasados por referencia. Deben pasarse por valor

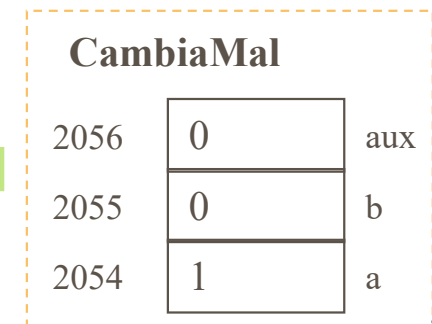
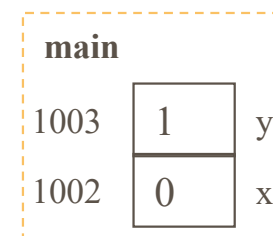
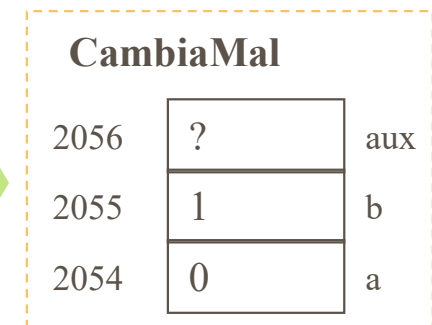
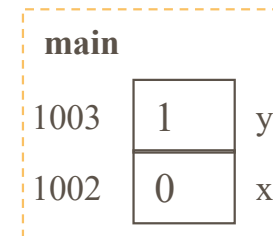
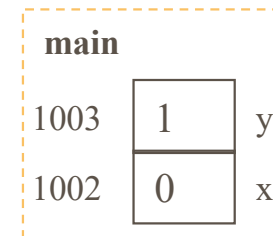
Paso de parámetros por referencia

■ Ejemplo: Intercambiar el valor de dos variables (MAL)

```
#include <stdio.h>
void cambiaMal(char a, char b)
{
    char aux;
    aux = a;
    a = b;
    b = aux;
}

int main()
{
    char x=0, y=1;
    printf("\ncambiaMal:x=%d y=%d =>", x, y);
    cambiaMal(x, y);
    printf("x=%d y=%d", x, y);
}
```

	a	b	aux
inicio	0	1	?
aux=a	0	1	0
a=b	1	1	0
b=aux	1	0	0



Paso de parámetros por referencia

- Ejemplo: Intercambiar el valor de dos variables (BIEN)

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

```
void cambiaBien(char *a, char *b) { 1
```

```
    char aux;
```

```
    aux = *a; 2
```

```
    *a = *b;
```

```
    *b = aux;}
```

```
int main() {
```

```
    char x=0, y=1;
```

```
    printf("\ncambiaBien: x=%d y=%d => ", x, y);
```

```
    cambiaBien(&x, &y); 3
```

```
    printf("x=%d y=%d", x, y);
```

```
    return 0;
```

```
}
```


Paso de parámetros por referencia

- Traza del intercambio del valor de dos variables en C.

**a y b almacenan
una dirección de
memoria**

main		
1002	0	x
1000	1	y

main		
1002	0	x
1000	1	y

CambiaBien		
aux	?	1508
a	1002	1504
b	1000	1500

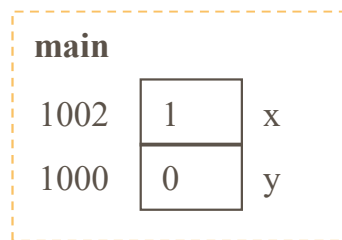
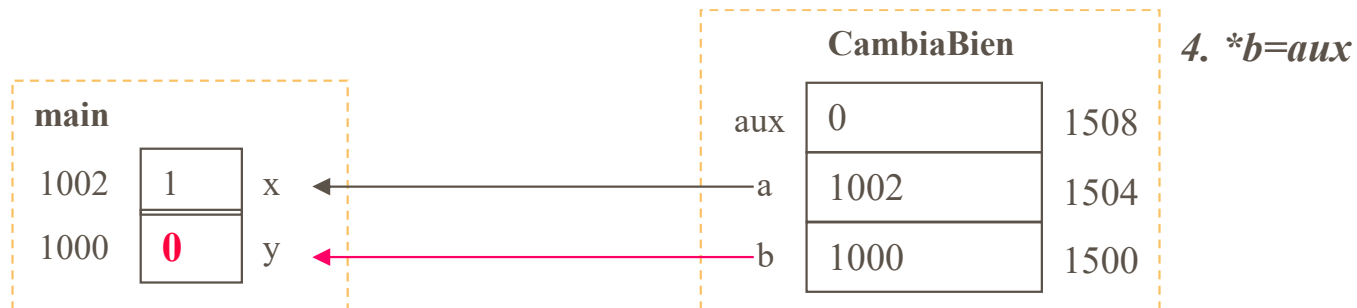
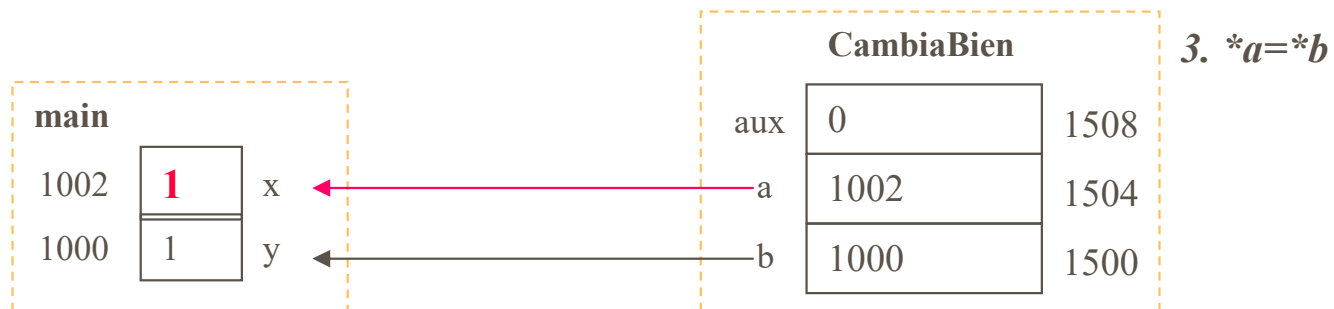
1. Llamada a función

main		
1002	0	x
1000	1	y

CambiaBien		
aux	0	1508
a	1002	1504
b	1000	1500

*2. $aux = *a$*

Paso de parámetros por referencia



Paso de parámetros por referencia

■ Pasos por referencia entre módulos

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

```
void B(int *vb)
```

```
{
```

```
    *vb = *vb * 5;
```

```
}
```

```
void A(int *va)
```

```
{
```

```
    *va = *va + 1;
```

```
    B(va);
```

```
}
```

```
int main()
```

```
{
```

```
    int h;
```

```
    h = 5;
```

```
    A(&h);
```

```
    printf("%i",h);
```

```
    return 0;
```

```
}
```

va es un parámetro de entrada y salida

Como *va* ya es un puntero, no hay que poner &

Paso de parámetros por referencia

- EJEMPLO: Cálculo de cociente y resto de una división entera.

```
#include <stdio.h>

void CocienteEntero (int divdo, int divsor, int* c)
{
    *c= divdo / divsor; //División entera
}

int RestoEntero (int divdo, int divsor)
{
    return (divdo % divsor);
}

1 void Division(int dividendo,int divisor,int *c,int *r)
{
    CocienteEntero (dividendo, divisor, c);
    2 *r = RestoEntero (dividendo, divisor);
}

int main()
{
    int a,b,z,w;
    a = 6;
    b = 3;
    Division(a, b, &z, &w); 3
    printf("%i entre %i = %i\n",a,b,z);
    printf("y el resto es %i",w);
    return 0;
}
```

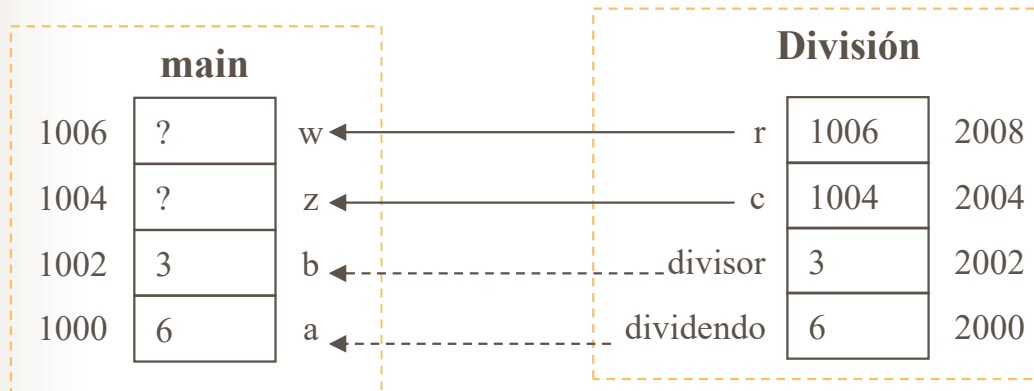
Procurar agrupar primero
los parámetros por valor

Como *c* ya es un puntero, no hay
que poner &

Paso de parámetros por referencia

- Traza del cálculo de cociente y resto de una división entera.

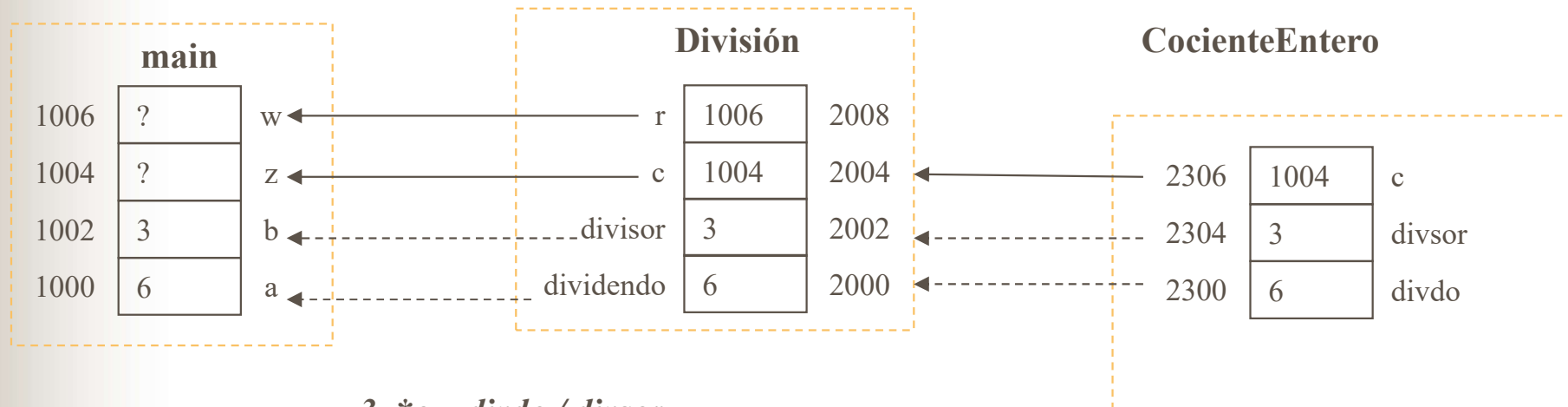
main		
1006	?	w
1004	?	z
1002	3	b
1000	6	a



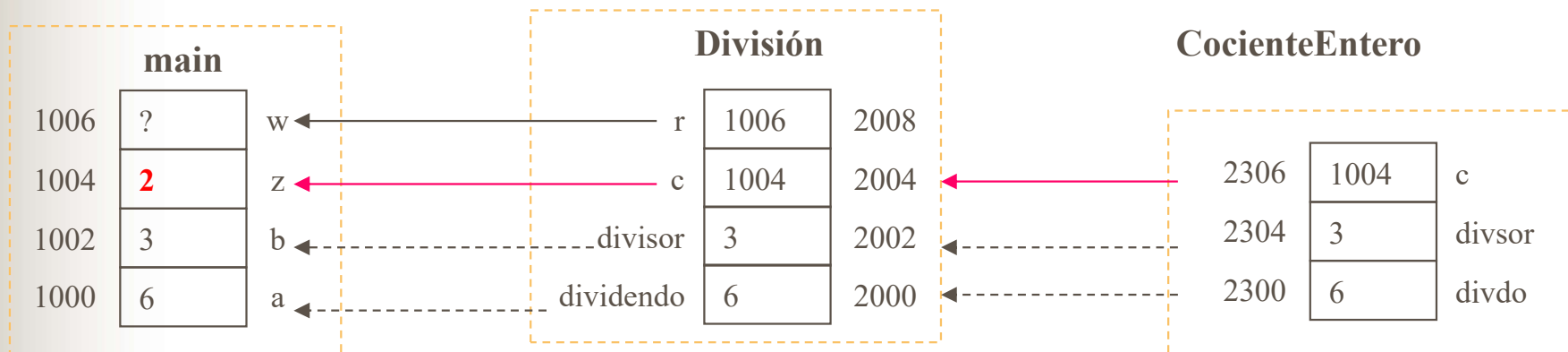
1. Llamada a función División

Paso de parámetros por referencia

2. Llamada CocienteEntero(dividendo, divisor, c)

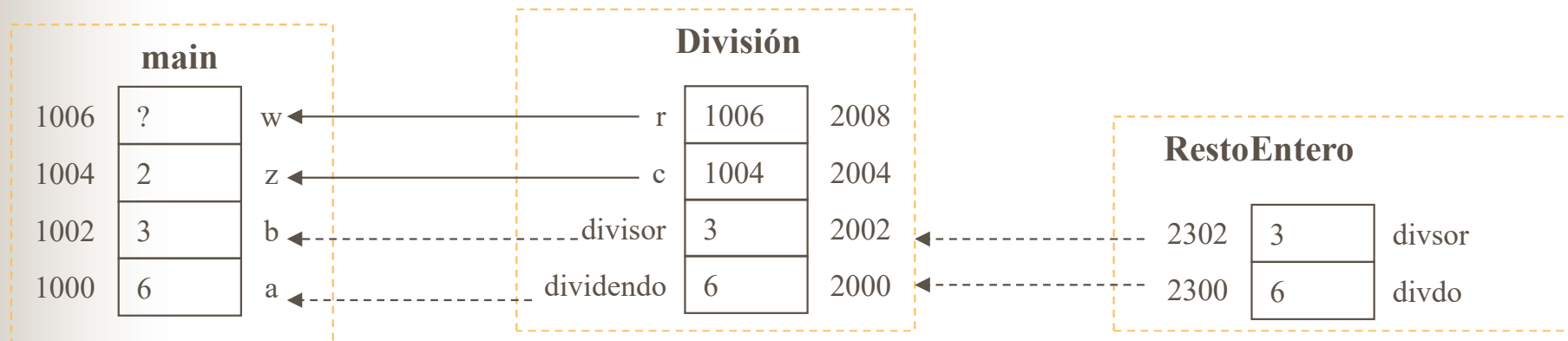


3. *c = divdo / divsor



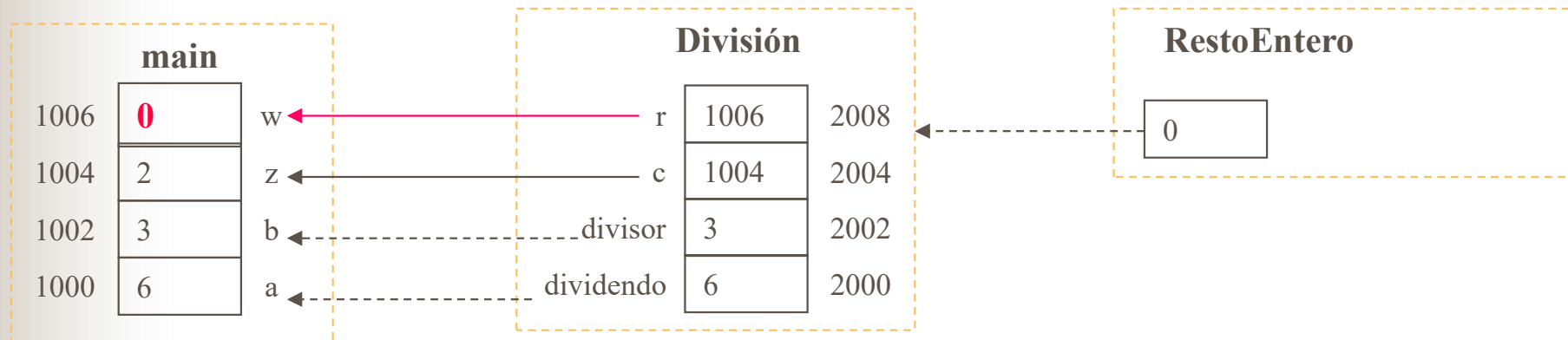
Paso de parámetros por referencia

4. Llamada RestoEntero(dividendo, divisor)



5. return(divdo % divsor)

**r = RestoEntero(dividendo, divisor)*



Paso de estructuras a funciones

- C permite pasar estructuras a funciones, bien por valor, bien por referencia utilizando el operador `&` en la llamada a la función
- Paso por referencia cuando:
 - La estructura es grande, pues el tiempo necesario para copiar el parámetro *struct* a la pila puede ser prohibitivo
 - Queremos cambiar dentro de la función el contenido de algún campo de la estructura

Paso de estructuras a funciones

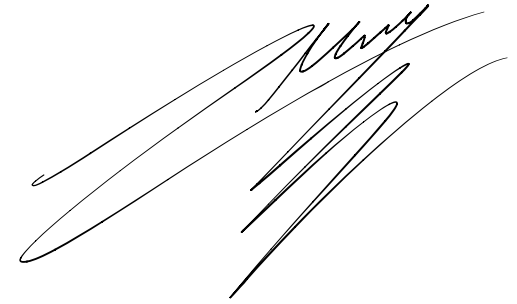
■ Paso por valor

```
void funcion1(struct punto p)
{ //Estos cambios no se ven fuera de la función
    p.x = -3;
    p.y=-3;
}
```

■ Paso por referencia

```
void funcion2(struct punto* p)
{
    p->x=-4; //Equivale a (*p).x = -4;
    p->y=-4; //Equivale a (*p).y = -4;
}
```

```
void main()
{
    struct punto p;
    p.x=2;
    p.y=2;
    funcion1(p); //Paso por valor
    funcion2(&p); //Paso por referencia
}
```



Aritmética de punteros



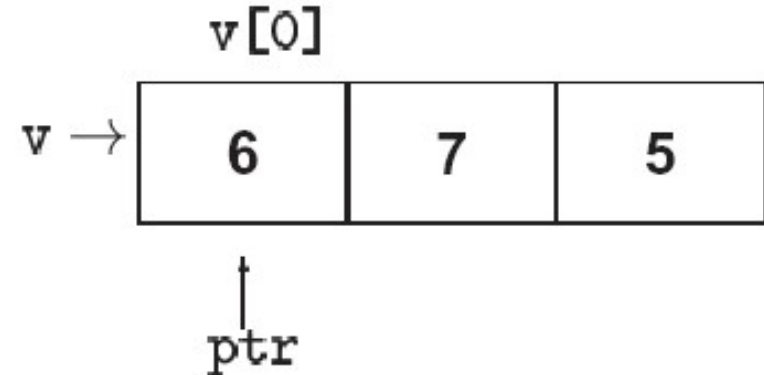
Eva Lucrecia Gibaja Galindo

Dpto. Informática y Análisis Numérico

Punteros y arrays unidimensionales

- El identificador de un vector **estático** es un puntero **CONSTANTE** a la dirección de memoria que contiene el primer elemento

```
int v[3];
int* ptr;
...
ptr=&v[0]; //ptr=v
```



- v es igual a $\&v[0]$
- $*v$ equivale a $v[0]$
 - $v[0]=6$ equivale a $*v=6$ y a $\&v[0]=6$

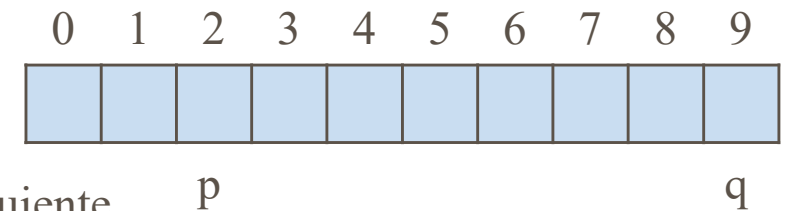
Aritmética de punteros

■ En general

- Asignación (=). Han de ser del mismo tipo, en otro caso utilizar *casting*
 - $p=q$
- Relacionales
 - $p==q$, $p!=q$, $p==NULL$, $p!=NULL$

■ Solo con vectores

- Suma/resta con literales (enteros)
 - $p=p+n$. Si p es un puntero a un *tipo*, $p+n$ devuelve un puntero a la posición de memoria *sizeof(tipo)*n bytes*
 - $p=p+10$
 - $p+=n$ //(p=p+n)
 - $p-=n$ //(p=p-n)
 - $p++$ //(p=p+1) //Dirección del elemento siguiente
 - $p--$ //(p=p-1) //Dirección del elemento anterior
- Relacionales
 - $p<q$, $p>q$
 - $q-p$ es el número de objetos que separa a p y q . Por ejemplo, si q apunta al elemento 10 y p apunta al elemento 3, $q-p$ es 7



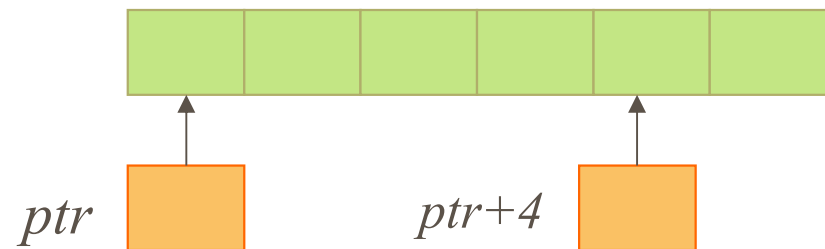
■ Operaciones **NO** válidas

- Sumar, multiplicar o dividir punteros

Punteros y arrays unidimensionales

- Como puntero, un vector v obedece las leyes de la aritmética de punteros

- v apunta a $v[0]$
- $(v+i)$ apunta a $v[i]$
- $*(v+i)$ es equivalente a $v[i]$



- Recíprocamente, a los punteros se les puede poner corchetes:

- $*(ptr+i)$ es equivalente a $ptr[i]$

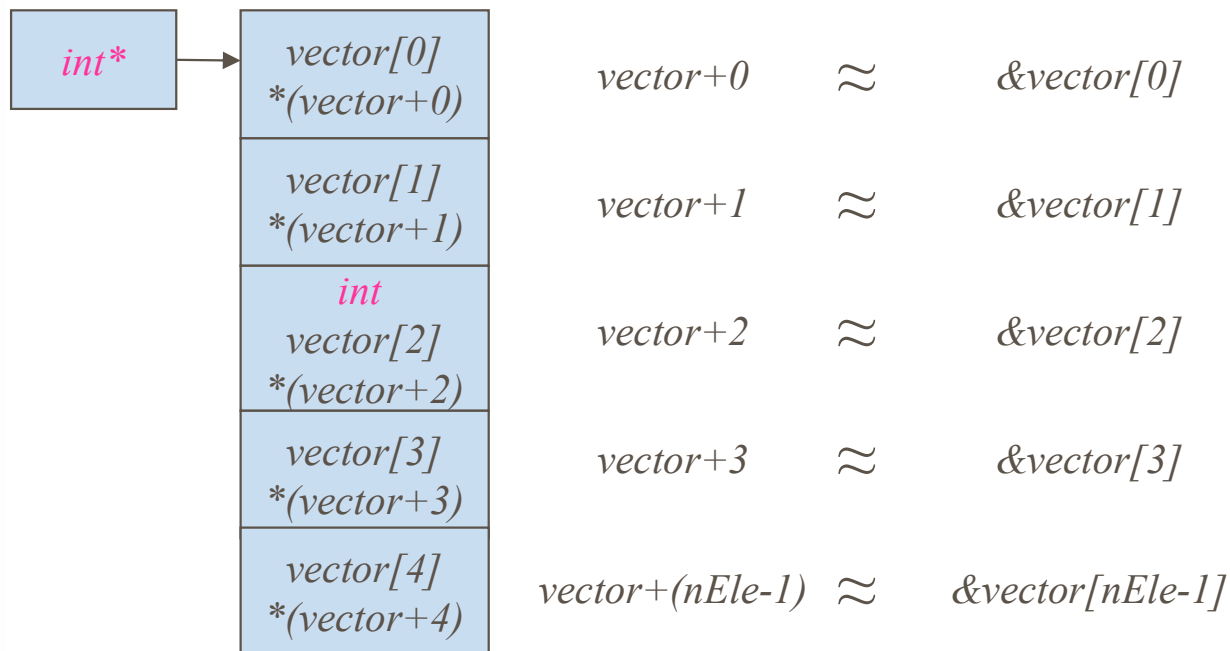
- Pero recordad: Un vector estático es un puntero constante

```
int v[2]={0, 1}  
v++; //error de compilación  
p=v;  
p++; //si es válido
```

Punteros y arrays unidimensionales

```
int vector[5]={10, 20, 30, 40, 50};
```

vector



Punteros y arrays unidimensionales

```
//Distintas formas de sumar los elementos de un vector
void main()
{
    int i, suma, nEle=5;
    int* ptr, *ptrfin;
    int V[5]={1, 2, 3, 4, 5};

    /*-----*/
    suma=0;
    for(i=0; i<nEle; i++)
    {
        suma = suma+V[i];
    }
    printf("Resultado para suma=suma+V[i]:%d\n", suma);
}
```

Punteros y arrays unidimensionales

```

/*-----*/
suma=0;
ptr=V;
for(i=0; i<nEle; i++)
{
    suma = suma+ptr[i];
}
printf("Resultado para suma = suma+ptr[i]:%d\n", suma);
/*-----*/
suma=0;
ptr=V;
for(i=0; i<nEle; i++)
{
    suma = suma+*(ptr+i);
}
printf("Resultado para suma = suma+*(ptr+i):%d\n", suma);

```


Punteros y arrays unidimensionales

```
/*-----*/
```

```
suma=0;
for(i=0; i<nEle; i++)
{
    suma = suma+(* (V+i));
}
```

```
printf("Resultado para suma = suma+(* (V+i)):%d\n", suma);
```

```
/*-----*/
```

```
suma=0;
ptrfin=V+nEle;
for(ptr=V; ptr<ptrfin; ptr++)
{
    suma = suma+*ptr;
}
```

```
printf("Resultado para suma = suma+*ptr:%d\n", suma);
```

```
}
```

Punteros y arrays unidimensionales

```

/*-----*/
suma=0;
ptr=V;
for(i=0; i<nEle; i++)
{
    suma = suma+*ptr++;
}
printf("Resultado para suma = suma+*ptr++:%d\n", suma);

```

*primero se aplica **
luego se aplica ++

Punteros y arrays bidimensionales

- Dado un *array* bidimensional (matriz estática): `int M[DIMF][DIMC];`

- Los elementos se almacenan fila a fila **consecutivos** en memoria

$$\begin{pmatrix} 5 & 1 & 9 \\ 7 & 8 & 6 \end{pmatrix} \Rightarrow 5 \quad 1 \quad 9 \quad 7 \quad 8 \quad 6$$

- `M[i]` es un puntero que apunta al principio de la fila *i* de la matriz

`M[0]` apunta a `M[0][0]`

`M[1]` apunta a `M[1][0]`

- Para resolver la dirección de `M[i][j]`, el compilador calcula:

$$\text{dirección}(i,j) = \text{dirección}(0,0) + i * \text{DIMC} + j$$

- *i**dimC nos permite acceder a la fila correcta
- *j* nos indica el desplazamiento dentro de la fila
- Por este motivo, cuando pasamos un array bidimensional a una función, no es necesario indicar el número máximo de filas:

```
void imprimirMatriz(int Matriz[][DIMC], int nFil, int nCol)
```

- Para acceder al elemento `M[i][j]` utilizamos el operador `*`:

$$M[i][j] = *dirección(i, j) = *(dirección(0,0) + (i * DIMC + j))$$

Punteros y arrays bidimensionales

- Ejemplo: Elimina los valores negativos de una matriz bidimensional, sustituyéndolos por ceros

```
#define DIMF 3
#define DIMC 2
int main()
{
    int i, j, nElem, * ptr;
    int matriz[DIMF][DIMC]={1, 1}, {-1, -1}, {1, 1}};

    //Recorrido de la matriz utilizando un puntero a entero
    nElem = DIMF*DIMC;
    ptr=matriz[0]; //ptr=&matriz[0][0]
    for(i=0; i<nElem; i++)
    {
        if(*ptr<0)
            *ptr=0;
        ptr++;
    }
    //Mostrar los elementos de la matriz utilizando un int*
    ptr=matriz[0]; //ptr=&matriz[0][0]
    for(i=0; i<DIMF; i++)
        for(j=0; j<DIMC; j++)
            printf("matriz[%d][%d]: %d   ptr(%d): %d\n", i, j, matriz[i][j], i*DIMC+j, *(ptr+i*DIMC+j));

    return 0;
}
```

La matriz se almacena en posiciones consecutivas en memoria

Arrays unidimensionales y funciones

- Un módulo puede recibir de entrada un dato de tipo vector (*array* unidimensional)
- Dado que un vector es un puntero, las siguientes cabeceras también serían válidas
 - `void imprimirElementos(int vector[], int tope);`
 - **`void imprimirElementos(int *vector, int tope);`**
- Al hacer el paso de un vector a una función, decimos que se hace *“POR REFERENCIA”*
 - El vector en si (el puntero al primer elemento), se pasa **por valor**, no puede cambiar la dirección de comienzo del vector
 - Los elementos apuntados por el vector si que pueden ser modificados y ese cambio se verá fuera de la función

Arrays bidimensionales y funciones

- ¿Podemos generalizar lo visto a *arrays* bidimensionales?
 - **NO!!!!**
- El nombre de un *array* bidimensional es un puntero al primer elemento de un *array* de punteros (`int* ptr[]`), no un puntero a puntero (`int** ptr`)
- Con memoria estática:
 - Utilizaremos la notación de corchetes
 - `void imprimirMatriz(int mat[][DIMC], int nfil, int ncol);`
 - NO podemos hacer:
 - `void imprimirMatriz(int** mat, int nfil, int ncol);`
- Con memoria dinámica:
 - Utilizaremos notación de punteros
 - `void imprimirMatriz(int** mat, int nfil, int ncol);`

Punteros. Cadenas de caracteres



Eva Lucrecia Gibaja Galindo

Dpto. Informática y Análisis Numérico

Cadenas de Caracteres

■ *Ejemplo.* Longitud de una cadena

```
#include <stdio.h>

#define TOPE 30

void main()
{
    char cadena[TOPE] = "Hola";

    int longitud;

    char* ptr;

    /*Primera version, con indices*/

    for(longitud=0; cadena[longitud]!='\0'; longitud++);

    //No hacemos nada, solo nos interesa incrementar el valor de longitud

    printf("\nLongitud (primera version): %d", longitud);

    /*Segunda version, con un puntero para recorrer el vector*/

    for(ptr=cadena; *ptr!='\0'; ptr++);

    printf("\nLongitud (segunda version): %d", ptr-cadena);

}
```

cadena	'H'	'o'	'l'	'a'	'\0'	...	'?'
	0	1	2	3	4	...	29

Manejo de Cadenas de Caracteres

- Las funciones más útiles de C para manejar cadenas se encuentran en el archivo de cabecera **string.h**.
- **int strlen(char *)** . Devuelve la longitud de la cadena de caracteres (sin incluir el carácter nulo)

```
#include <stdio.h>
#include <string.h>
void main()
{
    char cad[15]="Hola";
    printf("%d", strlen(cad)); //Imprime 4
}
```

Manejo de Cadenas de Caracteres

■ **char *strcpy(char *dest, char *orig)**

- Copia orig en dest. Devuelve dest. Se suele ignorar el valor devuelto y se usa como si fuera una función void

```
#include <stdio.h>
#include <string.h>
void main()
{
    char cad[15]="Hola", cad2[15];
    cad2=cad; //Error
    strcpy(cad2, cad); //Bien
}
```

Manejo de Cadenas de Caracteres

- **int strcmp(char *cad1, char *cad2).** Compara ambas cadenas y devuelve:
 - 0 si las dos son idénticas
 - <0 si la primera cadena precede *alfabéticamente* a la segunda
 - >0 si la segunda cadena precede *alfabéticamente* a la primera

```
#include <stdio.h>
#include <string.h>
void main()
{
    char cad[15]="Hola", cad2[15]="Mola";
    if(strcmp(cad, cad2)==0)
        printf("Las cadenas son iguales");
}
```

Manejo de Cadenas de Caracteres

- **`char *strcat(char *cad1, char *cad2)`**
 - Concadena `cad2` a la cadena `cad1`. Devuelve `cad1`. Se suele ignorar el valor devuelto y se usa como si fuera una función `void`. Debemos asegurarnos de que `cad1` tiene espacio suficiente para albergar el resultado de la concatenación
- **`char *strstr(char *cad1, char *cad2)`**
 - Búsqueda de subcadenas. Devuelve un puntero a la primera aparición de `cad2` en `cad1`, o `NULL` si no se encuentra

Manejo de Cadenas de Caracteres

■ `char *strchr(char *cad, int c)`

- Búsqueda de un carácter en una cadena. Devuelve un puntero a la primera aparición de `c` en `cad`, o `NULL` si no se encuentra

■ `char *strrchr(char *cad, int c)`

- Búsqueda de un carácter en una cadena. Devuelve un puntero a la última aparición de `c` en `cad`, o `NULL` si no se encuentra

Manejo de cadenas de caracteres

```
#include <stdio.h>

#include <string.h>

void main()

{   char cadena[100]="Hola como estas";

    char* ptr;

    ptr=strstr(cadena, "como"); //Busqueda de subcadenas

    printf("\n<%s> contiene a <como> en la direccion <%p> posicion %d", cadena,
ptr, ptr-cadena);

    ptr=strchr(cadena, 'a'); //Busca carácter en cadena (primera aparicion)

    printf("\nPrimera aparicion de <a> en <%s>: direccion <%p> posicion %d",
cadena, ptr, ptr-cadena);

    ptr=strrchr(cadena, 'a'); //Busca carácter en cadena (ultima aparicion)

    printf("\nUltima aparición de <a> en <%s>: direccion <%p> posicion %d",
cadena, ptr, ptr-cadena);
```

Errores comunes II

```
int main ()
{
    char cad[] = "abcde";
    char* ptr = cad;
    //error en tiempo de ejecucion
    //sobrepasamos el espacio reservado para cad
    strcpy(ptr, "Hasta luego, encantada de conocerte");
}
```

```
int main()
{
    char *p;
    //Error, no hemos reservado memoria para los datos
    //apuntados por el puntero
    strcpy (p, "wxsjkwe");
    printf("<%s>", p);
    return 0;
}
```

Errores comunes II

- La función retorna un puntero que apunta a una variable o *array* declarados como **locales**

```
#include <stdio.h>
char * f1()
{
    char buffer[128];
    printf("\nEscriba su nombre: ");
    scanf("%s", buffer);
    return buffer;
}
```

- Al salir de la función todas las variables locales desaparecen

- **No** retornar nunca un puntero que apunte a una variable declarada como local

```
int main()
{
    char* ptr;
    ptr = f1();
    printf("\n<%s>", ptr);
}
```


Punteros. Complementario



Eva Lucrecia Gibaja Galindo

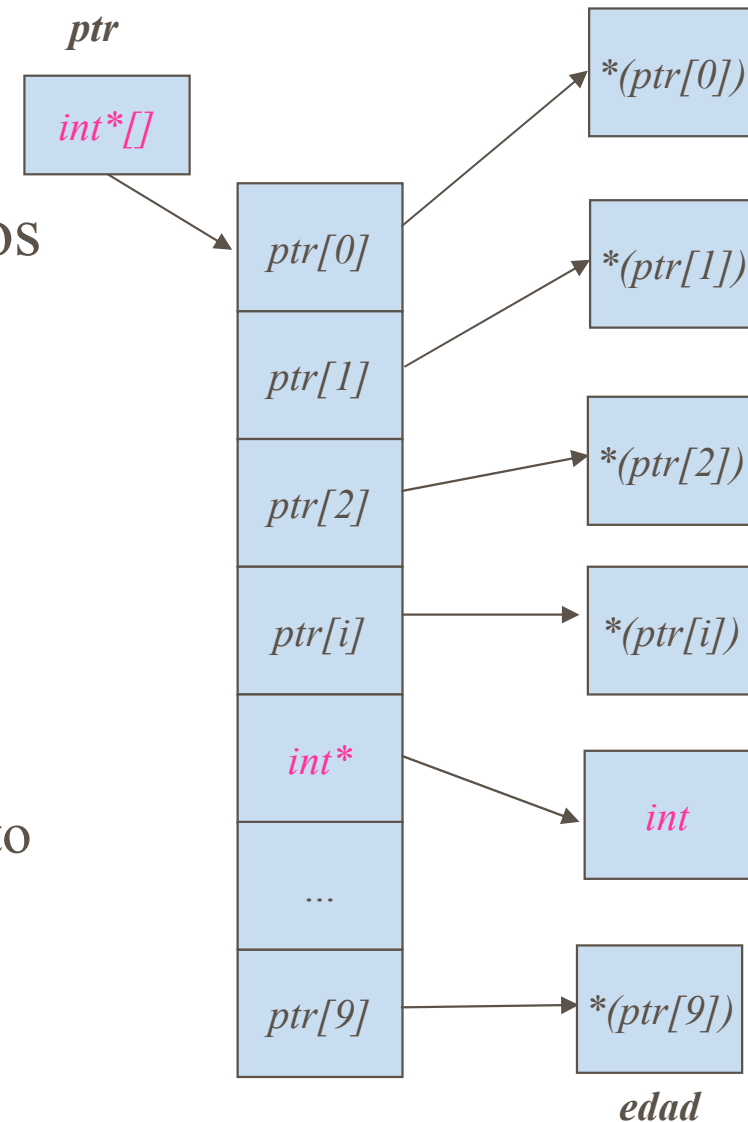
Dpto. Informática y Análisis Numérico

Punteros y literales de cadena

- Un literal de cadena es un conjunto de caracteres encerrados entre comillas
 - `char * ptr="Hasta luego";`
 - Crea dos entidades:
 - El puntero en la zona de datos locales
 - El literal en la zona de datos
- Los literales de cualquier tipo son tratados como valores **constantes** y son almacenados, a diferencia de las variables locales, en el segmento de datos
- Un error muy común es intentar cambiar el contenido del objeto apuntado por `ptr`. Es constante!!

Arrays de punteros

- Cuando se necesite reservar muchos punteros a muchos valores diferentes
- Array de punteros:
 - //Reserva un array de 10 punteros a enteros.
 - `int* ptr[10];`
 - Cada elemento `ptr[i]` almacena una dirección de memoria, que no tiene por qué estar consecutiva con el resto de elementos
 - `ptr[9]=&edad;`



Arrays de punteros

```
#include <stdio.h>
void main()
{
    int* ptr[10];
    int vector[10];
    int i;

    //Inicializar contenido;
    for (i=0; i<10; i++)
    {
        //Inicializar vector de enteros
        vector[i]=i;
        //Inicializar vector de punteros
        ptr[i]=&vector[i];
        *(ptr[i])=vector[i];
        //Imprimir
        printf("\n*(ptr[%d]): %d", i, *(ptr[i]));
    }
}
```


Punteros constantes y punteros a constantes

	Declaración	Ejemplo	p	*p
Punteros constantes	<tipo> *const <nombrePuntero> = <direccionDeVariable>	int x; int* const p = &x;	Es constante. No puede cambiar su valor	Es variable. Puede cambiar su valor
Punteros a constantes	const <tipo> * <nombrePuntero> = <direccionDeConstante>	const int x=25; const int* p = &x;	Es variable, puede cambiar su valor (apuntar a otra constante)	Es constante. No puede cambiar su valor
Punteros constantes a constantes	const <tipo> * const <nombrePuntero> = <direccion de constante>	const int x=25; const int* const p = &x;	Es constante	Es constante

Punteros constantes y punteros a constantes

```
int y=0;  
//Puntero constante  
int x=3;  
int* const p = &x;
```

```
//Puntero a constante  
const int x1=4;  
const int* p1=&x1;
```

```
//Puntero constante a constante  
const int x2=5;  
const int* const p2=&x2;
```

```
//p=&y; warning: p es constante, no podemos cambiar su valor  
*p=30; //Si podemos cambiar *p  
//*p1=30;warning: *p1 es constante, no podemos cambiar su valor  
p1=&y; //Si podemos cambiar p1  
//*p2=30; warning: *p2 es constante, no podemos cambiar su valor  
//p2=&y; warning: p2 es constante, no podemos cambiar su valor
```