Punteros

Definición

Un puntero es un dato que contiene una dirección de memoria.

NOTA: Existe una dirección especial que se representa por medio de la constante NULL (definida en <stdlib.h>) y se emplea cuando queremos indicar que un puntero no apunta a ninguna dirección.

Declaración

```
<tipo> *<identificador>
```

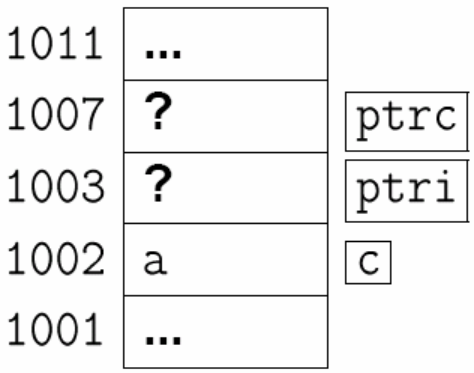
`<tipo>`
Tipo de dato del objeto referenciado por el puntero

`<identificador>`
Identificador de la variable de tipo puntero.

Cuando se declara un puntero se reserva memoria para albergar una dirección de memoria, pero NO PARA ALMACENAR EL DATO AL QUE APUNTA EL PUNTERO.

El espacio de memoria reservado para almacenar un puntero es el mismo independientemente del tipo de dato al que apunte: el espacio que ocupa una dirección de memoria.

```
char c = 'a';

char *ptrc;

int *ptri;
```



Operaciones básicas con punteros

Dirección

Operador &

`&<id>` devuelve la dirección de memoria donde comienza la variable `<id>`.

El operador & se utiliza para asignar valores a datos de tipo puntero:

```
int i;
int *ptr;
...
ptr = &i;
```

Indirección

Operador *

`*<ptr>` devuelve el contenido del objeto referenciado por el puntero `<ptr>`.

El operador * se usa para acceder a los objetos a los que apunta un puntero:

```
char c;
char *ptr;
…
ptr = &c;
*ptr = ‘A’;      // Equivale a escribir: c = ‘A’
```

Asignación

Operador =

A un puntero se le puede asignar una dirección de memoria concreta, la dirección de una variable o el contenido de otro puntero.

Una dirección de memoria concreta:

```
int *ptr;
...
ptr = 0x1F3CE00A;
...
ptr = NULL;
```

La dirección de una variable del tipo al que apunta el puntero:

```
char c;
char *ptr;
...
ptr = &c;
```

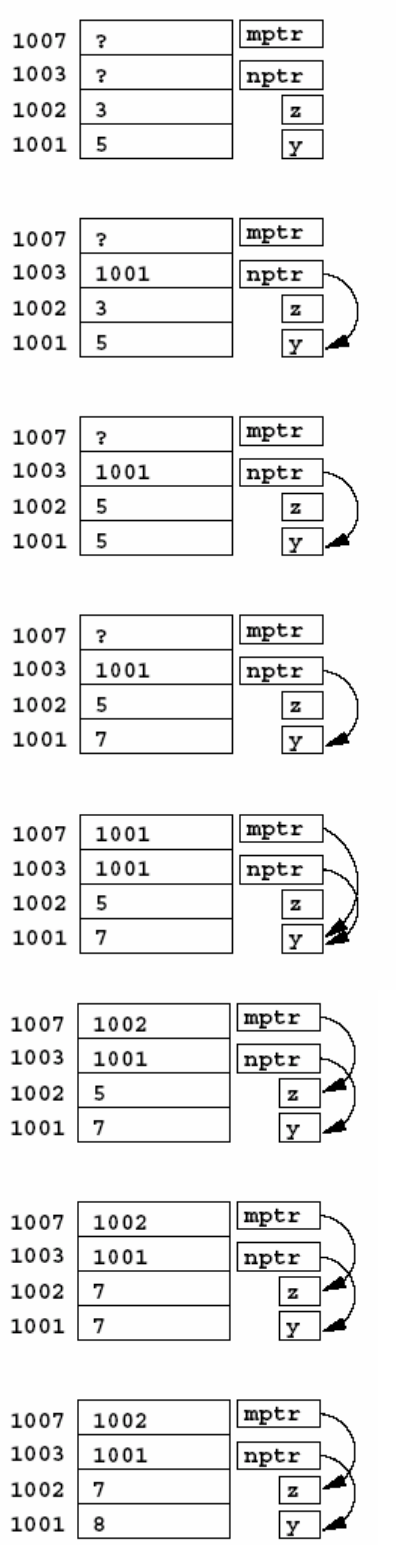
Otro puntero del mismo tipo:

```
char c;
char *ptr1;
char *ptr2;
…
ptr1 = &c;
ptr2 = ptr1;
```

Como todas las variables, los punteros también contienen “basura” cuando se declaran, por lo que es una buena costumbre inicializarlos con NULL.

Ejemplo

```
int main ()
{
  int y = 5;
  int z = 3;
  int *nptr;
  int *mptr;


  nptr = &y;




  z = *nptr;




  *nptr = 7;


  mptr = nptr;




  mptr = *z;


  *mptr = *nptr;


  y = (*nptr) + 1;

  return 0;
}
```



Errores comunes

Asignar punteros de distinto tipo

```
int a = 10;
int *ptri = NULL;
double x = 5.0;
double *ptrf = NULL;
...
ptri = &a;
ptrf = &x;
ptrf = ptri;     // ERROR
```

Utilizar punteros no inicializados

```
char *ptr;

*ptr = ‘a’;      // ERROR
```

Asignar valores a un puntero y no a la variable a la que apunta

```
int n;
int *ptr = &n;

ptr = 9;         // ERROR
```

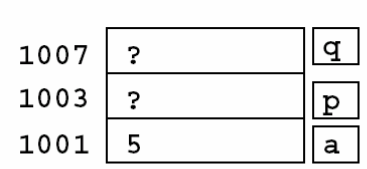
Intentar asignarle un valor al dato apuntado por un puntero cuando éste es NULL

```
int *ptr = NULL;

*ptr = 9;        // ERROR
```

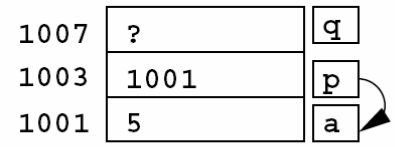
Punteros a punteros

Un puntero a puntero es…
un puntero que contiene la dirección de memoria de otro puntero-

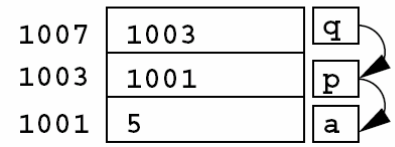
```
int main ()
{
  int a = 5;
  int *p;    // Puntero a entero
  int **q;   // Puntero a puntero
```




```
  p = &a;
```




```
  q = &p;
}
```



Para acceder al valor de la variable a podemos escribir

`a` (forma habitual)

`*p` (a través del puntero `p`)

`**q` (a través del puntero a puntero `q`)

`q` contiene la dirección de `p`, que contiene la dirección de `a`

Aritmética de punteros
Correspondencia entre punteros y vectores

Cuando declaramos un vector

```
<tipo> <identificador> [<dim>]
```

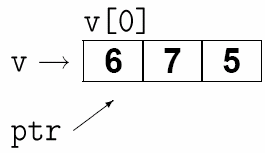
en realidad

1. Reservamos memoria para almacenar `<dim>` elementos de tipo `<tipo>`.

2. Creamos un puntero `<identificador>` que apunta a la primera posición de la memoria reservada para almacenar los componentes del vector.

Por tanto, **el identificador del vector es un puntero.**

```
int v[3];
int *ptr;
...
ptr = v;  // Equivale a ptr = &v[0]
v[0] = 6; // ≡ *v = 6; ≡  *(&v[0]) = 6;
```



Aritmética de punteros

```
<tipo> *ptr;
```

`ptr + <desplazamiento>` devuelve un puntero a la posición de memoria `sizeof(<tipo>)*<desplazamiento>` bytes por encima de `ptr`.

```
int v[];
int *ptr = v;    ➔   ptr+i  apunta a  v[i]

                     *(ptr+i) ≡ v[i]
```

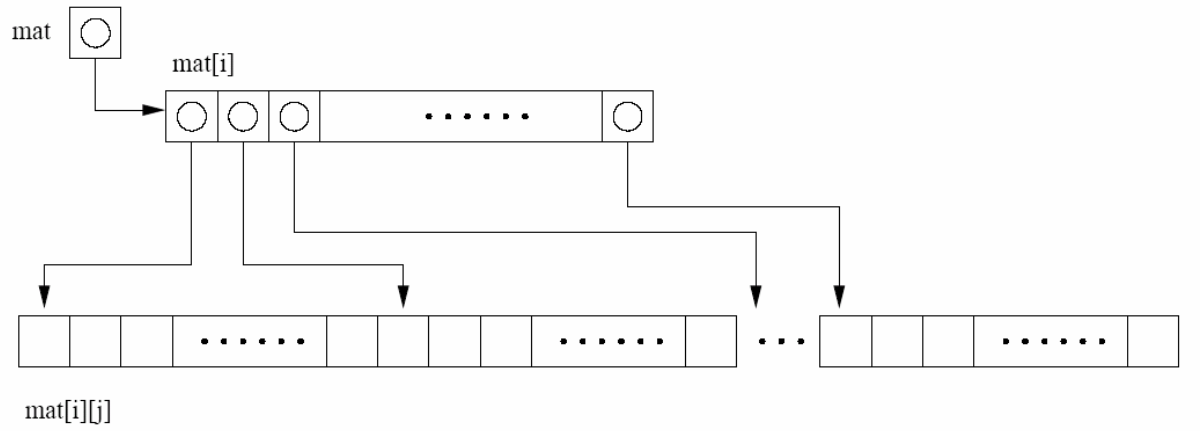
NOTA: La suma de punteros no tiene sentido y no está permitida. La resta sólo tiene sentido cuando ambos apuntan al mismo vector y nos da la “distancia” entre las posiciones del vector (en número de elementos).

Ejemplo: Distintas formas de sumar los elementos de un vector

```
int suma ( int v[], int N)
{
  int i, suma;
  int *ptr, *ptrfin;

                                /* Alternativa 1 */
  suma = 0;
  for (i=0 ; i<N ; i++)
      suma = suma + v[i];

                                /* Alternativa 2 */
  suma = 0;
  for (i=0 ; i<N ; i++)
      suma = suma + (*(v+i));

                                /* Alternativa 3 */
  suma = 0;
  ptrfin = ptr + N-1;
  for (ptr=v ; ptr<=ptrfin ; ptr++)
       suma = suma + *ptr;

  return suma;
}
```

Punteros y matrices

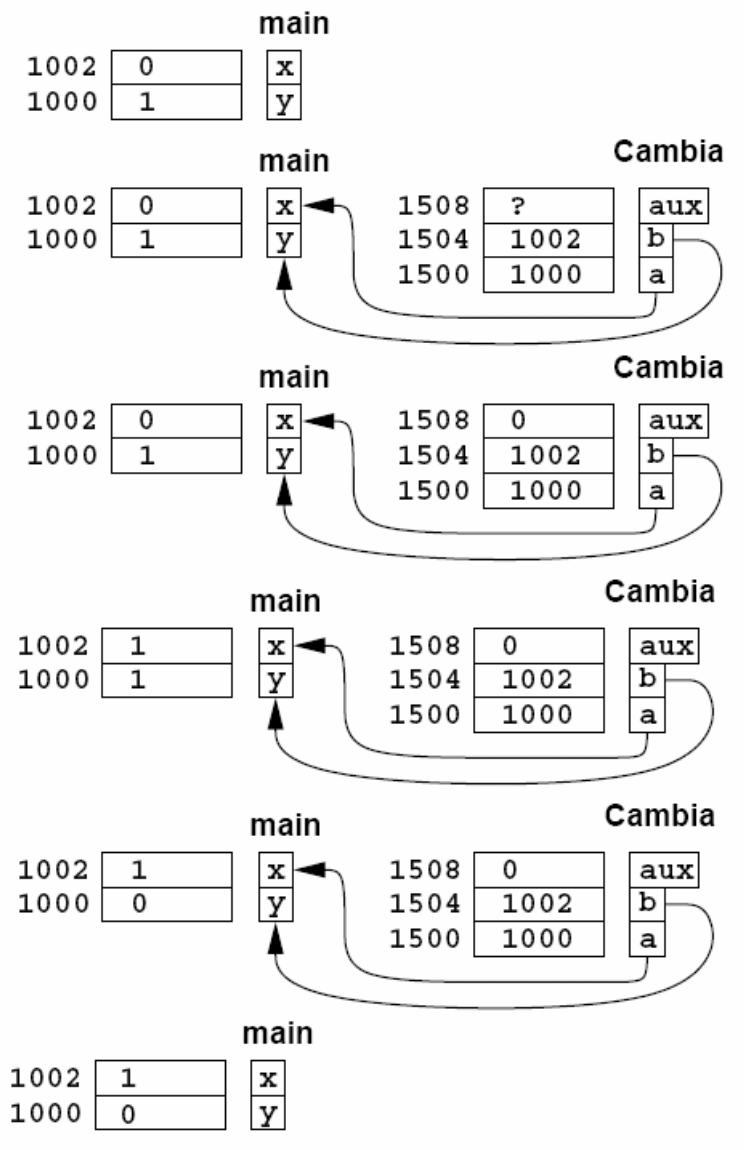
```
<tipo> mat [<dimF>][<dimC>];
```



dirección(i,j) = dirección(0,0) + i*dimC + j

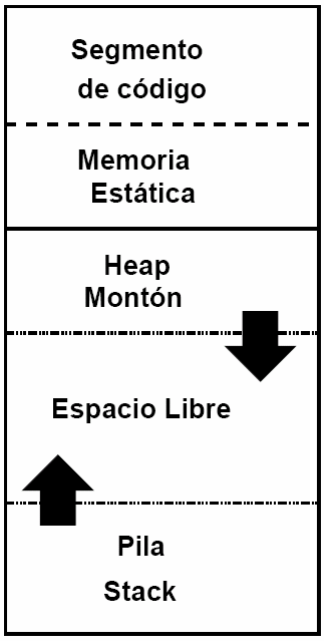
Ejemplo: Intercambio de valores

```
void Cambia(int *a, int *b)
{
  int aux;

  aux = *a;
  *a = *b;
  *b = aux;
}

int main()
{
  int x=0, y=1;
  …
  Cambia(&x,&y);
  …
  return 0;
}
```



Gestión dinámica de la memoria

Organización de la memoria



- Segmento de código (código del programa).
- Memoria estática (variables globales y estáticas).
- Pila (stack): Variables automáticas (locales).
- Heap ("montón"): Variables dinámicas.

Reserva y liberación de memoria

Cuando se quiere utilizar el heap, primero hay que reservar la memoria que se desea ocupar:

ANSI C:	Función `malloc`
C++:	Operador `new`

Al reservar memoria, puede que no quede espacio libre suficiente, por lo que hemos de comprobar que no se haya producido un fallo de memoria (esto es, ver si la dirección de memoria devuelta es distinta de NULL).

Tras utilizar la memoria reservada dinámicamente, hay que liberar el espacio reservado:

ANSI C:	Función `free`
C++:	Operadore `delete`

Si se nos olvida liberar la memoria, ese espacio de memoria nunca lo podremos volver a utilizar…

Ejemplo: Vector de tamaño dinámico

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

float media (float v[], int n)
{
    int   i;
    float suma = 0;

    for (i=0; i<n; i++)
        suma += v[i];

    return suma/n;
}


int main(int argc, char *argv[])
{
    int i;
    int n;
    float *v;

    printf("Número de elementos del vector: ");
    scanf("%d",&n);

    // Creación del vector

    v = malloc(n*sizeof(float));

    // Manejo del vector

    for (i=0; i<n; i++)
        v[i] = i;

    printf("Media = %f\n", media(v,n));

    // Liberación de memoria

    free(v);

    return 0;
}
```

Ejemplo: TDA Vector Dinámico

Tipo de los elementos del vector dinámico

```
typedef int Dato;
```

Estructura de datos del vector

```
typedef struct Vector {
    Dato *datos;          // Vector de datos
    int   usado;          // Elementos usados del vector
    int   capacidad;      // Capacidad del vector
  };

typedef struct Vector *Vector;
```

Creación del vector (constructor)

```
Vector crearVector (void)
{
  Vector v =(Vector) malloc ( sizeof(struct Vector) );

  v->usado      = 0;
  v->capacidad  = 2;
  v->datos = malloc ( (v->capacidad)*sizeof(Dato) );

  return vector;
}
```

Destrucción del vector (destructor)

```
void destruirVector (Vector *v)
{
  free ( (*v)->datos );
  free ( *v );

  *v = NULL;
}
```

Constructor y destructor nos permiten manejar vectores sin tener que conocer su estructura de datos interna (ni siquiera tendremos que utilizar `malloc` y `free`).

Acceso al contenido del vector
Funciones que permiten ocultar los detalles de implementación del TDA

Número de elementos del vector:

```
int elementosVector (Vector v)
{
  return v->usado;
}
```

Acceso a los elementos concretos del vector:

Obtención del valor almacenado en una posición del vector:

```
Dato obtenerDato (Vector v, int pos)
{
  if ((pos>=0) && (pos<elementosVector(v)))
     return v->datos[pos];
  else
     return NULL;
}
```

Modificación del valor almacenado en una posición del vector:

```
void guardarDato (Vector v, int pos, Dato dato)
{
  if ((pos>=0) && (pos<elementosVector(v))) {
     v->datos[pos] = dato;
  }
}
```

Inserción de datos

```
void agregarDato (Vector v, Dato dato)
{
  int i;
  Dato *datos;

  if (v->usado == v->capacidad) {

      // Redimensionar el vector
      v->capacidad *= 2;
      datos = malloc ( (v->capacidad)*sizeof(Dato) );

      for (i=0; i < v->usado; i++)
          datos[i] = v->datos[i];

      free(v->datos);
      v->datos = datos;
  }

  v->datos[v->usado] = dato;
  v->usado ++;
}
```

Eliminación de datos

```
void eliminarDato (Vector v, int pos)
{
  int i;

  if ((pos>=0) && (pos<elementosVector(v))) {

     for (i=pos; i<elementosVector(v)-1; i++)
     v->datos[i] = v->datos[i+1];

     v->usado --;
  }
}
```

¡OJO! En la implementación mostrada no contemplamos la posibilidad de que la función malloc devuelva NULL (algo que siempre deberemos hacer al programar).

Ejemplo de uso del TDA Vector Dinámico

```
#include <stdio.h>
#include “vector.h”

/* Rutina auxiliar */

void mostrarVector (Vector v)
{
  int i;

  printf( "Vector de tamaño %d:\n",
          elementosVector(v) );

  for (i=0; i<elementosVector(v); i++)
      printf("- %d\n", obtenerDato(v,i));
}


/* Programa principal */

int main ()
{
  Vector v = crearVector();

  mostrarVector (v);

  agregarDato (v,1);
  agregarDato (v,2);
  agregarDato (v,3);
  mostrarVector (v);

  eliminarDato (v,1);
  mostrarVector (v);

  guardarDato (v, 0, obtenerDato(v,0)+2);
  mostrarVector (v);

  destruirVector(&v);

  return 0;
}
```