Tema 1

Estructuras de Datos y Algoritmos

Punteros

Prof. Mary Luz Mouronte López

Universidad Francisco de Vitoria
UFV Madrid

Grado en Ingeniería Matemática
Escuela Politécnica Superior

Contenido

- **Concepto**
- **Declaración**
- **Operadores**
- **Punteros y Funciones**
- **Punteros y Vectores**
- **Asignación Dinámica de Memoria**
- **Errores Habituales al Utilizar Punteros**
- **Punteros a Estructuras Y Estructuras Auto-Referenciadas**

Concepto

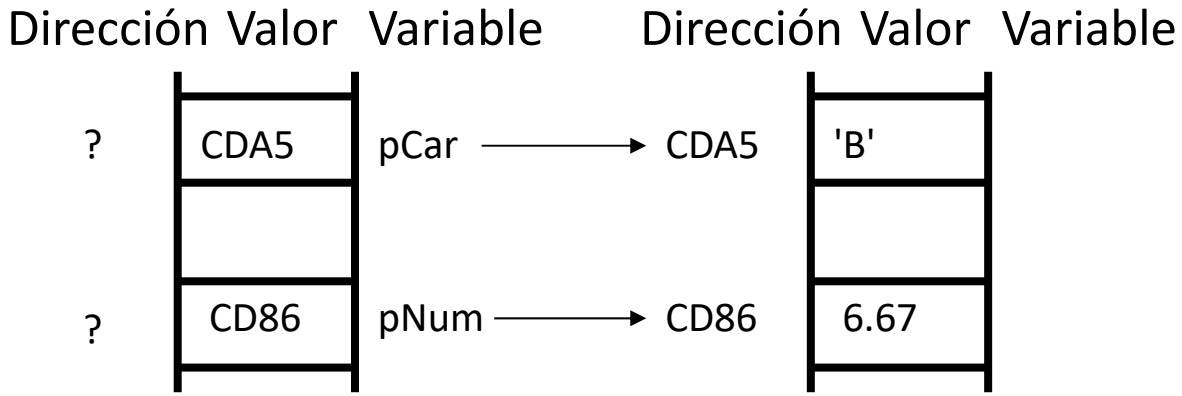
- Almacenan direcciones de memoria, en vez de valores.
- Cada posición de almacenamiento en la memoria de un ordenador, tiene una dirección de memoria.
- Cada posición de memoria almacena 1 byte.
- En lo aprendido hasta ahora, mediante la utilización de variables, se hace referencia a un dirección de memoria para conseguir su valor.
- Ahora, vamos a estudiar que mediante variables de tipo puntero, es posible manejar **direcciones de memoria** y **valores**. Este tipo de variables, tienen las siguientes ventajas:
  - Su utilización hace posible definir vectores y matrices que empleen únicamente la cantidad de memoria requerida, ajustando su tamaño dinámicamente.
  - Hacen posible implementar estructuras de datos más complejas, como por ejemplo, árboles y listas.
  - Su utilización permite que las funciones puedan devolver más de un valor.

Declaración

- **`tipo *nombre_puntero;`**
  - **nombre_puntero**: nombre de la variable de tipo puntero.
  - **tipo**: tipo de datos al que “apuntará” el puntero.
- Una variable de tipo puntero, una vez declarada, contiene un valor inicial cualquiera, es decir, apuntará a una dirección aleatoria que puede no existir.

Ejemplos:

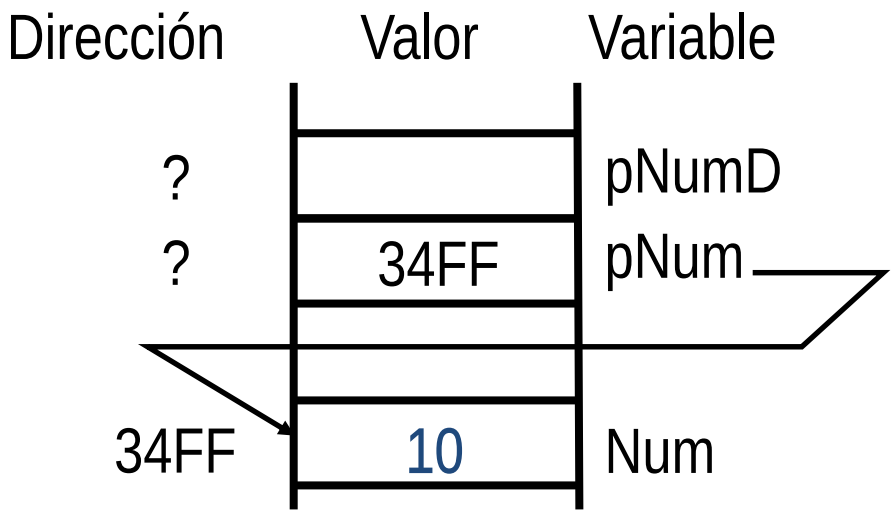
```
char *pCar; /* Variable pCar es de tipo puntero a carácter */
float *pNum;/* Variable pNum es de tipo puntero a float */
```



Operadores

- Operador **&**:
  - Operador unario de **dirección**.
  - Opera sobre cualquier tipo de variable.
  - Obtiene la dirección de memoria de su operando, es decir, la dirección de memoria de la variable sobre la que se aplica.
  - El compilador dará un aviso si se intenta realizar una asignación en la que no corresponden los tipos.

Operadores



Ejemplo:

```
int Num;
int *pNum;
double *pNumD;
...
Num = 10;
pNum = &Num;
/* se asigna la dirección de
numero a una variable puntero*/
pNumD = &Num;
/* Aviso, no se corresponden los
tipos */
...
```

Resultado: La variable Num esta en la posición (dirección) 34FF, después de la asignación a la variable puntero pNum tendrá el valor 34FF.

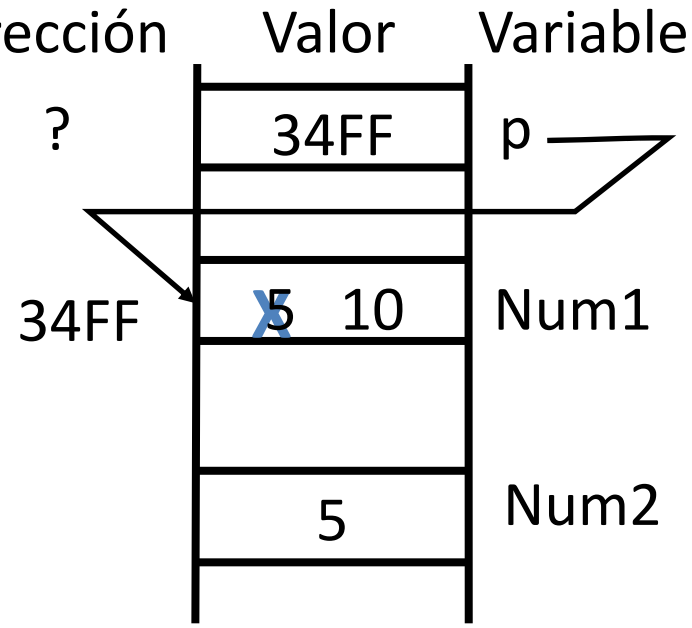
Operadores

- Operador * :
  - Operador unario de **indirección**
  - Complemento del operador &.
  - Permite operar sobre las variables a las que apunta la variable de tipo puntero.
  - Proporciona el valor de la dirección de memoria a la que apunta el puntero sobre el que se aplica, es decir, permite acceder al valor por medio del puntero.

Operadores

Ejemplo:

```
...
int Num1, Num2;
int  *p;
Num1 = 5;
p = &Num1;
Num2 = *p;
*p = 10;
...
```



Resultado: pondrá en `Num2` el valor de `Num1`, por tanto `Num2 = 5;`

ATENCIÓN: Después de la asignación `p = &Num1;` tenemos dos maneras de manipular los valores enteros almacenados en la variable `Num1`: directamente mediante `Num1` o indirectamente mediante el puntero `p`.

Operadores

- **Asignación de punteros:**
  - Es posible asignar una dirección de una variable a un puntero (ejemplo 1)
  - Es posible asignar el contenido de un puntero a otro puntero (ejemplo 2)

```
#include <stdio.h>
int main(void)
{
   int a;
   int *pNum1;
   int *pNum2;
   a = 10;
   pNum1 = &a;            /* ejemplo1 */
   pNum2 = pNum1;         /* ejemplo2 */
   printf(" El valor %d esta en la posicion %x ", *pNum2, pNum2);
   return 0;
}
```

Pregunta: ¿Cuál es el resultado?

Operadores

- **Incrementar o decrementar variables de tipo puntero.**
  - Debe entenderse como modificaciones en la dirección a la que apunta el puntero (se produce un cambio en la dirección de memoria contenida en el puntero).
  - El incremento o decremento de un puntero depende exclusivamente del **tipo de dato base** dado en su declaración.
  - En la declaración de un puntero es necesario indicar a qué tipo de dato apunta para que al utilizar los incrementos o decrementos se conozca cuánto hay que sumar o restar.
  - La operación de sumar 1 a un puntero hace que su dirección se incremente la cantidad necesaria para pasar a apuntar al siguiente dato del mismo tipo (cantidad que coincide con el número de bytes que ocupa dicho tipo de dato).
    Por lo tanto, sólo en el caso de variables que ocupan 1 byte en memoria (variables de tipo “char”) la operación de incremento aumenta en 1 la dirección de memoria; en los demás casos aumenta más.

Operadores

- **Resta de punteros:**
  - Tiene sentido si apuntan a direcciones diferentes de un mismo VECTOR.
  - El resultado de la resta es la diferencia de posiciones del VECTOR que existe entre ambos

- Es posible comparar punteros mediante cualquier operador de comparación y relacional, pero:
  - Es necesario que ambos punteros sean del mismo tipo.
  - Resultan de gran utilidad al trabajar con vectores utilizando punteros.

- **Operaciones PROHIBIDAS CON PUNTEROS**
  - Suma de punteros
  - Multiplicación de punteros
  - División de punteros

Operadores

Ejemplo 1: Incremento de un puntero

```
#include <stdio.h>
int main( void)
{
   int Num=7;
   int *pNum;
   pNum = &Num;
   printf("\nLa dirección del dato =%x",&Num);
   printf("\nEl valor que apunta el puntero = %d", *pNum);
   printf("\nEl contenido de pNum =%x",pNum);

  pNum++;

   printf("\nnel contenido de pNum =%x",pNum);
   printf("\nEl valor que apunta el puntero = %d",*pNum);
   return 0;
}
```

Pregunta: ¿Cuál es el resultado?

Resultado:

```
La dirección del dato =baffe4e4
El valor que apunta el puntero = 7
El contenido de pNum =baffe4e4

el contenido de pNum =baffe4e8
El valor que apunta el puntero = -1157634840
```

Operadores

Ejemplo 2: Incremento de un puntero

…

```
int  *pNumI;
```

Si el contenido de **pNumI** (la dirección de una variable) = 1000, podemos escribir:

pNumI	**1000**
pNumI +1	**1004**
pNumI +2	**1008**
pNumI +3	**100C**
…	

```
char *pChar;
```

Si el contenido de **pChar** (la dirección de una variable) = 1000, podemos escribir:

pChar	**1000**
pChar +1	**1001**
pChar +2	**1002**

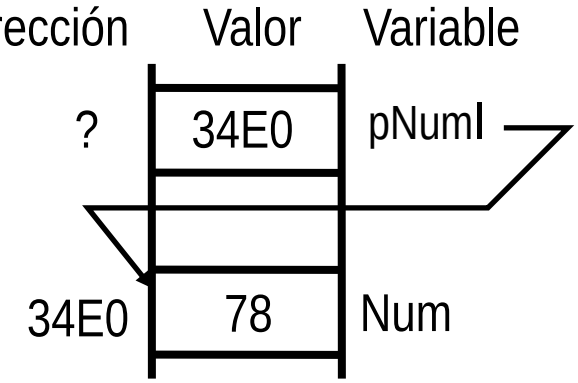
Operadores

No confundir <u>incremento o decremento del puntero</u> con el <u>incremento o decremento del valor a donde apunta el puntero</u>

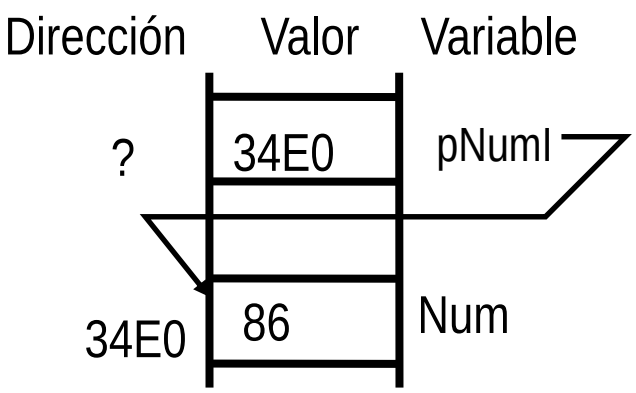
- **Ejemplo 3**: Incremento de un puntero

```
int  *pNumI;
int  Num;
...
1 Num = 78;
  pNumI = &Num;
2 *pNumI += 8;
3 pNumI += 8;
...
```

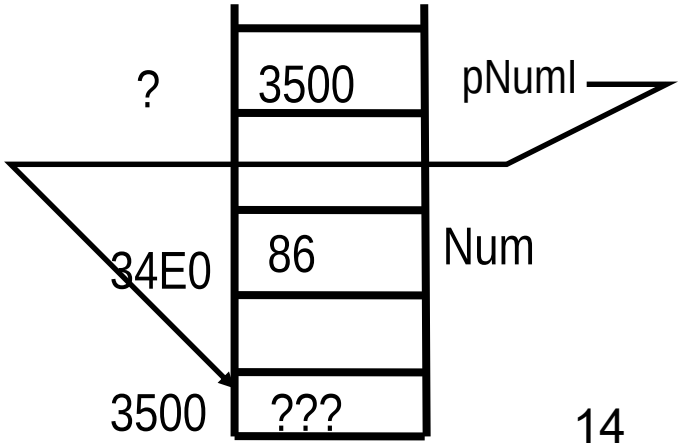
1



2 Después de `*pNumI += 8`



3 Después de `pNumI += 8`



Operadores

Ejemplo 4: Sustracción de punteros

```
#include <stdio.h>
int main(void)
{
   int vector[5] = {2,4,6,8,10};
   int  *p1, *p2;
   p1= &vector[0];
   p2 = &vector[4];
   printf("\n dirección del primer elemento =%x  ",p1);
   printf("\n dirección del ultimo elemento =%x  ",p2);
   printf("\n p2-p1 = %d", p2-p1);
   printf("\n *p2-*p1 = %d", *p2-*p1);
   return 0;
}
```

Resultado:

```
dirección del primer elemento = 8276
dirección del ultimo elemento = 8286
p2 - p1 = 4          (8286-8276) / 4 = 4  (*)
*p2 - *p1 = 8        ¡Valores hexadecimal!
```

Operadores

Ejemplo 5: comparación de punteros

```
#include <stdio.h>
int main(void)
{  int vector[5] = {2,4,6,8,10};
   int  *p1, *p2;
   p1= &vector[0];
   p2= &vector[4];
   if (p1 > p2) {
      printf("\np1 apunta a una dirección más alta que p2");
   }
   else {
      if (p1 == p2) {
        printf("\np1 y p2 apuntan al mismo elemento del vector");
        printf("\n Y es el elemento %d", *p1);
      }
   }
   return 0;
}
```

Operadores

En cada dirección de memoria se guarda 1 byte.

Si el tipo del puntero ocupa más de 1 byte al incrementar el puntero se adelantan tantas posiciones de memoria como nº de bytes ocupe el tipo de puntero.

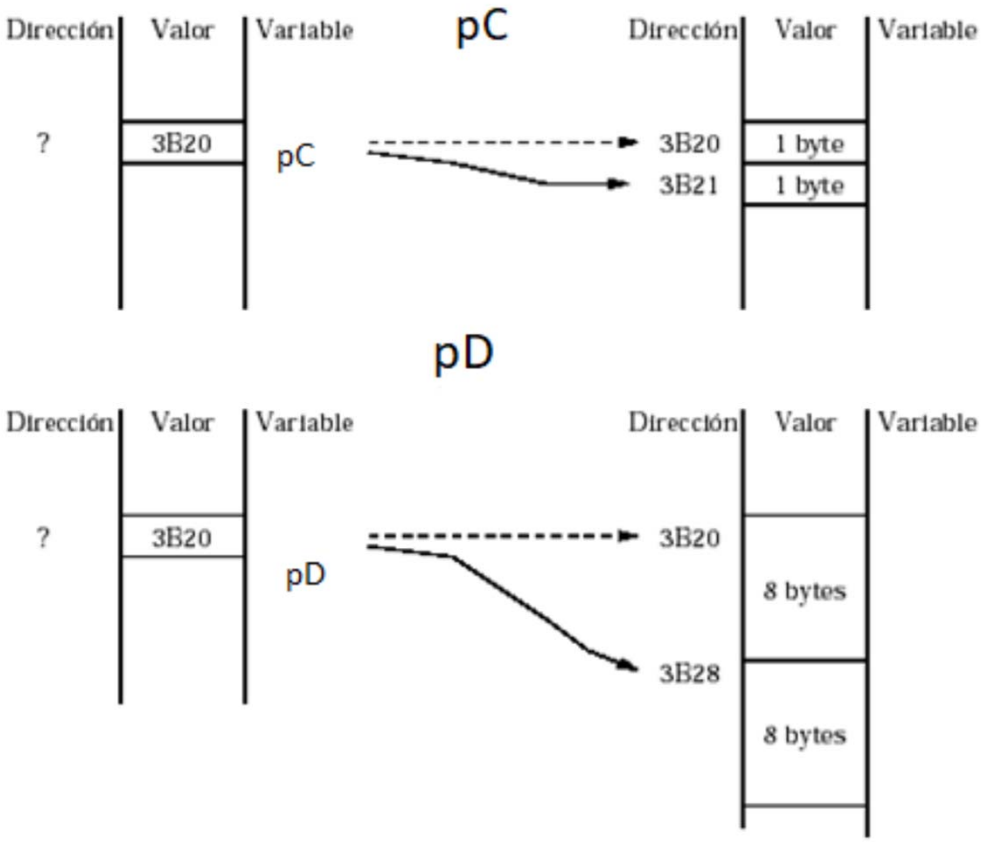
Ejemplo:

```
  char   *pC;
  double *pD;
 ...
 /*Sumamos 1 a un
 puntero a char*/

 pC++;

/*Sumamos 1 a un
 puntero a double*/

pD++;
```



Operadores - SÍNTESIS

- Asignar la dirección de una variable ordinaria a un puntero
  - `p = &variable;`
- Asignar el valor de un puntero a otro
  - `punt1 = punt2;`
- Se puede asignar un valor inicial nulo (cero) a un puntero
  - `puntero = NULL;`
- Se puede sumar o restar un valor entero a un puntero
  - `punt+2`, o `--punt.`
- Tiene sentido restar un puntero a otro si ambos apuntan a los elementos de un mismo vector.
- Se puede comparar los punteros cuando apunten al mismo tipo de dato.
- **PROHIBIDO:**
  - Sumar punteros
  - Dividir punteros
  - Multiplicar punteros

Punteros y Funciones

- Los punteros permiten realizar un **paso por referencia de parámetros** a funciones:
  - Es posible cambiar el valor de una variable en el interior de una función y conservar su valor una vez terminada la ejecución de la función.

- **Idea:**
  - En el **paso por valor**, lo único que ocurre es que se hace una copia del valor del parámetro real sobre el parámetro formal, de manera que cualquier cambio en el parámetro formal NO afecta al parámetro real.
  - En el **paso por referencia**, lo que en realidad se pasa es la dirección de la variable, de manera que los cambios, por tanto, se pueden hacer manipulando el puntero de manera que afecten directamente al contenido de la variable apuntada.

Punteros y Funciones

Ejemplo 6: Ejemplo paso por referencia:

```
#include <stdio.h>
void cambiar (int *x, int *y);
int main(void)
{
   int  a, b;
   a = 3;
   b= 5;
   printf(" \n Antes de intercambiar: a = %d, y b = %d", a, b);
   cambiar(&a, &b);    /* Se pasa la dirección de las dos variables */
   printf("\n Después de intercambio: a = %d, y b = %d", a, b);
   return 0;
}
void cambiar(int *x, int *y)
{
   int aux;
   aux = *x; /* A través de puntero x, puedo acceder al valor original */
   *x = *y;
   *y = aux;
}
```

Resultado:

```
Antes de intercambiar: a = 3, y b = 5
Después de intercambio: a = 5, y b = 3
```

Punteros y Funciones

Ejemplo 7: Paso por valor. NO SE INTERCAMBIAN. NO FUNCIONA CORRECTAMENTE.

```
#include <stdio.h>
void cambiar(int x, int y);
int main(void)
{
   int  a, b;
   a = 3;
   b= 5;
   printf(" \nAntes de intercambiar: a = %d, y b = %d", a, b);
   cambiar(a, b);
   printf("\nDespués de intercambio: a = %d, y b = %d", a, b);
    return 0;
}
void cambiar(int x, int y)
{
   int aux;
   aux = x;
        x = y;
   y = aux;
}
```

Resultado:

```
Antes de intercambiar: a = 3, y b = 5
Después de intercambio: a = 3, y b = 5
```

Universidad Francisco de Vitoria UFV Madrid

Punteros y Funciones

Ejemplo 7 (bis): Paso por referencia. FUNCIONA CORRECTAMENTE.

```
#include <stdio.h>
void cambio (int a, int *b);
int main(void)
{
    int a, b;
    a = 2;
    b = 2;
    printf("\nValores antes de hacer llamada: a = %d, y b = %d",a,b);
    cambio(a,&b);
    printf("\nValores después de hacer llamada: a = %d, y b = %d",a,b);
    return 0;
}

void cambio(int a, int *b) {
    a = 5 * a;
    *b = 5 * (*b);
}
```

Resultado:

```
Valores antes de hacer la llamada: a = 2, y b = 2
Valores después de hacer la llamada: a = 2, y b = 10
```

Punteros y Vectores

- Una de las aplicaciones habituales de los punteros es el manejo de **vectores y cadenas de caracteres**.

- Como todos los elementos de un vector se almacenan en posiciones consecutivas de memoria, es suficiente conocer la posición de memoria del primer elemento para poder recorrer todo el vector con un puntero.

Punteros y Vectores

```
#include <stdio.h>
#define N 10

int main (void)
{  double v[N];
   int  i;
   double *pNumD;

   /* Inicialización */
   for (i = 0; i < N; i++){
      v[i] = 7.8 *i;
   }

   /* Imprimir valores */
   pNumD = &v[0];
   for (i= 0;i < N; i++) {
      printf("%f\n", *pNumD);
      pNumD++;
   }
   return 0;
}
```

Inicialización de vector v[] y muestra su contenido mediante un puntero

Apunta al 1er elem del vector

(*pNumD) es de tipo **double**

pNumD pasa a puntar al siguiente elem de v

Copia de manera inversa

pV apunta al 1er elem del vector v

pW apunta al último elem del vector w

```
pV = &v[0];

pW = &w[N-1];
for (i= 0;i < N; i++) {
    *pW = *pv;
     pv++;
     pW- -;
}
```

Copio un elem

Avanzo un elem de v

Retrocedo un elem de w

EQUIVALENTE

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

```
pa = a;
```

```
a[3]=27;
```

```
*(a+3) = 27;
```

Universidad Francisco de Vitoria UFV Madrid

Punteros y Vectores

- **Ejemplo 8:** Copiar el vector **v** en el vector **w** de manera que sea su inverso. Si **v** vale [1, 2, 3, 4], después del programa **w** vale [4, 3, 2, 1]

```
#include <stdio.h>
#define N 4
int main(void)
{  int v[N] = {1,2,3,4};    /* Vector */
   int w[N];
   int *pv;
   int *pw;


   /* Copia de manera inversa */
   pv = &v[0];         /* pv apunta al primer elemento del vector v */
   pw = &w[N-1];       /* pw apunta al último elemento del vector w */
   for (i= 0;i < N; i++) {
      *pw = *pv;       /*  Copio un elemento */
      pv++;            /* Avanzo un elemento */
      pw- -;           /* Retrocedo un elemento */
   }
   return 0;
}
```

Punteros y Vectores

- Equivalencia de punteros y vectores
  - Cuando se define un vector: `double a[N];` lo que ocurre es que se reserva espacio en memoria para almacenar `N` elementos del tipo `double` y se crea un puntero constante llamado a que apunta al principio del bloque de memoria reservada.
    - **Puntero constante**: El programa no puede cambiar la dirección almacenada en él.

    Por lo tanto, para hacer que el puntero pd apunte al principio del vector a se puede hacer de cualquiera de las dos maneras:

    1. `pd = &a[0];`
    2. `pd = a;`

  - El **operador [ ]**

    Cuando se accede a un elemento de un vector con `a[3] = 2.7;`

    el programa toma el puntero constante a, le suma el valor que hay escrito entre los corchetes (según el tamaño de cada tipo de dato), y escribe en dicha dirección el valor `2.7`.

    Por tanto, es equivalente a: `*(a+3) = 2.7;`

Punteros y Vectores

- **Ejemplo 9:** Se tiene un vector (v) de 100 elementos y se quiere asignar el valor 45 al 5º elemento. Se puede realizar de varias maneras:

(a)
```
v[4] = 45;
```

(b)
```
p = v;
p += 4;   /* Mover el puntero hasta la posición 5 */
*p =45;   /* Asignarle ese valor */
p -= 4;   /* Devolver p al principio del vector v */
```

(c)
```
p = v;
*(p+4) = 45;   /* Se calcula la dirección directamente y se asigna*/
```

(d)
```
p = v;
p[4] = 45;
```

(e)
```
*(v+4) = 45;
```

Punteros y Vectores
Paso de vectores a funciones

- **Paso de VECTORES a funciones**:
  - El nombre de un vector contiene la dirección del primer elemento del vector. Por este motivo no se utiliza el **&** en el parámetro real al realizar una llamada a una función.
  - El parámetro formal puede especificarse con dos notaciones:
    1. Un vector con tamaño no especificado (**`int m[ ]`**)
    2. Un puntero al primer elemento del vector (**`int *m`**)

  - No olvidar que en un vector: `int v[5];`
    - El **valor** del primer elemento = (`v[0]`)
    - La **dirección** del primer elemento = (`&v[0]`)
      ó con el nombre del vector (`v`)
    - En general, la dirección del elemento (i+1) es `&v[i]` ó `(v+i)`
      - Cuando se indica `(v+1)` hay que tener en cuenta que el desplazamiento del puntero depende del tipo base
    - El **contenido** del elemento (i+1): `v[i]` ó `*(v+i)`

Punteros y Funciones

```
#include <stdio.h>
#define N 5
void escribir (int m[ ]);  /* Opción 1 */
void escribir (int *m);    /* Opción 2 */
int main(void )
{
   int v[N]
   int i;
   for(i = 0; i<N ; ++i) {
      v[i] = i;
   }
   escribir(v);
   return 0;
}
void escribir(int m[] ) ó void escribir (int *m) /* Según la opción */
{
      int cont;
      for( cont = 0; cont<N; cont++){
         printf("%d ", m[cont]);
      }
}
```

Resultado:

```
0 1 2 3 4
```

Punteros y Funciones

```
#include <stdio.h>
#include <ctype.h>
void CalcularNumVocales (char *cadena, int *v);
int main( void)
{
    char cadena[80];
    int vocales;
    printf("\nIntroduzca una linea de texto:");
    gets(cadena);
    CalcularNumVocales (cadena, &vocales);
    printf("\nEl nº de vocales: %d", vocales);
    return 0;
}
void CalcularNumVocales(char *cadena, int *v)
{
   char c;
   int cont = 0;
   *v = 0;
   while ( cadena[cont] != '\0') {
      c = toupper(cadena[cont]));
      if (c == 'A' || c =='E' || c == 'I' || c =='O' || c =='U')
          (*v)++;
      cont++;
    }
}
```

Punteros y Funciones

```
#include <stdio.h>
#define N 5
int main(void)
{
   int  v[N];
   int i;
   for(i= 0;i<N; i++) {
      printf("elemento %d=",i+1);
      scanf("%d",&v[i]);
   }
   for(i = 0; i<N; ++i) {
      printf("elemento %d =%d, dirección =%x",i+1,*(v +i),(v+ i));
      printf("\n");
   }
   return 0;
}
```

¿Cuál es el resultado?

Resultado:

```
elemento 1=1
elemento 2=2
elemento 3=3
elemento 4=4
elemento 5=5
elemento 1 =1, dirección =32ea03b0
elemento 2 =2, dirección =32ea03b4
elemento 3 =3, dirección =32ea03b8
elemento 4 =4, dirección =32ea03bc
elemento 5 =5, dirección =32ea03c0
```

Asignación Dinámica de Memoria

- Se utiliza esta técnica cuando se desconoce hasta el momento de la ejecución la cantidad de memoria necesaria para almacenar datos.
- Por lo tanto, la asignación dinámica de memoria consiste en que, **una vez que se está ejecutando el programa** y se conoce la cantidad de memoria que se necesita, se realiza una llamada al **sistema operativo** (SO) para solicitarle un bloque de memoria libre del tamaño adecuado.
  - Si queda memoria, el sistema operativo devuelve un **puntero** que apunta al comienzo de dicho bloque.
  - Este puntero permite acceder a la memoria asignada tal y como se desee, siempre y cuando, **no se salga de los límites del bloque.**
- Una vez que se termina de usar la memoria, ésta debe **liberarse** al sistema operativo.

Asignación Dinámica de Memoria

- Funciones para trabajar con asignación dinámica de memoria:
  - **Solicitar** al SO un bloque de memoria: **`calloc()`** y **`malloc()`**
  - **Liberar** la memoria asignada dinámicamente: **`free()`**
  - Están en el archivo de cabecera "`stdlib.h`" en la librería estándar

- **Solicitar memoria:**

```
void *calloc (size_t numero_elementos, size_t tamaño_elemento);
void *malloc (size_t tamaño_bloque);
```

  - Ambas funciones solicitan al sistema operativo un bloque de memoria de un tamaño dado.
  - Ambas funciones devuelven un puntero al principio del bloque solicitado o `NULL` si no hay suficiente memoria.
  - Se debe verificar, cuando se solicite memoria al sistema operativo, que éste nos devuelve un puntero válido y no `NULL` para indicarnos que no tiene memoria disponible.

Asignación Dinámica de Memoria

- **Solicitar memoria:**

`void *calloc (size_t` *`numero_elementos`*`, size_t` *`tamaño_elemento`*`);`

  - Función **calloc()** reserva un bloque de memoria para un *`numero_elementos`* de *`tamaño_elemento`*
  - **Inicializa con ceros el bloque de memoria**
  - Devuelve el puntero genérico (*`void *`*) que apunta al principio del bloque o *`NULL`* en caso de que no exista suficiente memoria libre ó el número de elementos sea un valor negativo. Para un número de elementos 0, no devuelve *`NULL`*.

Ejemplo: Crea un vector de n enteros

```
int *pI;
int n;
...    /* Petición del valor n con la función scanf() */
pI = (int *) calloc (n, sizeof(int));
if (pI == NULL) {
   printf("Error: No hay suficiente memoria ");
}
else {
  ...
  /* Si hay memoria el programa continuaría y podría usar el
  vector recién creado */
   pI[0] = 23;
}
...
```

Asignación Dinámica de Memoria

- **Solicitar memoria:**

```
void *malloc (size_t tamaño_bloque);
```

  - Función **malloc()** reserva un bloque de memoria de tamaño *tamaño_bloque* (medido en bytes)
  - Devuelve el puntero genérico (*void **) que apunta al principio del bloque o *NULL* en caso de que no exista suficiente memoria libre.
  - En este caso **NO se inicializa el bloque con 0**.

Ejemplo: Crea un vector de n enteros,

```
int *pI;
int n;
...    /* Petición del valor n con la función scanf() */
pI = (int *) malloc (n * sizeof(int));
if (pI == NULL) {
   printf("Error: No hay suficiente memoria ");
}
else {
   …
  /* Si hay memoria el programa continuaría y podría usar el
  vector recién creado */
   pI[0] =23;
}
```

Asignación Dinámica de Memoria

- **Solicitar memoria:**
  - Ambas funciones devuelven un puntero genérico (`void *`)
  - Este puntero genérico ha de convertirse mediante una conversión de tipos (“*cast”)* al tipo de dato que se va a introducir en el bloque.
    - En los ejemplos anteriores puede verse el “casting”:

```
pI = (int *) calloc (n, sizeof(int));
```

  - En ambas funciones se utiliza el operador **sizeof(int)** para especificar la cantidad de memoria que se necesita, lo que es muy importante de cara a la portabilidad del programa.
  - Existe otra función para reasignar memoria cuando se comprueba que con la memoria que se había reservado de manera dinámica no es suficiente. Esta función se llama **realloc()**.

Asignación Dinámica de Memoria

- **Liberar memoria:**

  `void free (void *puntero_al_bloque);`

  – Una vez que se ha terminado de usar la memoria es necesario liberarla para que quede disponible para los demás programas.

  – Esta función libera la memoria **previamente asignada** mediante las funciones `calloc()` o `malloc()`.

  – La función libera el bloque a cuyo principio apunta el *`puntero_al_bloque`*.

  – Es importante que el *`puntero_al_bloque`* apunte exactamente al principio del bloque (debe ser el puntero devuelto por `calloc()` o `malloc()`).

  Ejemplo: Liberar la memoria previamente asignada

```
int *pI;
int n;
... /* Petición del valor n con la función scanf() */
pI = (int *) calloc (n, sizeof(int));
/* Comprobación */
...
/* Liberar memoria solo en el caso de asignación correcta*/
free(pI);
```

Errores Habituales al Utilizar Punteros

- Utilizar un puntero antes de que se le haya asignado memoria.

```
int *pI;
*pI = 3;
```

- Acceder a una posición de memoria que no nos pertenece

```
int *pI;
int mat[N];
mat[N] = 34;  /* Error, el último elemento del vector es el N-1 */
pI = (int *) calloc (N , sizeof(int));
if (pI == NULL) .....
else{
   *(pI+N) = 34, /* Error, la última posición donde apunta pI es
  N-1 */
}
```

- Solicitar memoria y no comprobar si se ha realizado correctamente la asignación

```
double *pI;
pI = (double *) calloc (n, sizeof(double));
pI[0] = 12.76; /* Error si ocurre que el puntero devuelto por calloc es NULL */
```

Errores Habituales al Utilizar Punteros

- Liberar memoria sin tener el puntero apuntando al principio del bloque

```
int *pI;
pI = (int *) calloc (n, sizeof(int));
if (pI == NULL){
    ....
}
else {
   pI ++; /* se modifica el puntero original */
   free(pI); /* El programa aborta pues pI no apunta al principio del bloque */
}
```

- Seguir usando la memoria una vez liberada

```
int *pI;
pI = (int *) calloc (n, sizeof(int));
if (pI == NULL)
    ....
free(pI);
a = pI[0];
/* El programa aborta pues puntero apunta a un bloque que ya  no nos pertenece */
```

Punteros a Estructuras Y Estructuras Auto-Referenciadas

- Una estructura es representada por un bloque de memoria que ha sido dividido en sub-bloques. Cada sub-bloque posee un tamaño, un tipo y un nombre por el que podemos referirnos a él.

```
struct tLibro {
   char *Titulo;
   char *Autor;
   int Paginas;
   int Precio;
};


struct tLibro Biblioteca[1000];
```

- Es posible agrupar distintos tipos de datos (en este ejemplo, dos enteros y dos punteros a carácter) relacionados lógicamente entre sí. La declaración anterior define un nuevo tipo de dato, de nombre tLibro, de tal forma que pueden declararse variables de ese tipo:

```
struct tLibro {
   char *Titulo;
   char *Autor;
   int Paginas;
   int Precio;
} Libro, Biblioteca[1000];
```

Punteros a Estructuras Y Estructuras Auto-Referenciadas

- Acceso a campos de la estructura se hace mediante operador punto '**.**' Ejemplo:

```
Libro.paginas=372;
Libro.precio=24;
Libro.Titulo=(char *)malloc(50*sizeof(char));
strcpy(Libro.Titulo,"Aurelius Augustinus");
```

- Esto abre la interesante posibilidad de que una estructura de un tipo contenga punteros a estructuras del mismo tipo, lo que **nos permite crear listas**. Por ejemplo:

```
struct tLibro{
    struct tLibro *Siguiente;
    char *Titulo;
    char *Autor;
    int precio;
    int paginas;
} *Lista;
```

Punteros a Estructuras Y Estructuras Auto-Referenciadas

- Lista es un puntero (al que no se ha asignado ningún valor, y por tanto no puede aún utilizarse) a una estructura que contiene, además de otros datos, un puntero a otra estructura del mismo tipo. Así:

```
Lista=(struct tLibro *)malloc(sizeof(struct tLibro));
```

- Para acceder a los campos de estructuras apuntadas puede utilizarse el operador '**->**':

```
Lista->Precio=20;
Lista->Paginas=400;
Lista->Autor=(char *)malloc(50*sizeof(char));

(*Lista).Precio=20;
(*Lista).Paginas=400;
(*Lista).Autor=(char *)malloc(50*sizeof(char));
```

Punteros a Estructuras Y Estructuras Auto-Referenciadas

- Si después de haber almacenado el primer libro es preciso almacenar otro, creamos un nuevo nodo:

```
Lista->Siguiente=(struct tLibro *)malloc(sizeof(struct tLibro));
```

Derechos de Autor