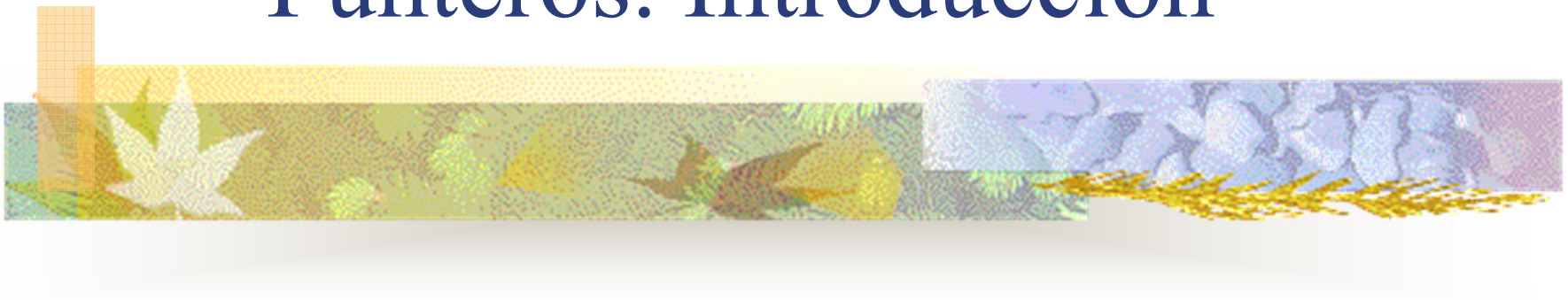
Punteros. Introducción

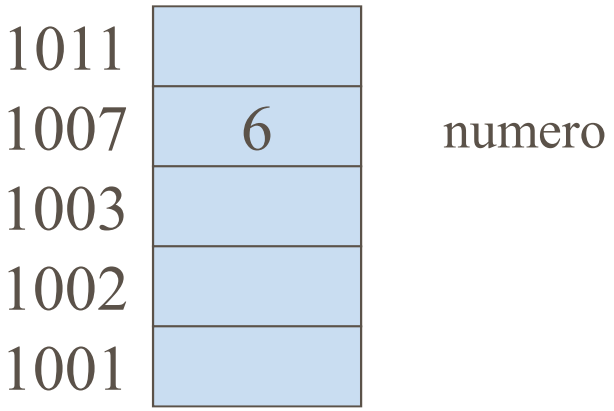


Eva Lucrecia Gibaja Galindo
Dpto. Informática y Análisis Numérico

Introducción

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

```
int numero=6;
```



- 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

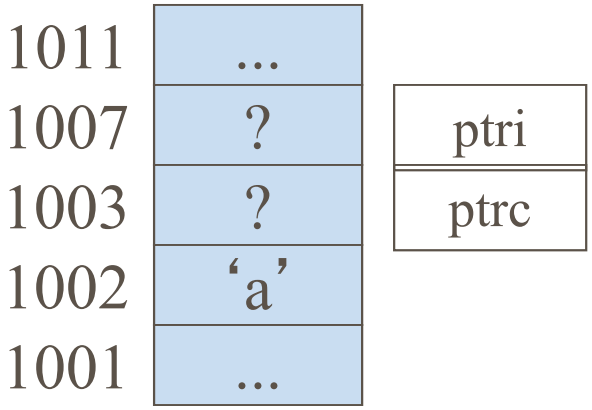
```
Tipo de variable *  Nombre de variable
```

  - 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;
```



- 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*

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)

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*

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

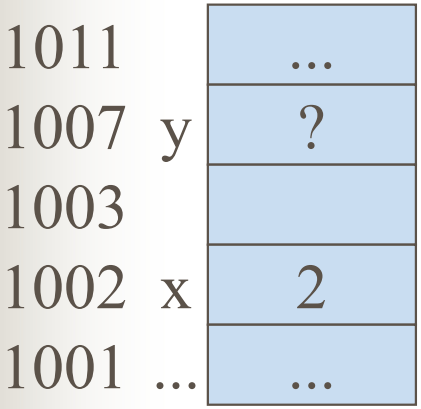
```
char c, *ptrc;
ptrc=&c;
*ptrc='A' //Equivalente a c='A'
```

    - *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*

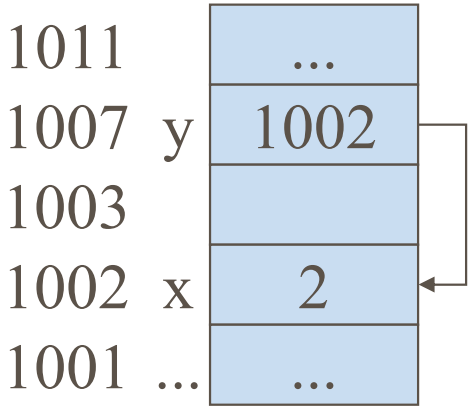
Operadores

- Ejemplo:

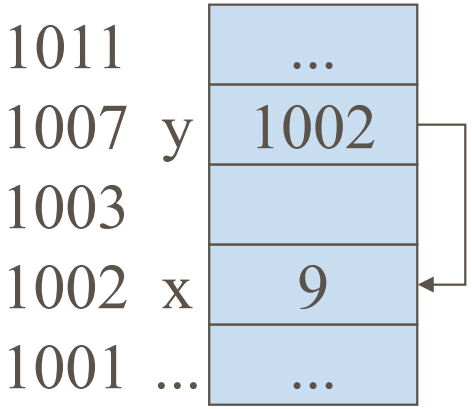
```
[1] int x=2; /* variable normal*/
[2] int * y; /*variable puntero*/
[3] y=&x; /* y contiene la dirección de x*/
[4] x=(*y)+7;
```



[1] y [2]



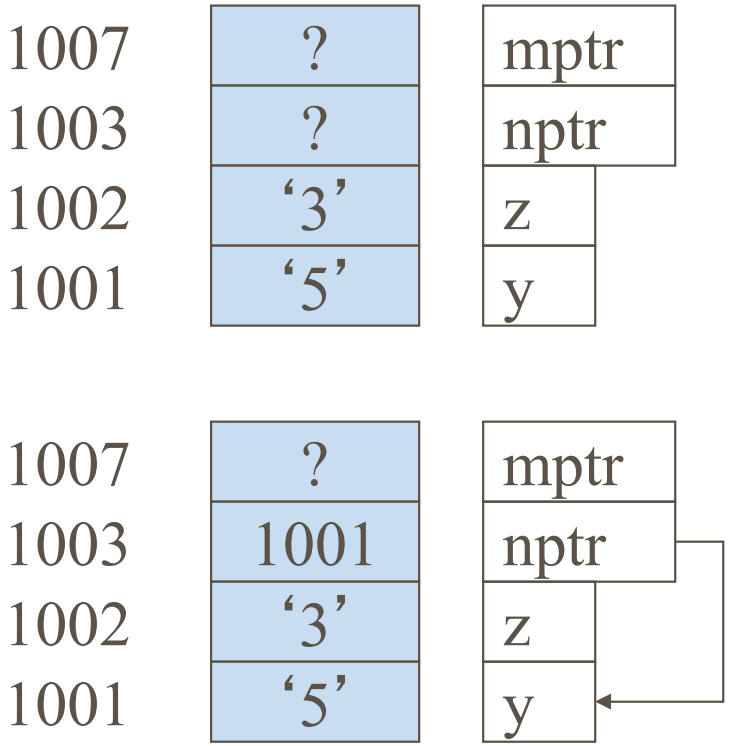
[3]



[4]

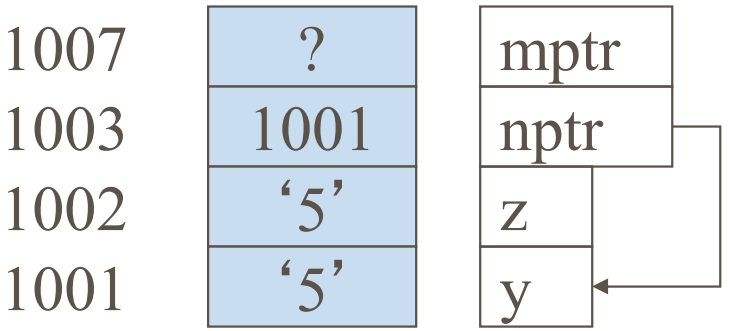
Ejemplo

```
void main()
{
   char y = '5';
   char z ='3';
   char* nptr;
   char* mptr;

   nptr=&y;
```

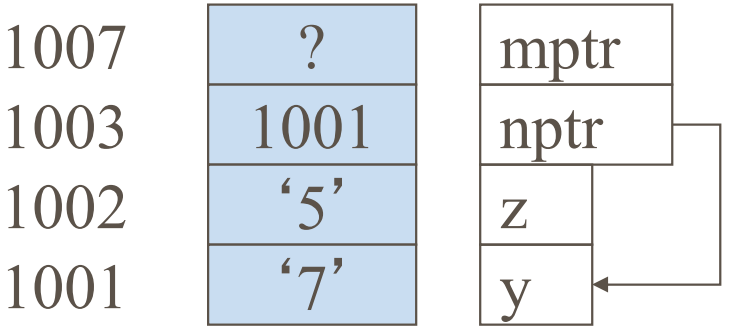


Ejemplo

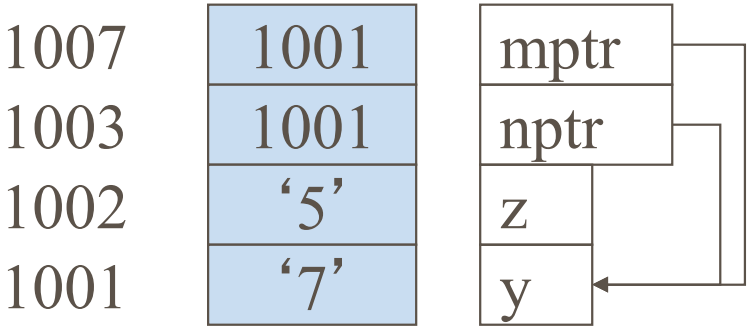
```
z=*nptr;
```




```
*nptr='7';
```

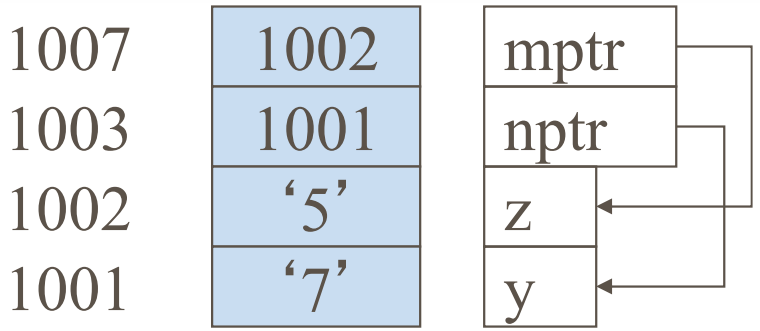



```
mptr = nptr;
```

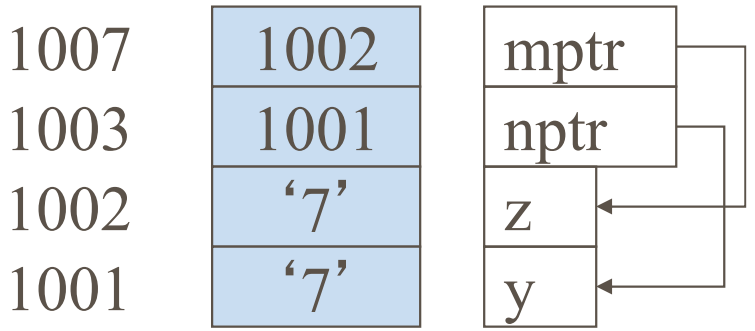


Ejemplo

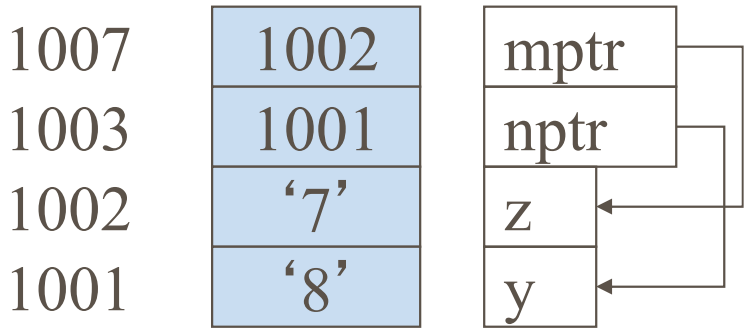
```
   mptr=&z;
```




```
   *mptr = *nptr;
```




```
   y=(*mptr)+1;
}
```



Errores comunes I

- **Uso de punteros no inicializados**

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

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

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

- **Asignar punteros de distinto tipo**

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

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

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;`

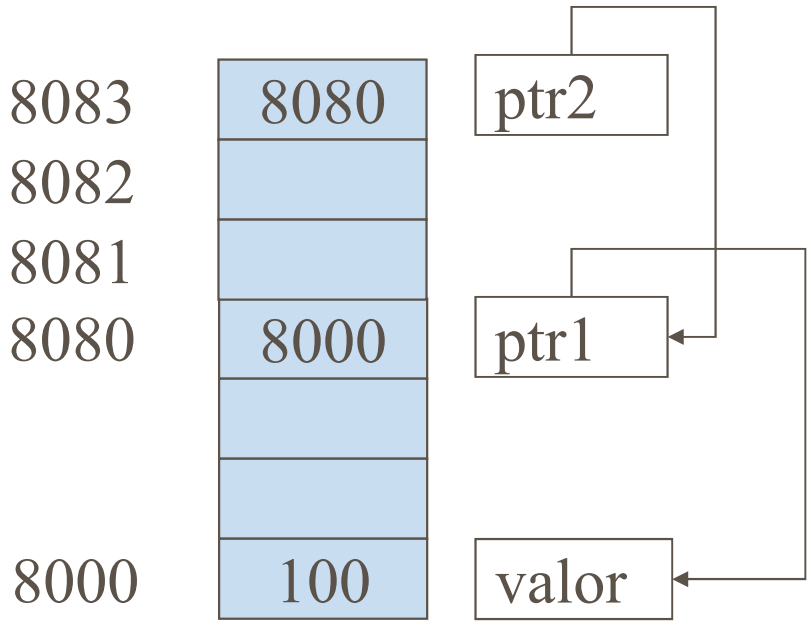
Punteros a punteros

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;
```

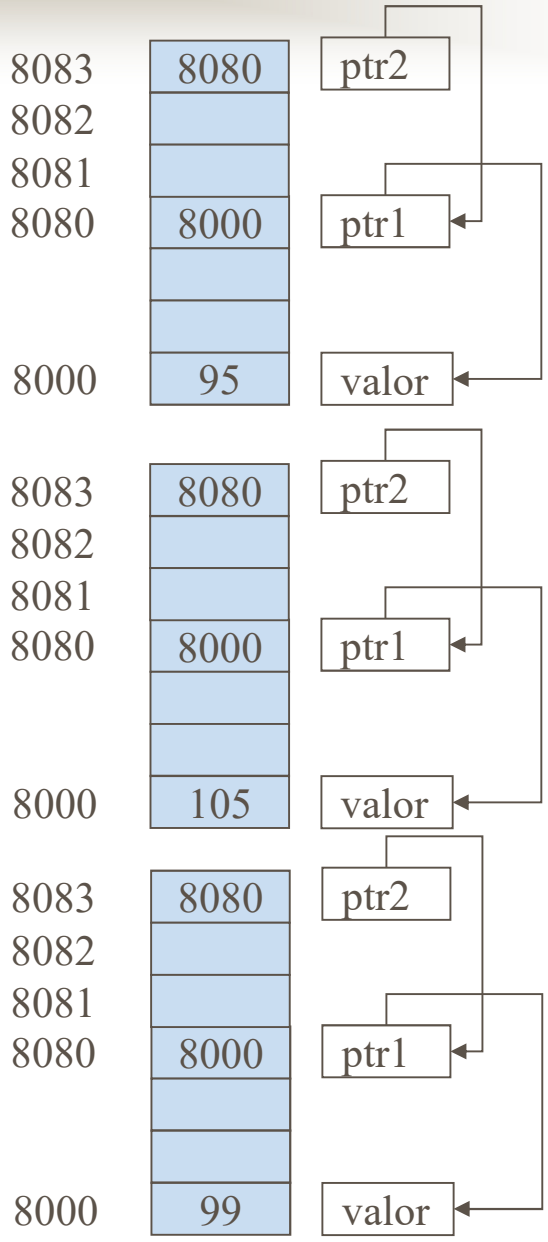


Punteros a punteros

```
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);
```

- 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→miembro de estructura*" se refiere al miembro en particular

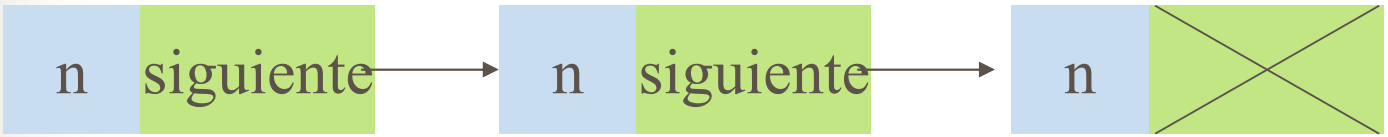
```
struct punto a;
struct punto* p;
p=&a;
printf("%d, %d", p->x, p->y);
// Equivale a printf("%d, %d", (*p).x, (*p).y);
```

Estructuras autoredefinidas

- Son estructuras que se definen de forma recursiva (las veremos en el segundo cuatrimestre)

```
struct nodoLista
{
  int campoClave;
  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;`
     - `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

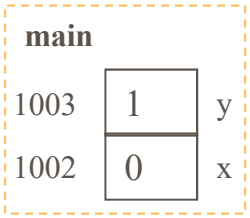
cambia.c

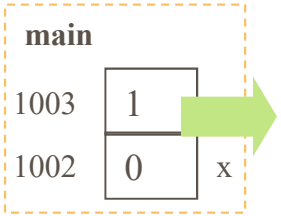
Paso de parámetros por referencia

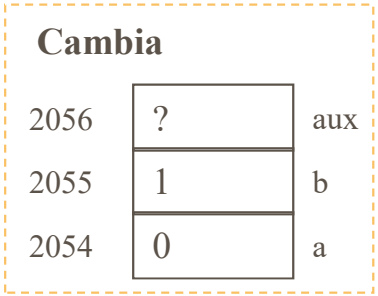
- Ejemplo: Intercambiar el valor de dos variables (MAL)

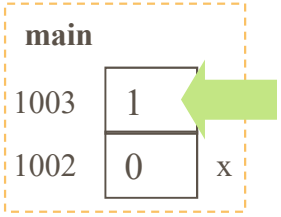
```
#include <stdio.h>
void cambiaMal(char a,char b)
{
    char aux;
    aux = a;
    a = b;
    b = aux;
}

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);
}
```

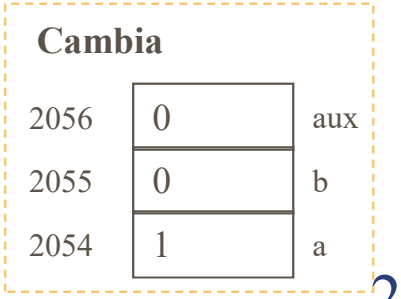
	a	b	aux
inicio	0	1	?
aux=a	0	1	**0**
a=b	**1**	1	0
b=aux	1	**0**	0











cambia.c

Paso de parámetros por referencia

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

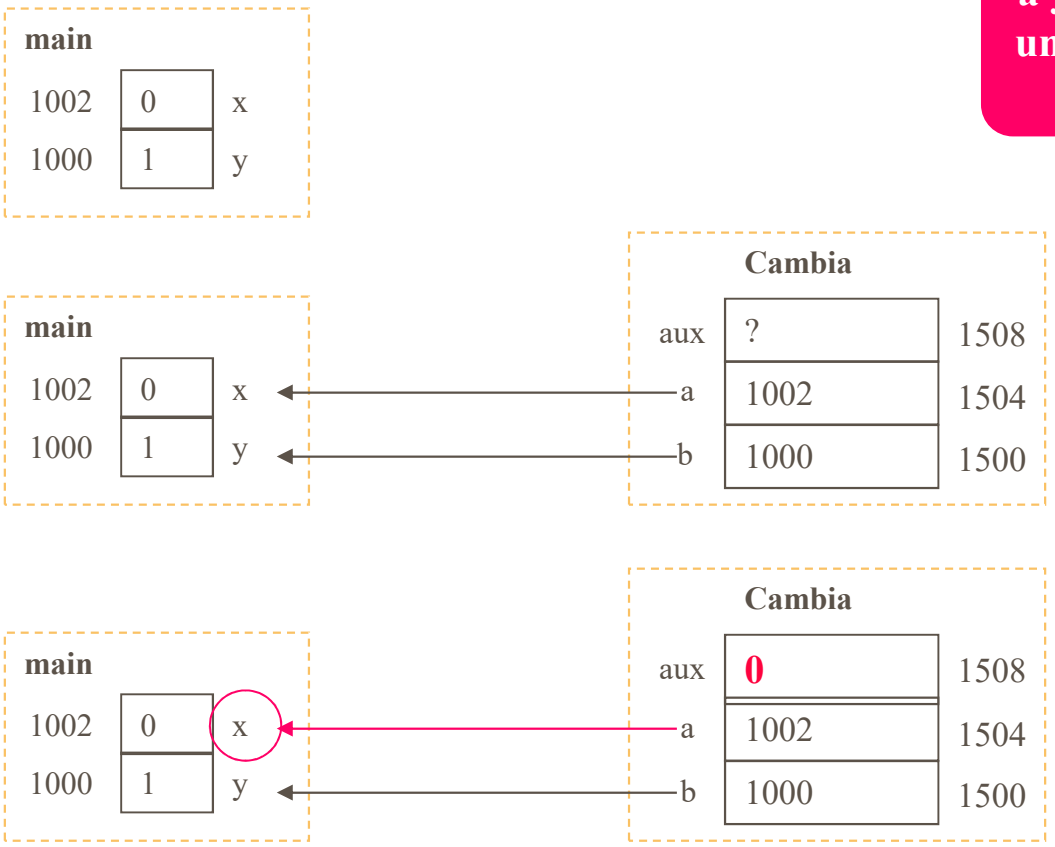
```
#include <stdio.h>
void cambiaBien(char *a, char *b) {
    char aux;
    aux = *a;
    *a = *b;
    *b = aux;}
int main() {
    char x=0, y=1;
    printf("\ncambiaBien: x=%d  y=%d => ", x, y);
    cambiaBien(&x,&y);
    printf("x=%d  y=%d", x, y);
    return 0;
}
```

1 2 3

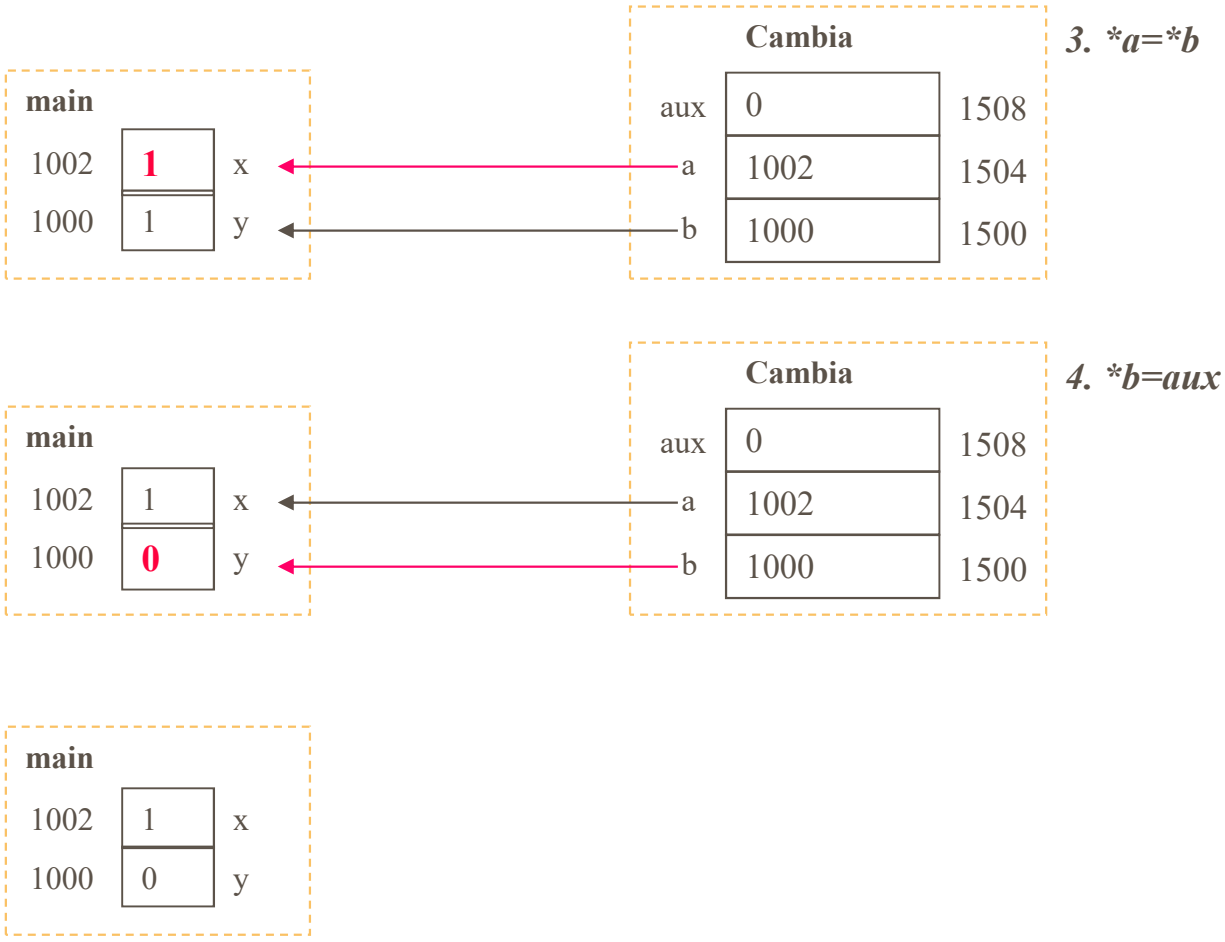
Paso de parámetros por referencia

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

a y b almacenan una dirección de memoria



Paso de parámetros por referencia



pasoReferencia.c

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 &

cocienteResto.c

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;
}
int RestoEntero (int divdo, int divsor)
{
  return (divdo % divsor);
}
void Division(int dividendo,int divisor,int *c,int *r) }
{
  CocienteEntero (dividendo, divisor, c);
  *r = RestoEntero (dividendo, divisor);
}
```

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

1

2

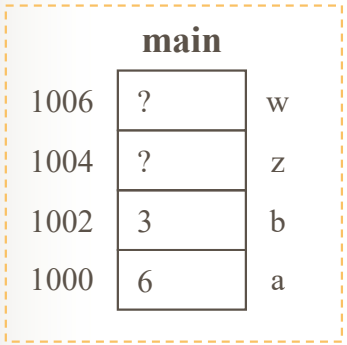
3

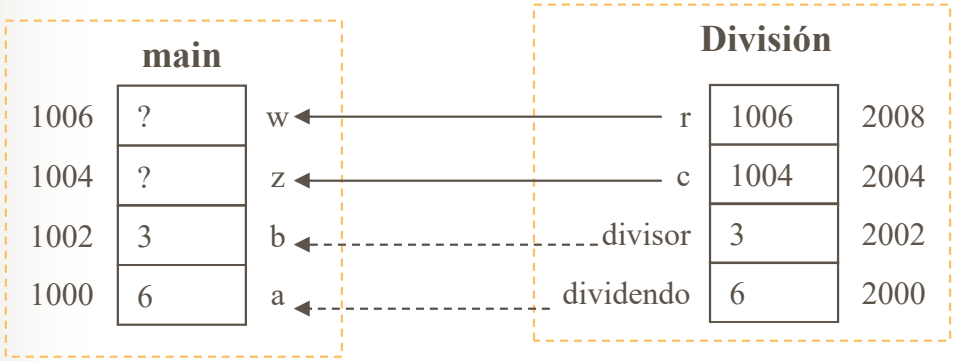
Procurar agrupar primero los parámetros por valor

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.

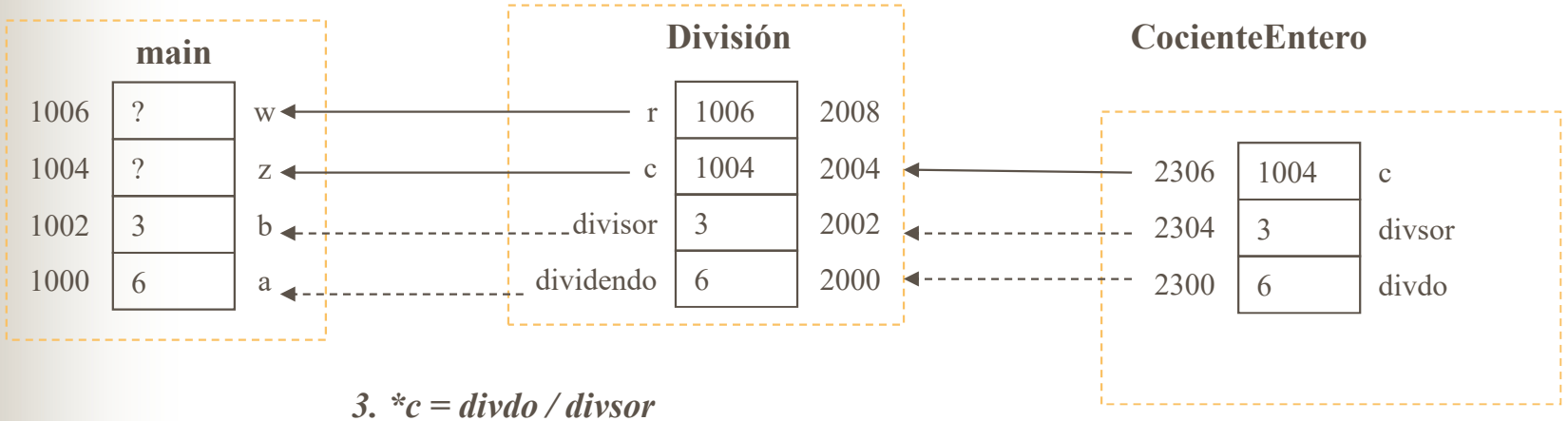




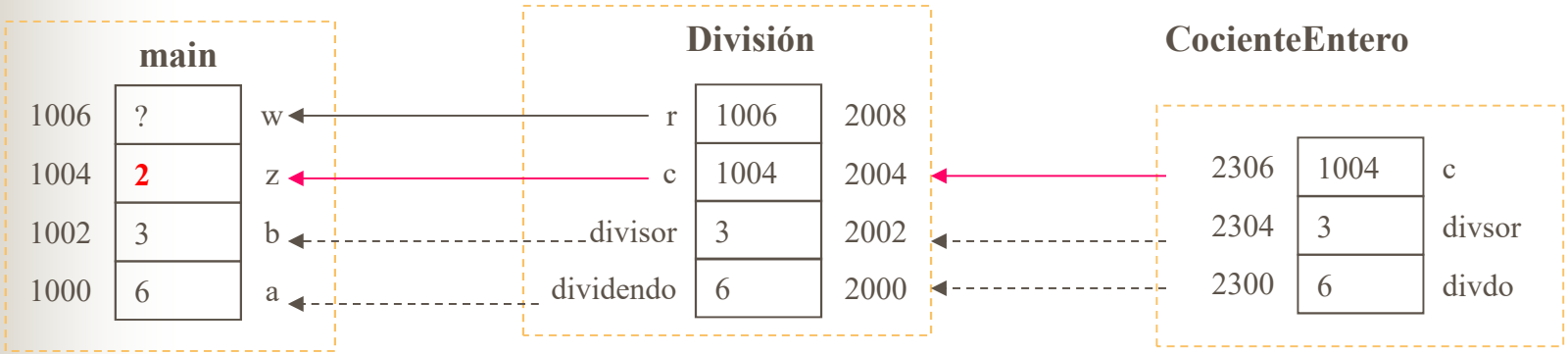
1. Llamada a función División

Paso de parámetros por referencia

2. Llamada CocienteEntero(dividendo, divisor, c)

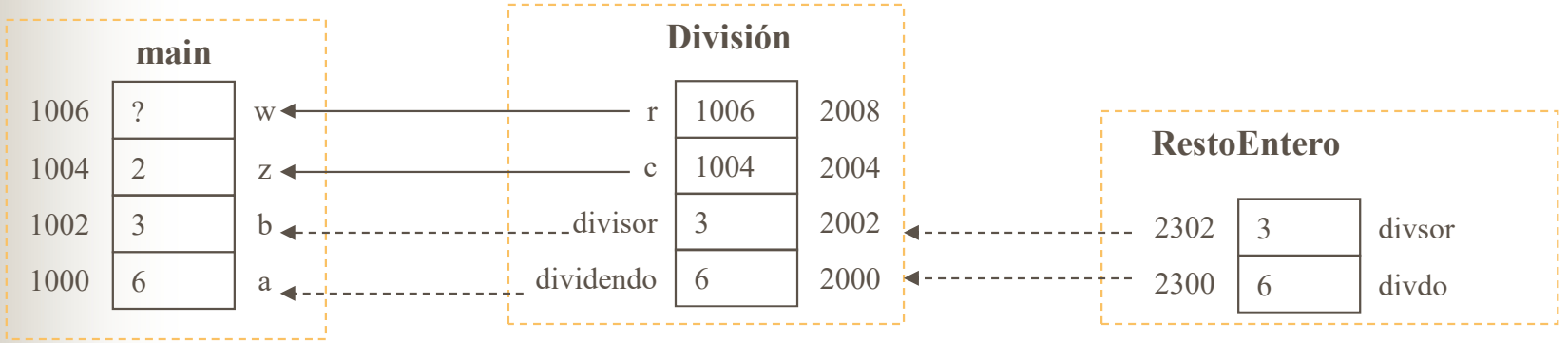


*3. *c = divdo / divsor*



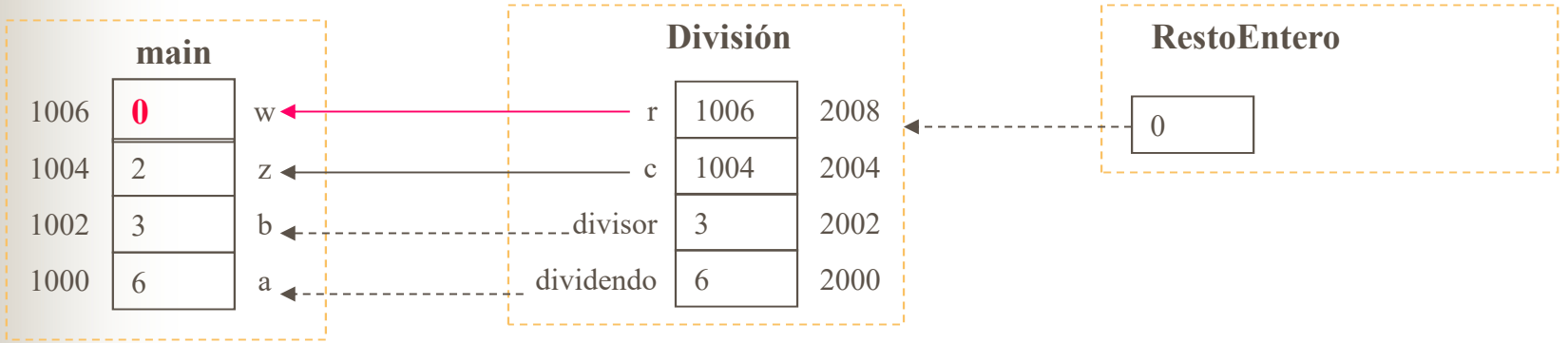
Paso de parámetros por referencia

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 *&*
- 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;
}
```

```
void main()
{
    struct punto p;
    funcion1(p);    //Paso por valor
    funcion2(&p);   //Paso por referencia
}
```

- Paso por referencia

```
void funcion2(struct punto* p)
{
   p->x=-2; //Equivale a (*p).x = -2;
   p->y=-2; //Equivale a (*p).y = -2;
}
```

Aritmética de punteros

Eva Lucrecia Gibaja Galindo
Dpto. Informática y Análisis Numérico

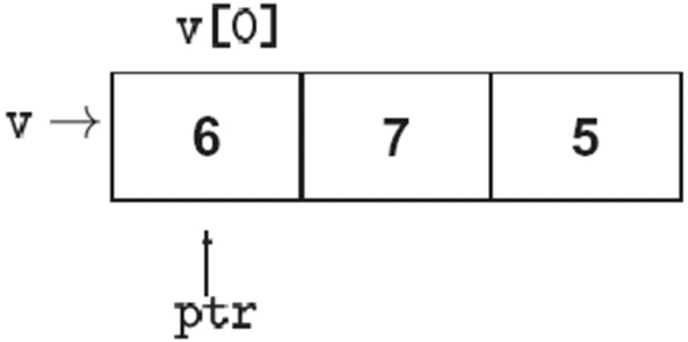
Aritmética de punteros

- Asignación (=). Han de ser del mismo tipo, en otro caso utilizar *casting*
  - p=q
- Suma/Resta con literales (enteros). Solo para vectores
  - Si *p* es un puntero a un *tipo,* la expresión ***p+desplazamiento*** devuelve un puntero a la posición de memoria ***sizeof(tipo)*desplazamiento bytes*** por encima de *p*
  - p++ //(p=p+1) //Dirección del elemento siguiente
  - p+=n //(p=p+n)
  - p=p+10
  - p-- //Dirección del elemento anterior
- Relacionales.
  - p==q, p!=q, p==NULL, p!=NULL
  - p<q, p>q, p-q → Solo para vectores
- Operaciones **NO** válidas:
  - Sumar, multiplicar o dividir punteros

Punteros y arrays unidimensionales

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

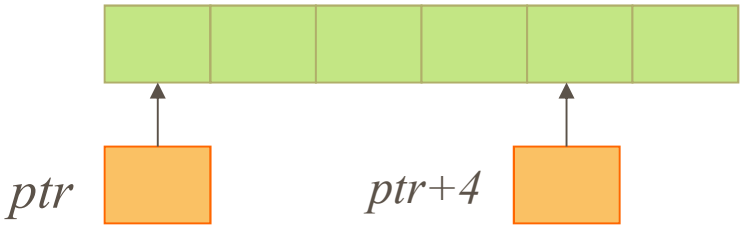
```
int v[3];
int* ptr;
...
ptr=&v[0]; //ptr=v
```



- *v* es igual a *&v[0]*
- **v equivale a v[0]*
  - *v[0]=6* equivale a **v=6* y a **(&v[0])=6*

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]

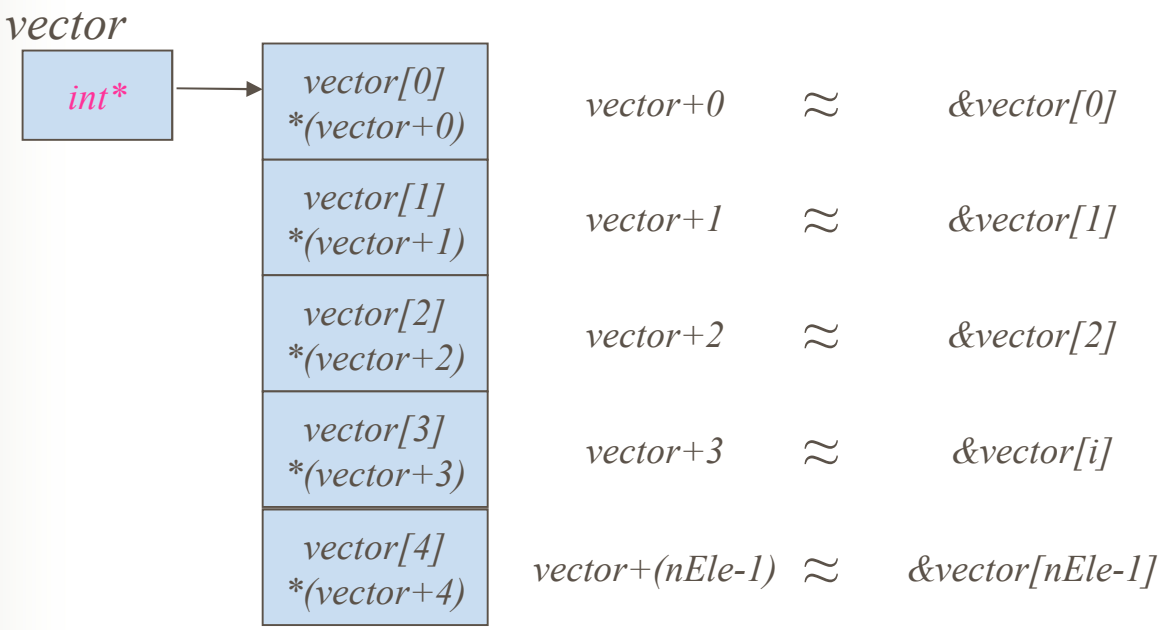


- Recíprocamente, a los punteros se les puede poner subíndices:
  - *(ptr+i) es equivalente a ptr[i]
- Pero recordad: Un vector es un puntero constante.

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

Punteros y arrays unidimensionales

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



Punteros y arrays unidimensionales

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

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

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

*primero se aplica **

luego se aplica ++

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 imprimeElementos(int vector[], int tope);`
  - **`void imprimeElementos(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

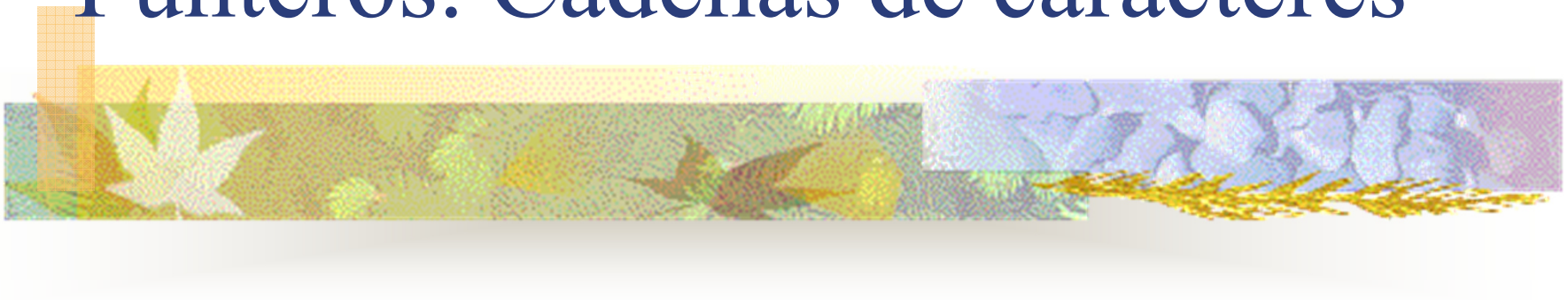
Arrays bidimensionales y funciones

- ¿Podemos generalizar lo visto a *arrays* bidimensionales?
  - **NO!!!!**
- El nombre de un *array* bidimensional es un puntero al primer elemento de un *array*, que es un *array* de punteros, no un puntero a puntero (ej. int* ptr[10] es distinto de int** ptr)
- Por tanto, con memoria estática, NO podemos hacer:
  - `void imprimeMatriz(int** mat, int nfil, int ncol);`

- Siempre que estemos con memoria estática y arrays de más de una dimensión utilizaremos la notación de corchetes

```
void imprimeMatriz(int mat[][10], int nfil, int ncol);
```

Punteros. Cadenas de caracteres



Eva Lucrecia Gibaja Galindo

Dpto. Informática y Análisis Numérico

cuentaCaracteres.c

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);
}
```

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

- **`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 apacicion 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);
```

errorCadena.c

wildPointer.c

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**
- Al salir de la función todas las variables locales desaparecen
- **No** retornar nunca un puntero que apunte a una variable declarada como local

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

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


¡NUEVO!
SPRAY
ANTIVIOLACIONES
DE SEGMENTO

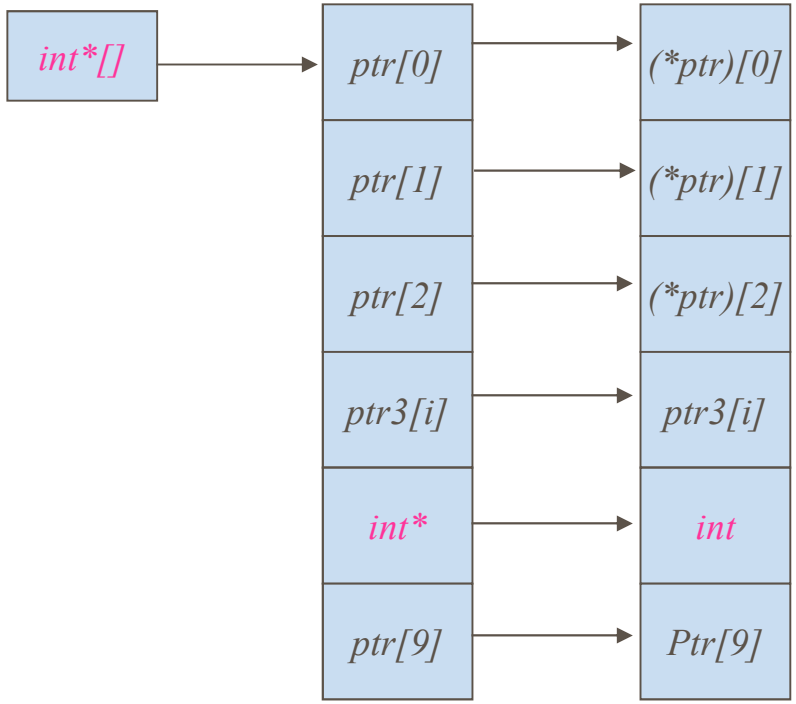

I SEE DEAD POINTERS...
TOO MANY HOURS DEBBUGING 3D C CODE...
HORN

Punteros. Complementario

Eva Lucrecia Gibaja Galindo
Dpto. Informática y Análisis Numérico

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 almacena una dirección de memoria
    - ptr[5]=&edad;



arrayaPunteros.c

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

PunterosConstantes.c

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
```

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!!