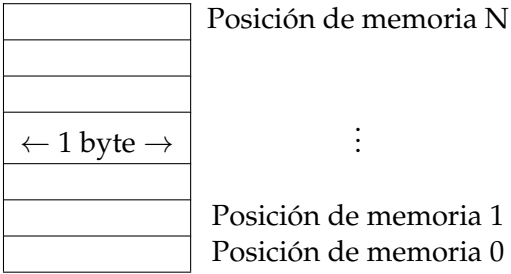
Tema 7: Punteros

Oscar Perpiñán Lamigueiro - David Álvarez

Datos y Memoria

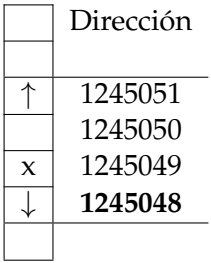
- Los datos de un programa se almacenan en la memoria del ordenador.
- La memoria del ordenador está estructurada en **bytes** (8 bits).
- Cada byte tiene una posición en la memoria (dirección).



Dirección de memoria de una variable

- Ejemplo: un dato `int` ocupa 4 bytes.

```
int x;
```



Operador &

El operador `&` (*ampersand*) aplicado a una variable cualquiera proporciona su dirección de memoria.

```c
#include <stdio.h>

int main()
{
  // No hace falta asignar valor inicial
  // para que la variable
  // tenga dirección de memoria
  int x;

  printf("La variable x está almacenada en %lli.\n", &x);

  return 0;
}
```

¿Qué es un puntero?

Un puntero apunta a una variable

Un **puntero** (*pointer*) es una **variable** (tipo número entero) que contiene la dirección de memoria de una variable:

- El puntero es una referencia de la variable a la que apunta.
- El valor del puntero es la dirección de memoria de la variable.
- La variable está apuntada por el puntero.

Un puntero es una variable `int`

- El contenido de una variable puntero es la dirección de memoria, un valor de tipo entero.
- Su tamaño depende del sistema:
  - Sistemas de 32 bits ocupan 4 bytes.
  - Sistemas de 64 bits ocupan 8 bytes.

```c
#include <stdio.h>

int main()
{
  char x, *p;
  // sizeof devuelve el numero de bytes de una variable o tipo de datos
  printf("La variable x ocupa %i bytes.\n", sizeof(x));
  printf("El puntero p ocupa %i bytes.\n", sizeof(p));

  return 0;
}
```

Operador *

El operador * aplicado a un puntero proporciona el valor de la variable apuntada por el puntero.

```
#include <stdio.h>

int main()
{
  float x = 10.2, *p;
  // Operador & para obtener la direccion de x
  p = &x;
  // *p y x son lo mismo
  printf("La variable apuntada vale %f", *p);
}
```

Punteros y vectores

El identificador de un vector es un puntero *constante* que apunta al primer elemento del vector.

```
#include <stdio.h>

int main()
{
  int vector[3] = {1, 2, 3};
  int *p;
  // p apunta al primer elemento del vector
  p = &vector[0];
  printf("El primer elemento es %i\n", *p);
  // De forma mas concisa
  p = vector;
  printf("El primer elemento es %i\n", *p);

  return 0;
}
```

Punteros y cadenas

El identificador de una cadena es un puntero *constante* que apunta al primer elemento.

```
#include <stdio.h>

int main()
{
  char letras[3] = {'a', 'b', 'c'};
  char *p;
  // p apunta al primer elemento
  p = &letras[0];
  printf("El primer elemento es %c\n", *p);
  // De forma mas concisa
  p = letras;
  printf("El primer elemento es %c\n", *p);

  return 0;
}
```

Punteros a estructuras

- Un puntero a una estructura se declara igual que un puntero a un tipo simple.
- Para acceder a un miembro de la estructura se emplea el operador ->.

```
#include <stdio.h>
typedef struct
{
  int y, m, d;
} fecha;

int main(){
  fecha f = {2000, 10, 15}, *p;
  // p apunta a la estructura
  p = &f;

  printf("%i-%i-%i", f.d, f.m, f.y);

  printf("%i-%i-%i", p->d, p->m, p->y);

  return 0;
}
```

Paso por referencia

- El uso de punteros en funciones permite el **paso por referencia**. De esta forma la función puede acceder (y **modificar**) a la variable original (*sin copia*).
- Las funciones que emplean **vectores y cadenas** como argumentos funcionan con **paso por referencia** (*el identificador de un vector es un puntero*).

Ejemplo

```
#include <stdio.h>
void operaciones (float x, float y, float *suma, float *prod, float *div);
int main(){
  float a = 1.0, b = 2.0; // Datos
  float suma, producto, division; // Resultados
  operaciones(a, b, &suma, &producto, &division);

  printf("S: %f \t P: %f \t D: %f \t", suma, producto, division);
  return 0;
}
//Funcion con varios resultados
void operaciones (float x, float y, float *suma, float *prod, float *div)
{
// Cada puntero sirve para un resultado
  *s = x + y;
  *p = x * y;
  *d = x / y;
}
```

`malloc` y `free`

- La **asignación dinámica** de memoria permite definir **objetos (p.ej. vectores) de dimensión variable**.
- La función `malloc` permite asignar, durante la ejecución del programa, un bloque de memoria de n bytes consecutivos para almacenar los datos (devuelve `NULL` si no es posible la asignación)
- La función `free` permite liberar un bloque de memoria previamente asignado.

Uso de malloc y free

```
int *pInt;
...
// Reservamos la memoria suficiente para almacenar
// un int y asignamos su dirección a pInt
pInt = malloc(sizeof(int));

// Comprobamos si la asignación
// se ha realizado correctamente
if (pInt == NULL) {
     printf("Error: memoria no disponible.\n");
     exit(-1);
}

... // Codigo usando el puntero

free(pInt); // Liberamos memoria al terminar
```

Ejemplo

```c
#include <stdio.h>
#include <stdlib.h> //Necesaria para malloc y free
int main () {
  int *vec, i, N = 100;
  vec = malloc(sizeof(int) * N);
  //Comprueba si malloc ha funcionado
  if (vec == NULL) {
     printf("Error: memoria no disponible.\n");
     exit(-1);
    }
  // El resultado es un puntero-vector de N elementos
  for (i = 0; i < N; ++i)
    vec[i] = i * i; // Rellenamos el puntero-vector
  for (i = 0; i < N; ++i) // Mostramos contenido
    printf("%i \t", *(vec + i));
  free(vec); // Liberamos el puntero-vector
  return 0;
}
```