

PUNTEROS EN C

PUNTEROS

- Permiten simular el pasaje de parámetros por referencia.
- Permiten crear y manipular estructuras de datos dinámicas.
- Su manejo es de fundamental importancia para programar en C.



PUNTEROS

- Un puntero es una variable que contiene una dirección de memoria.
- Por lo general, una variable contiene un valor y un puntero a ella contiene la dirección de dicha variable.
- Es decir que la variable se refiere **directamente** a un valor mientras que el puntero lo hace **indirectamente**.

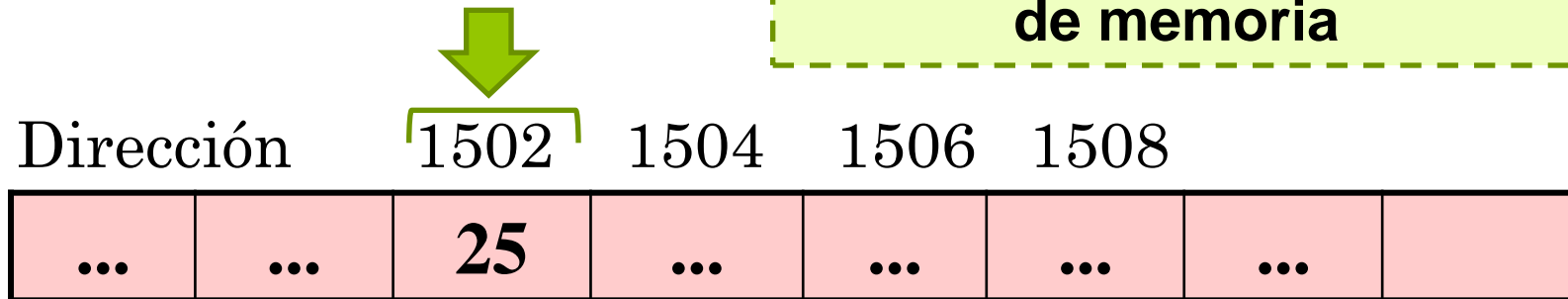


DIRECCIÓN Y CONTENIDO DE MEMORIA

- Una dirección de memoria y su contenido no es lo mismo.

```
int x = 25;
```

Un **puntero** es una variable que contiene una **dirección de memoria**



La **dirección** de la variable `x` es 1502

El **contenido** de la variable `x` es 25



DECLARACIÓN DE PUNTEROS

○ Ejemplo

```
int *countPtr, count;
```



Puntero a un
entero



Es un entero
(NO un puntero)

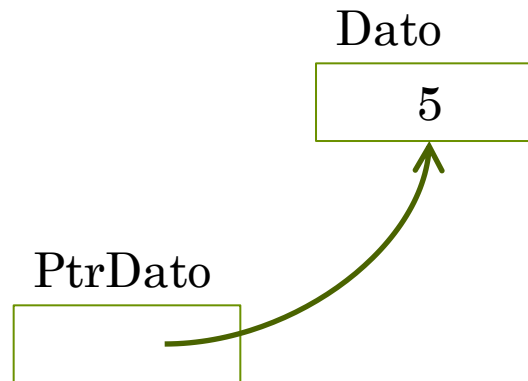
- El * no se aplica a todos los nombres de variables de una declaración. Cada puntero debe llevar su nombre precedido por *.



OPERADORES DE PUNTEROS

- El operador & u *operador de dirección*, es un operador unario que retorna la dirección de su operando.
- **Ejemplo**

```
int Dato = 5;  
int *PtrDato;  
  
PtrDato = &Dato;
```



OPERADORES DE PUNTEROS

- El operador *, también llamado *operador de indirección*, retorna el valor del objeto hacia el cual apunta su operando.
- **Ejemplo**

```
int Dato = 5, *PtrDato;
```

```
PtrDato = &Dato;
```

```
printf("%d\n", *PtrDato);
```



Imprime 5



PUNTEROS

- El puntero debe contener una dirección a un elemento del mismo tipo que la variable apuntada.

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

```
int main()
```

```
{  int *ptr;  
    int dato=30;
```



Declara un puntero a un entero

```
ptr = &dato;
```

```
*ptr = 50;
```

```
printf("Dato = %d\n", dato);
```

```
return 0;
```

```
}
```



PUNTEROS

- El puntero debe contener una dirección a un elemento del mismo tipo que la variable apuntada.

```
#include <stdio.h>
int main()
{   int *ptr;
    int dato=30;

    ptr = &dato; ←
    *ptr = 50;

    printf("Dato = %d\n", dato);

    return 0;
}
```

& es el **operador de dirección**: permite obtener la dirección de memoria de la variable que le sigue



PUNTEROS

- El puntero debe contener una dirección a un elemento del mismo tipo que la variable apuntada.

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

```
int main()
```

```
{  int *ptr;  
    int dato=30;
```



```
ptr = &dato;
```

```
*ptr = 50;
```



```
printf("Dato = %d\n", dato);
```

```
return 0;
```

```
}
```

No hay que confundir el *
que aparece en la
declaración
con
el **operador de
indirección**



PUNTEROS

- El puntero debe contener una dirección a un elemento del mismo tipo que la variable apuntada.

```
#include <stdio.h>
int main()
{   int *ptr;
    int dato=30;

    ptr = &dato;
    *ptr = 50;

    printf("Dato = %d\n", dato);

    return 0;
}
```



Cámbielo por
float * ptr

Ejecute y observe el
resultado obtenido



VISUALIZANDO EL VALOR DE UN PUNTERO

- Puede utilizarse printf con la especificación de conversión **%p** para visualizar el valor de una variable puntero en forma de entero hexadecimal.
- **Ejemplo**

```
int Dato = 5, *PtrDato;
```

```
PtrDato = &Dato;
```

```
printf("%p\n", PtrDato);
```

← Qué imprime?



VISUALIZANDO EL VALOR DE UN PUNTERO

- Puede utilizarse printf con la especificación de conversión **%p** para visualizar el valor de una variable puntero en forma de entero hexadecimal.
- **Ejemplo**

```
int Dato = 5, *PtrDato;
```

```
PtrDato = &Dato;
```

```
printf("%p\n", PtrDato);
```



0028FF1C



¿Qué imprime?

EJEMPLO

```
#include <stdio.h>
int main()
{
    int a = 34; // Declaración de variable entera de tipo entero
    int *puntero; // Declaración de variable puntero de tipo entero
    puntero = &a; // Asignación de la dirección memoria de a

    printf("El valor de a es: %d. \n"
           "El valor de *puntero es: %d. \n", a, *puntero);

    printf("La dirección de memoria de *puntero es: %p", puntero);

    return 0;
}
```

```
El valor de a es: 34.
El valor de *puntero es: 34.
La direccion de memoria de *puntero es: 000000000061FE14
```



INICIALIZACIÓN DE PUNTEROS

- Los punteros deben ser inicializados.
- Utilice el identificador **NULL** (definido en <stdio.h>) para indicar que el puntero no apunta a nada.
- El **0** es el único valor entero que puede asignarse directamente a un puntero y es equivalente a **NULL**.
- Cuando se asigna **0** a un puntero se realiza un casting previo automático al tipo apropiado.



EJEMPLO

```
#include <stdio.h>
int main()
{
    int *ptr1 = 45637325; ←
    int *ptr2 = 0;
    int *ptr3 = NULL;

    return 0;
}
```

No es posible asignarle un valor fijo a un puntero. No es posible saber si es una posición válida.



EJEMPLO

```
#include <stdio.h>
int main()
{
    int *ptr1 = 45637325;

    int *ptr2 = 0;    ←
    int *ptr3 = NULL;

    return 0;
}
```

El 0 es el único valor que puede asignarse a un puntero.
La conversión a (int *) es automática.



EJEMPLO

```
#include <stdio.h>
int main()
{
    int *ptr1 = 45637325;

    int *ptr2 = 0;

    int *ptr3 = NULL;

    return 0;
}
```

← NULL equivale a 0 y está
definido en <stdio.h>



PASAJE DE PARÁMETROS POR REFERENCIA

- Vimos que en C los parámetros de las funciones siempre se pasan por valor.
- Para simular el pasaje de parámetro por referencia se utiliza la dirección de la variable, es decir, que lo que se envía es un puntero a su valor.
- El puntero es un parámetro sólo de entrada que permite modificar el valor de la variable a la que apunta.



```
/* parámetro por referencia */
```

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

```
void cuadrado(int *);
```

```
int main()
```

```
{   int a = 5;
```

```
    printf("Valor original = %d\n", a);
```

```
    cuadrado(&a);
```



Envía la dirección de la variable (un puntero)

```
    printf("Valor al cuadrado = %d\n", a);
```

```
    return 0;
```

Recibe un puntero a un entero

```
}
```

```
void cuadrado(int * nro)
```

```
{
```

```
    *nro = *nro * *nro;
```

```
}
```

Valor de la variable apuntada por **nro**



PASAJE DE PARÁMETROS POR REFERENCIA

- Muchas de las funciones estándares de C, trabajan con punteros, como es el caso del **scanf** o **strcpy**.

```
char a;  
  
scanf("%c", &a);  
  
printf("%c\n", a);
```

```
char *a;  
  
scanf("%c", a);  
  
printf("%c\n", *a);
```

¿Compila?



RELACIÓN ENTRE PUNTEROS Y ARREGLOS

- El nombre del arreglo puede ser considerado un puntero constante.
- Los punteros pueden utilizarse para realizar cualquier tarea que involucre subíndices de arreglos.

```
int b[]={1, 2, 3, 4, 5}, *bPtr;  
  
bPtr=&b[0];  
  
printf("b[3]= %d o %d",  
       b[3], *(bPtr+3));
```



RELACIÓN ENTRE PUNTEROS Y ARREGLOS

```
int b[]={1, 2, 3, 4, 5}, *bPtr;
```

```
bPtr=&b[0];
```

```
printf("b[3]= %d o %d",  
      b[3], *(b + 3));
```

El mismo arreglo puede ser tratado como un puntero y utilizado en aritmética de punteros.



RELACIÓN ENTRE PUNTEROS Y ARREGLOS

```
int b[]={1, 2, 3, 4, 5}, *bPtr;
```

```
bPtr=&b[0];
```

```
printf("b[3]= %d o %d",  
      b[3], bPtr[3]);
```

Los punteros pueden tener subíndices
como los arreglos.



EXPRESIONES Y ARITMÉTICA DE PUNTEROS

- Los punteros son operandos válidos en expresiones: aritméticas, de asignación y de comparación.
- No todos los operadores utilizados en estas expresiones son válidos en conjunción con variables de tipo puntero.
- Veamos cuales son los operadores que pueden tener punteros como operandos y como se utilizan dichos operadores.

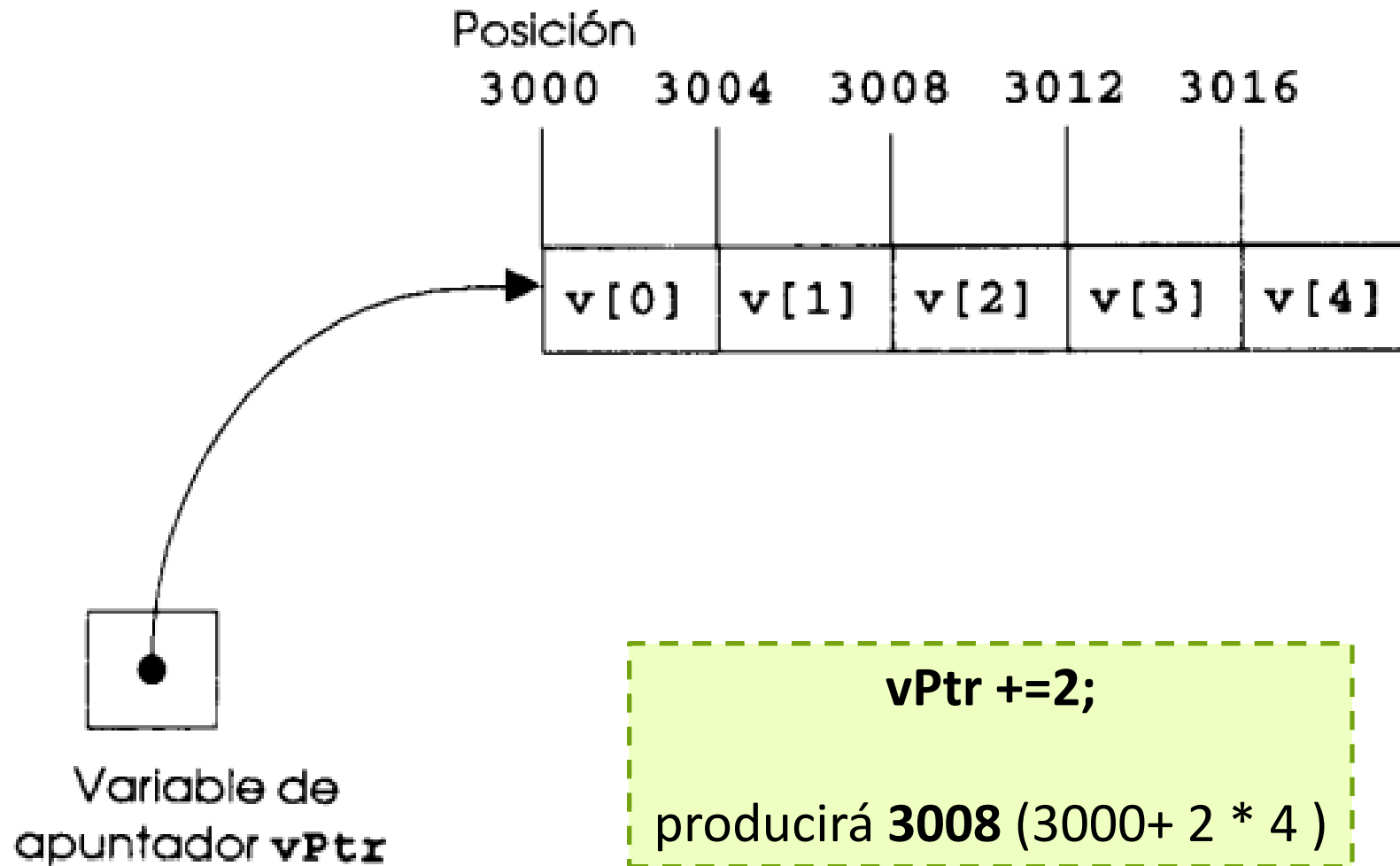


OPERACIONES ARITMÉTICAS CON PUNTEROS

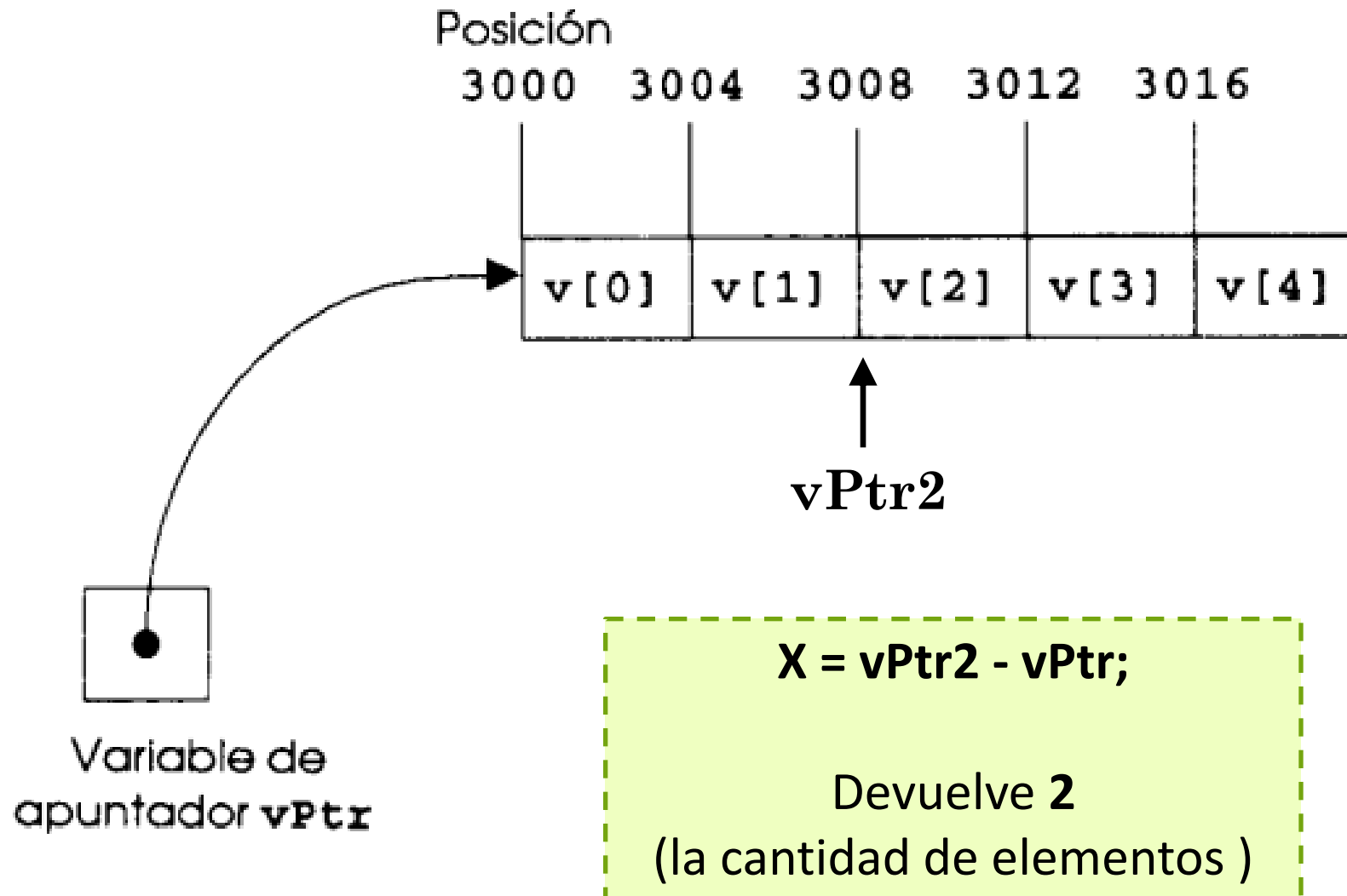
- Las operaciones aritméticas válidas sobre punteros son:
 - Incremento (++)
 - Decremento (--)
 - Añadir un entero a un puntero (+ o +=)
 - Restar un entero de un puntero (- o -=)
 - Un puntero puede ser restado de otro.
- Cuando se suma o resta un entero a un puntero, se suma dicho entero multiplicado por el tamaño del objeto al cual el puntero se refiere.



EJEMPLO: SUMA DE UN ENTERO A UN PUNTERO



EJEMPLO: RESTA DE PUNTEROS



EJEMPLO

¿Qué imprime?

```
#include <stdio.h>
int main()
{
    int x[10]={1,2,3,4},b,*pa;

    x[5]=10;
    pa=&x[5];

    b = *pa+1;
    printf("%d\n", b);

    b = *(pa+1);
    printf("%d\n", b);

    return 0;
}
```



¿Qué contiene el
vector x?

EJEMPLO

```
#include <stdio.h>
int main()
{
    int x[10]={1,2,3,4}, *pa, *pb;

    x[5]=10;
    pa=&x[5];

    pb = &x[1];

    *pb = 0;

    *pb += 2;

    (*pb) --;

    x[0] = *pb--;

    return 0;
}
```



EJEMPLO

```
#include <stdio.h>
float suma1(float [], int );
float suma2(float *, int );
#define SIZE 5
int main()
{
    float precios[SIZE] = {50.1, 10.2, 32, 10.5, 20.4};
    printf("Suma1 = %.1f\n", suma1(precios, SIZE));
    printf("Suma2 = %.1f", suma2(precios, SIZE));
    return 0;
}

float suma1(float V[], int cant)
{
    float suma2(float *P, int cant)
    {
```

Ambas funciones reciben un vector numérico y su longitud y retornan la suma de los valores del vector

Note que la invocación es la misma para ambas funciones.
Por qué?



EJEMPLO

```
#include <stdio.h>
float suma1(float [], int );
float suma2(float *, int );
#define SIZE 5
int main()
{
    float precios[SIZE] = {50.1, 10.2, 32, 10.5, 20.4};
    printf("Suma1 = %.1f\n", suma1(precios, SIZE));
    printf("Suma2 = %.1f", suma2(precios, SIZE));
    return 0;
}

float suma1(float V[], int cant)
{
    int i;
    float suma=0;
    for (i=0; i<cant; i++)
        suma += V[i];
    return(suma);
}

float suma2(float *P, int cant)
{
```

La función **Suma1**
accede al vector a
través del índice



EJEMPLO

```
#include <stdio.h>
float suma1(float [], int );
float suma2(float *, int );
#define SIZE 5
int main()
{
    float precios[SIZE] = {50.1, 10.2, 32, 10.5, 20.4};
    printf("Suma1 = %.1f\n", suma1(precios, SIZE));
    printf("Suma2 = %.1f", suma2(precios, SIZE));
    return 0;
}

float suma1(float V[], int cant)
{
    float suma2(float *P, int cant)
    {
        int i;
        float suma=0;
        for (i=0; i<cant; i++)
            suma += *P++;
        return (suma);
    }
}
```


La función **Suma2**
accede al vector a
través del puntero al
inicio



EJEMPLO

```
#include <stdio.h>
float suma1(float [], int );
float suma2(float *, int );
#define SIZE 5
int main()
{
    float precios[SIZE] = {50.1, 10.2, 32, 10.5, 20.4};
    printf("Suma1 = %.1f\n", suma1(precios, SIZE));
    printf("Suma2 = %.1f", suma2(precios, SIZE));
    return 0;
}

float suma1(float V[], int cant)
{
    float suma2(float *P, int cant)
    {
        int i;
        float suma=0;
        for (i=0; i<cant; i++)
            suma += *P++;
        return (suma);
    }
}
```



Podríamos haber usado
P[i] o bien *(P+i)
Qué diferencia hay?



ASIGNACIÓN DE PUNTEROS

- Un puntero puede ser asignado a otro si son del mismo tipo.
- Si son de tipos distintos hay que usar un operador de conversión (cast) salvo que uno de ellos sea un puntero void.
- Todos los tipos de punteros pueden ser asignados a un puntero void y un puntero void puede asignarse a cualquier tipo de puntero.
- Un puntero void no puede ser desreferenciado.



PUNTEROS VOID

- **Sintaxis**

`void * VoidPtr;`

- Un *puntero a void* es un **puntero genérico**, que puede recibir el valor de cualquier otro puntero incluso NULL
- Es decir que puede apuntar a objetos de cualquier tipo (con algunas excepciones).



```
#include <stdio.h>
int main () {
    int x = 1;
    float r = 1.0;
    void* vptr = &x; ←

    *(int *) vptr = 2;
    printf("x = %d\n", x);

    vptr = &r; ←
    *(float *) vptr = 1.1;
    printf("r = %1.1f\n", r);
}
```

Un puntero a void puede recibir el valor de cualquier tipo de puntero



```
#include <stdio.h>
int main () {
    int x = 1;
    float r = 1.0;
    void* vptr = &x;

    *(int *) vptr = 2;
    printf("x = %d\n", x);

    vptr = &r;
    *(float *) vptr = 1.1;
    printf("r = %1.1f\n", r);
}
```

Un puntero a void no puede ser desreferenciado, sin ser convertido previamente



```
#include <stdio.h>
int main () {
    int x = 1;
    float r = 1.0;
    void* vptr = &x;

    *(int *) vptr = 2;
    printf("x = %d\n", x);

    vptr = &r;
    *(float *) vptr = 1.1;
    printf("r = %1.1f\n", r);
}
```

Qué imprime?



COMPARACIÓN ENTRE PUNTEROS

- Pueden ser comparados mediante operadores de igualdad y relacionales.
- Sólo tiene sentido comparar punteros que señalan a elementos del mismo arreglo.
- La comparación se aplica a las direcciones almacenadas en dichos punteros (ej: para ver que un puntero señala a un elemento de numeración más alta en el arreglo que otro).
- Un uso común es determinar si un puntero es NULL.



EJERCICIO

- Escriba una función que reciba una cadena de caracteres y reemplace el primer blanco que encuentre en ella por un ‘\0’
- Para pasar la cadena como parámetro utilice
 - Un vector de caracteres
 - Un puntero a un char



EL CALIFICADOR **CONST** Y ARREGLOS

- La palabra clave **const** puede usarse para limitar el acceso de una función al valor del puntero o a lo apuntado por él.
- También puede aplicarse al contenido de un arreglo.
- A continuación se ejemplifican las cuatro combinaciones posibles.



```

/*  función UpperCase usando
    puntero NO constante y datos NO constantes */
#include <stdio.h>
void UpperCase(char *);
int main()
{
    char string[] = "caracteres";

    printf("El string antes de la conversión"
           " es : %s\n", string);
    UpperCase(string);
    printf("El string después de la conversión"
           " es : %s\n", string);
    return 0;
}

void UpperCase(char *s)
{
    while (*s != '\0') { /* no encontró el fin del string */
        if (*s >= 'a' && *s <= 'z')
            *s -= 32; /* convierte el ASCII a mayúscula */
        ++s; /* s apunta al próximo carácter */
    }
}

```

UpperCase_PtrNOCte_DatosNOCte.c

```
void UpperCase(char *s)
{
    while (*s != '\0')
    { /* no encontró el fin del string */
        if (*s >= 'a' && *s <= 'z')
            *s -= 32; /* convierte a mayúscula */

        ++s; /* s apunta al próximo carácter */
    }
}
```

$*s = *s - 'a' + 'A';$

$*s = *s - ('a' - 'A');$

$*s -= ('a' - 'A');$



```

#include <stdio.h>
double Promedio(const double *, double);
int main()
{
    double datos[] = {12, 10, 34};

    printf("El promedio es %5.2f\n",
           Promedio(datos, 3));

    return 0;
}

```

El valor del puntero puede cambiar pero los valores a los que apunta no.

```

double Promedio(const double * V, double cant)
{
    int i;
    double suma = 0;
    for (i=0; i<cant; i++, V++)
        suma += *V;
    return (suma / cant);
}

```

Promedio_PtrNOCte_DatosCte.c

```

#include <stdio.h>
double Promedio(const double *, double);
int main()
{
    double datos[] = {12, 10, 34};

    printf("El promedio es %5.2f\n",
           Promedio(datos, 3));

    return 0;
}

double Promedio(const double * V, double cant)
{
    int i;
    double suma = 0;
    for (i=0; i<cant; i++, V++)
        suma += *V;
    return (suma / cant);
}

```

Verifique que no es posible
modificar los valores de V.

```
#include <stdio.h>
double Promedio(const double * const, double);
int main()
{   double datos[] = {12,10,34};

    printf("El promedio es %5.2f\n ",
           Promedio(datos,3));
    return 0;
}

double Promedio(const double * const V, double cant)
{   int i;
    double suma = 0;
    for (i=0; i<cant; i++, V++)
        suma += *V;
    return(suma / cant );
}
```

Verifique que
no compila



```

1  /*Intento de modificar un dato usando
2   un puntero no cte. a un dato cte */
3  #include <stdio.h>
4  void f(const int *);
5  int main()
6  {   int y;
7
8      f(&y); /*intento de modific.illegal */
9
10     return 0;
11 }
12
13 void f(const int *x)
14 {   *x = 100; /* no se puede modificar */
15 }
16

```

Line	Message
	In function 'f':
14	error: assignment of read-only location '*x'
	=== Build finished: 1 errors, 0 warnings ===


```

1  /*Intento de modificar un
2   | puntero constante a un dato
3   | no constante */
4  #include <stdio.h>
5  int main()
6  {   int x, y;
7     | int * const ptr = &x;
8
9     | ptr = &y;
10    | return 0;
11    | }

```

El puntero es constante.
Sólo toma valor en su
declaración.

Line	Message
	In function 'main':
9	error: assignment of read-only variable 'ptr'
	=== Build finished: 1 errors, 0 warnings ===

PUNTEROS CONSTANTES

- Una declaración de puntero precedida de **const** hace que el objeto apuntado sea una constante pero el puntero puede cambiar su valor.

```
const char *p = "Taller de Leng.I";
```

```
p[0] = 't';
```



Produce un error en
compilación

```
p = "Ptr constante";
```

```
error: assignment of read-only location '*p'
```



PUNTEROS CONSTANTES

- Una declaración de puntero precedida de **const** hace que el objeto apuntado sea una constante pero el puntero puede cambiar su valor.

```
const char *p = "Taller de Leng.I";
```

```
p[0] = 't';
```

```
p = "Ptr constante";
```



Hace que el puntero señale otra dirección de memoria



PUNTEROS CONTANTES

- Para que el valor de puntero no cambie, **const** debe ubicarse inmediatamente a la izquierda de su nombre

El valor de **p** no puede cambiar

```
char * const p = "Ejemplo de ptr.";
```

```
p[0] = 'x';
```

```
p = "esto es un error";
```



PUNTEROS CONTANTES

- Para que el valor de puntero no cambie, **const** debe ubicarse inmediatamente a la izquierda de su nombre

```
char * const p = "Ejemplo de ptr.";
```

```
p[0] = 'x';
```



Esto es correcto

```
p = "esto es un error";
```



PUNTEROS CONTANTES

- Para que el valor de puntero no cambie, **const** debe ubicarse inmediatamente a la izquierda de su nombre

```
char * const p = "Ejemplo de ptr.";
```

```
p[0] = 'x';
```

```
p = "esto es un error";
```



Esto está
mal



ARREGLOS DE PUNTEROS

- Los punteros son variables, por lo tanto, es posible trabajar con arreglos de punteros.
- **Sintaxis**

```
int * PtrNros[4];
```

```
int a=1, b=2, c=3, d=4;
```

```
PtrNros[0] = &a;
```

```
PtrNros[1] = &b;
```

```
PtrNros[2] = &c;
```

```
PtrNros[3] = &d;
```



```
#include <stdio.h>
int main()
{
    int * PtrNros[4];

    int i, a=1, b=2, c=3, d=4;

    PtrNros[0] = &a;
    PtrNros[1] = &b;
    PtrNros[2] = &c;
    PtrNros[3] = &d;

    for(i=0; i<4; i++)
    {
        *(PtrNros[i]) *= 2;

        printf("%d\n", *PtrNros[i]);
    }
    return 0;
}
```

QUÉ IMPRIME?



ARREGLOS DE PUNTEROS

- Ejemplos

```
int a=1, b=2, c=3, d=4;
```

```
int * PtrNros[]={&a, &b, &c, &d};
```

```
char * Palabras[]={ "Uno", "Dos", "Tres", "Cuatro" };
```



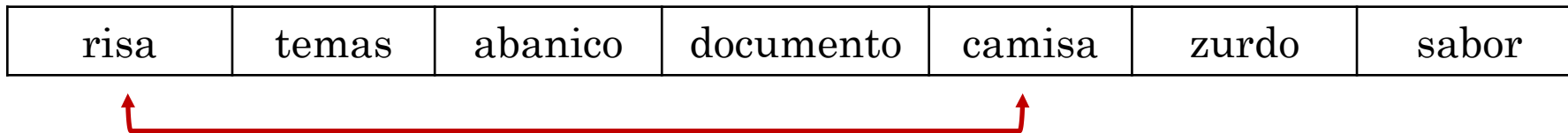
EJERCICIO

- Escriba una función que reciba un vector de palabras y su longitud y lo modifique de manera que la palabra más corta quede ubicada en la primera posición
- Ejemplo



EJERCICIO

- Escriba una función que reciba un vector de palabras y su longitud y lo modifique de manera que la palabra más corta quede ubicada en la primera posición
- Ejemplo



EJERCICIO

```
#include <stdio.h>
#include <string.h>
#define N 7
void Mostrar(char * [], int );
void MasCorta(char * [], int );

int main ()
{
    char * Palabras[N]={"camisa", "temas", "abanico", "documento", "risa", "zurdo", "sabor"};

    Mostrar(Palabras, N);

    MasCorta(Palabras, N);

    Mostrar(Palabras, N);

    return 0;
}
```



EJERCICIO

- Escriba una función que reciba un vector de palabras y lo retorne ordenado alfabéticamente.



EJERCICIO 4B.1

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

```
void Mostrar(char * P[], int N);
void Ordenar(char * P[], int N);
```

COMPLETAR

```
int main ()
{
    char * Palabras[]={"Uno", "Dos", "Tres", "Cuatro"};

    Mostrar(Palabras, 4);

    Ordenar(Palabras, 4);

    Mostrar(Palabras, 4);

    return 0;
}
```



MATRICES Y PUNTEROS

- Si la matriz se declara de la siguiente forma

int nros[5][15];

sus elementos se almacenarán en forma consecutiva por filas.

- Por lo tanto, puede accederse a sus elementos utilizando
 - **nros[filas][col]**
 - ***(nros + (15 * fila) + col)**



MATRICES Y PUNTEROS

- Una función que espera recibir como parámetro una matriz declarada de la siguiente forma

int nros[5][15];

puede utilizar cualquiera de las siguientes notaciones

function F (int M[][15], int FIL)

function F (int *M, int FIL, int COL)



EJERCICIO

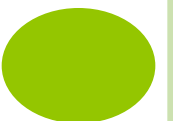
- Escriba la función **ReemplazarPares** que recibe como parámetros
 - La dirección del 1er. elemento de una matriz de enteros
 - La cantidad de filas y columnas que posee la matrizy retorna
 - La matriz con sus valores pares reemplazados por 0 (cero).



```
#include <stdio.h>
void mostrar(int[][4], int);
void ReemplazarPares(int[][4], int);
int main()
{
    int mat[3][4]={{1,2,3,4},{5,6,7,8},
                  {9,10,11,12}};

    mostrar(mat,3);
    ReemplazarPares(mat,3);
    mostrar(mat,3);

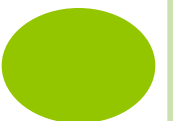
    return(0);
}
```



```
#include <stdio.h>
void mostrar(int * M, int fil, int col);
void ReemplazarPares(int *, int, int);
int main()
{
    int mat[3][4]={ {1, 2, 3, 4}, {5, 6, 7, 8},
                    {9, 10, 11, 12}};

    mostrar(mat, 3, 4);
    ReemplazarPares(mat, 3, 4);
    mostrar(mat, 3, 4);

    return(0);
}
```



EJERCICIO 4B.2

- Escriba la función **OrdenarColumnas** que recibe como parámetros
 - La dirección del 1er. elemento de una matriz de enteros
 - La cantidad de filas y columnas que posee la matrizy retorna
 - La matriz con sus columnas ordenadas en forma creciente.




```

#include <stdio.h>
void VerMatriz(int M[][5],int);
void OrdenarColumnas(int *, int, int);
int main()
{ int M[][5] = {{4,3,2,5,4},{3,2},{1,2,3,4,5}}};

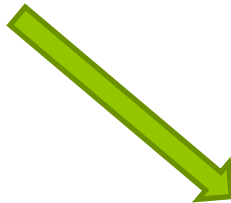
  VerMatriz(M,3);
  OrdenarColumnas(M, 3, 5);
  VerMatriz(M,3);

  return 0;
}

```



4	3	2	5	4
3	2	0	0	0
1	2	3	4	5



1	2	0	0	0
3	2	2	4	4
4	3	3	5	5

