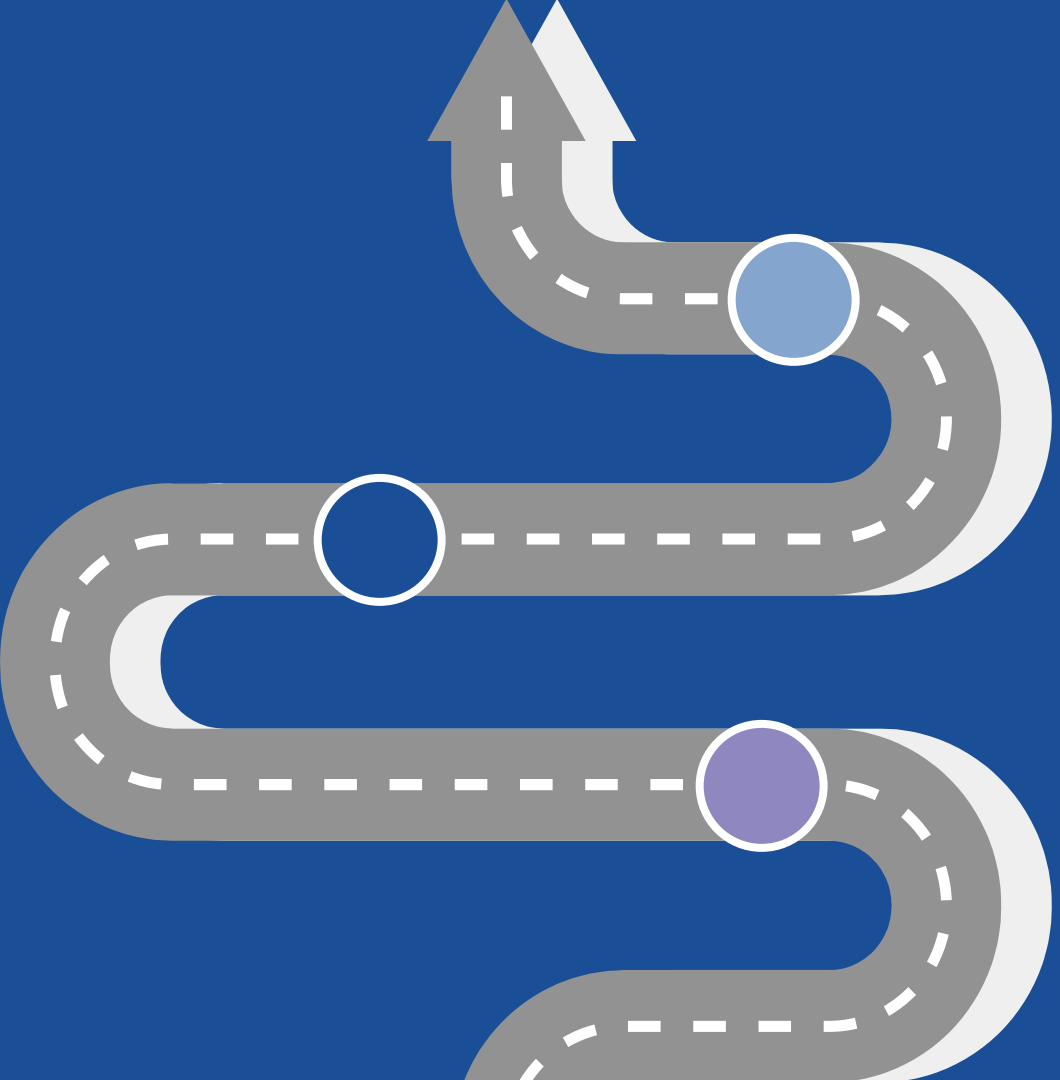


Punteros y gestión de memoria.

Profesor Yisheng León
Basado del trabajo de
Luis Hernández Yáñez
Facultad de Informática
Universidad Complutense

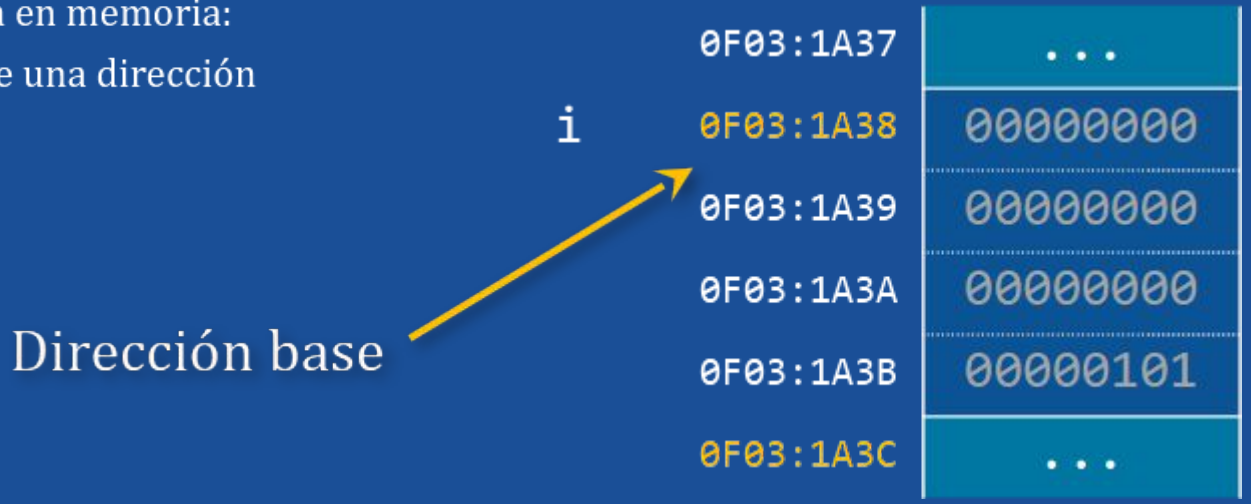
Direcciones en memoria

Los datos en la memoria

Todo dato se almacena en memoria:

Varios bytes a partir de una dirección

```
int i = 5;
```



El dato (`i`) se accede a partir de su *dirección base* (`0F03:1A38`)

Dirección de la primera celda de memoria utilizada por el dato

El tipo del dato (`int`) indica cuántos bytes (4) requiere el dato:

`00000000 00000000 00000000 00000101` → `5`

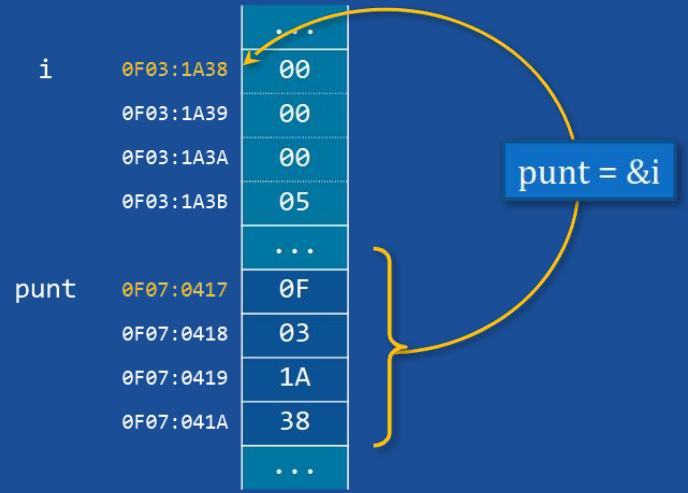
(La codificación de los datos puede ser diferente; y la de las direcciones también)

Variables punteros

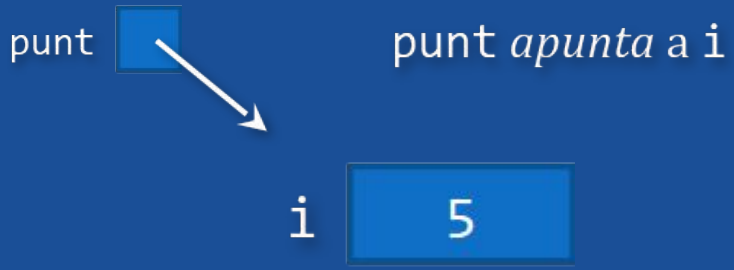
Los punteros contienen direcciones de memoria

Un *puntero* sirve para acceder a través de él a otro dato

El valor del puntero es la dirección de memoria base de otro dato



Indirección:
Acceso indirecto a un dato



Variables punteros

Los punteros contienen direcciones de memoria

¿De qué tipo es el dato apuntado?

La variable a la que apunta un puntero será de un tipo concreto

¿Cuánto ocupa? ¿Cómo se interpreta?

El tipo de variable apuntado se establece al declarar el puntero:

```
Tipo-base *nombre;
```

El puntero *nombre* apuntará a una variable del *tipo* indicado por el tipo base

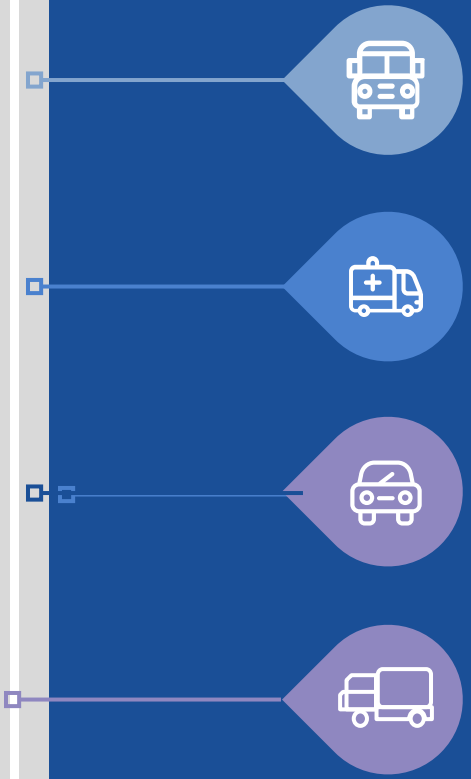
El asterisco (*) indica que es un puntero a datos de ese tipo-base

```
int *punt; // punt inicialmente contiene una dirección
           // que no es válida (no apunta a nada)
```

El puntero punt apuntará a una variable entera (int)

```
int i; // Dato entero  vs.  int *punt; // Puntero a entero
```

Punteros



Características

Los punteros contienen direcciones de memoria
Al declararlas sin inicializador contienen direcciones no válidas

A qué apunta?

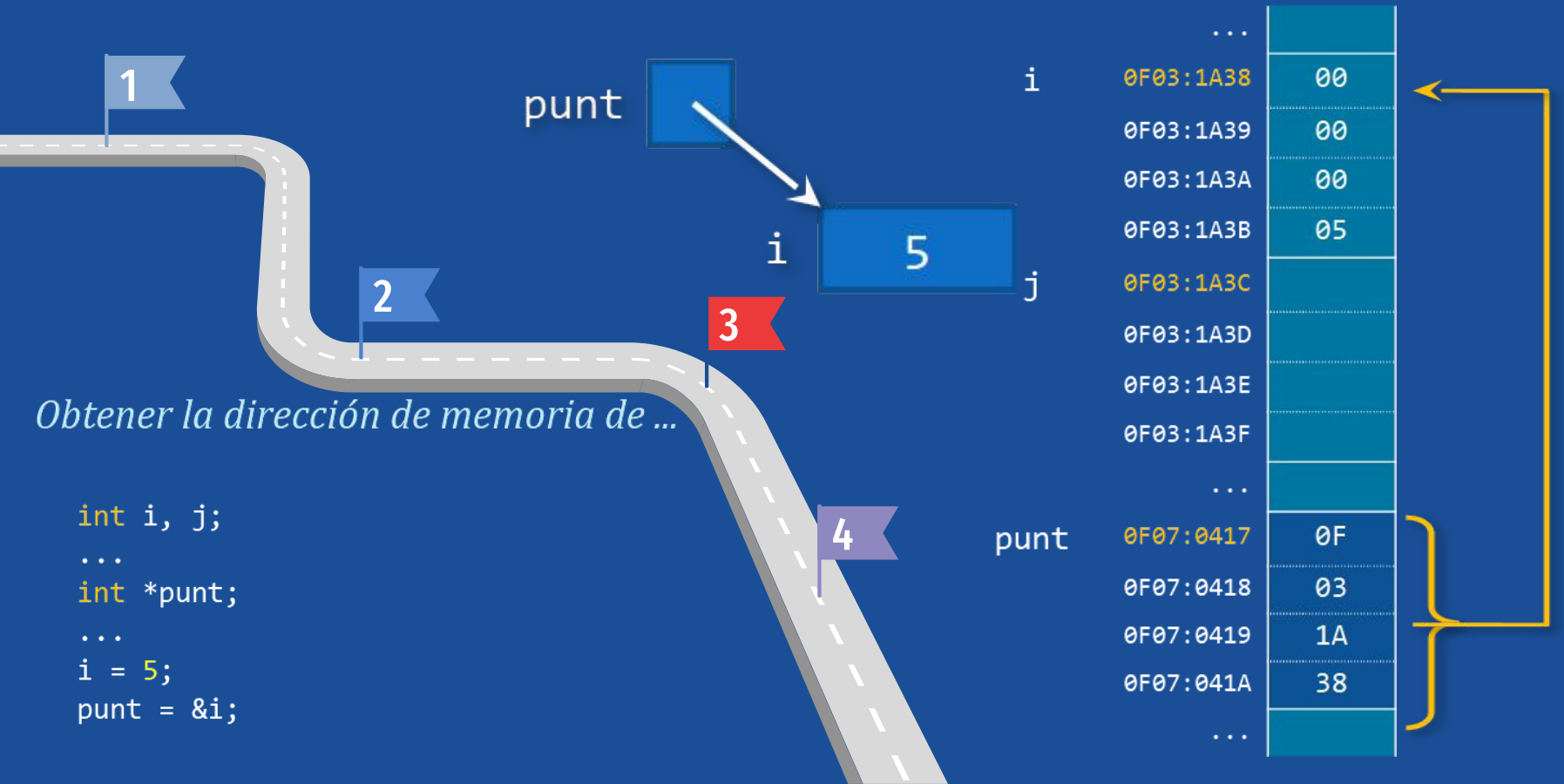
Un puntero puede apuntar a cualquier dato de su tipo base
Un puntero no tiene por qué apuntar necesariamente a un dato (puede no apuntar a nada: valor NULL)

```
int *punt; // punt inicialmente contiene una dirección
           // que no es válida (no apunta a nada)
```

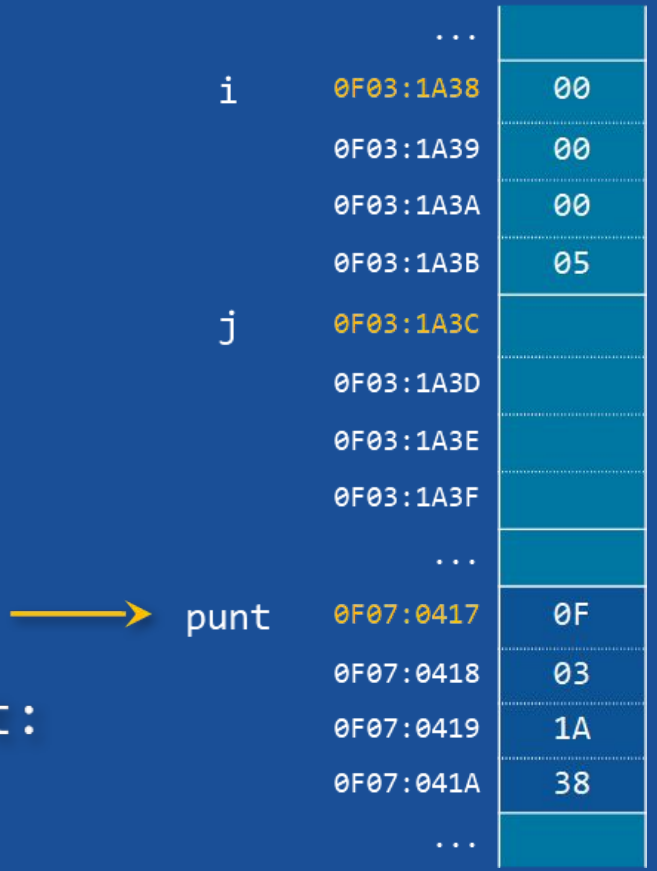
Para qué sirve?

Para implementar el paso de parámetros por referencia
Para manejar datos dinámicos
(Datos que se crean y destruyen durante la ejecución)
Para implementar los arrays

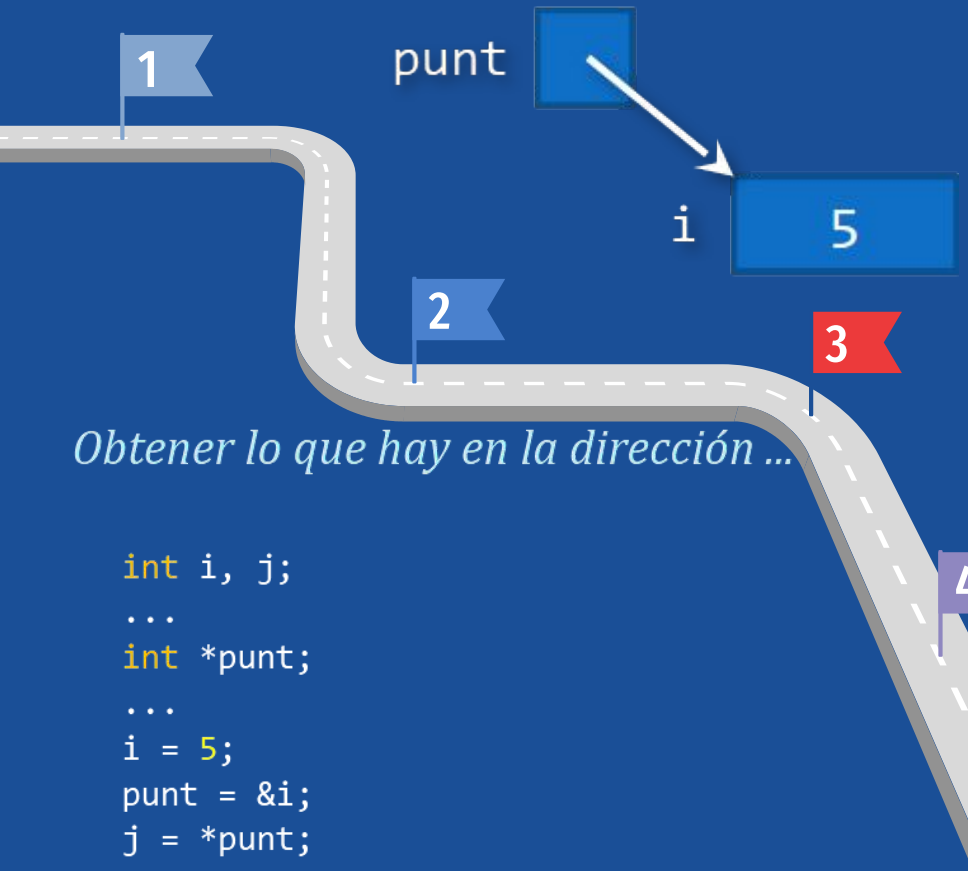
Operadores de punteros



Operadores de punteros



Operadores de punteros



Obtener lo que hay en la dirección ...

```
int i, j;
...
int *punt;
...
i = 5;
punt = &i;
j = *punt;
```

Direccionamiento indirecto (*indirección*)
Se accede al dato `i` de forma indirecta

*punt:

	...	
i	0F03:1A38	00
	0F03:1A39	00
	0F03:1A3A	00
	0F03:1A3B	05
j	0F03:1A3C	
	0F03:1A3D	
	0F03:1A3E	
	0F03:1A3F	
	...	
punt	0F07:0417	0F
	0F07:0418	03
	0F07:0419	1A
	0F07:041A	38
	...	

Punteros y direcciones válidas

Todo puntero ha de tener una dirección válida

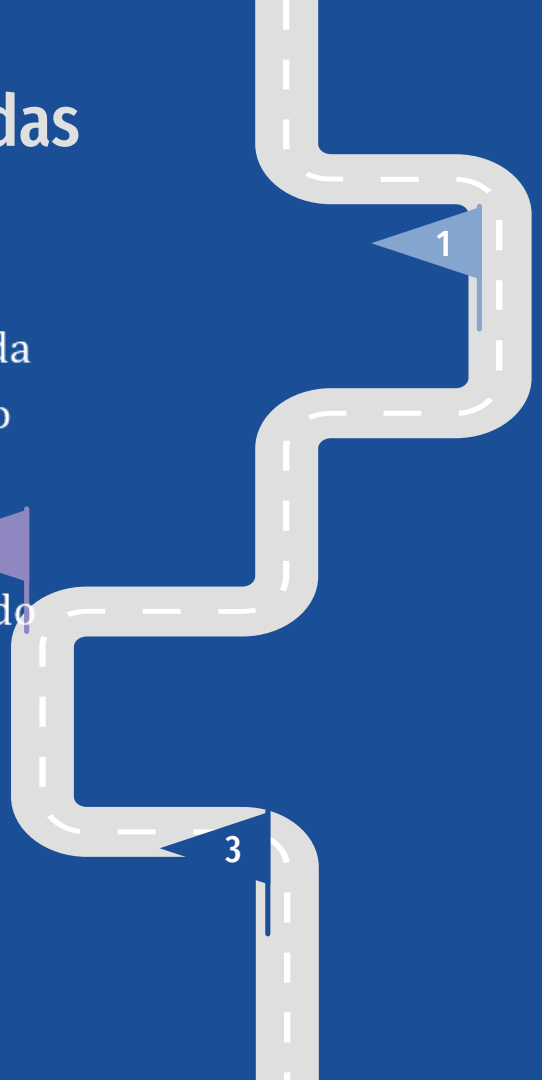
Un puntero sólo debe ser utilizado si tiene una dirección válida

Un puntero NO contiene una dirección válida tras ser definido

Un puntero obtiene una dirección válida:

- ✓ Asignando la dirección de otro dato (operador &)
- ✓ Asignando otro puntero (mismo tipo base) que ya sea válido
- ✓ Asignando el valor `NULL` (puntero nulo, no apunta a nada)

```
int i;
int *q; // q no tiene aún una dirección válida
int *p = &i; // p toma una dirección válida
q = p; // ahora q ya tiene una dirección válida
q = NULL; // otra dirección válida para q
```



Punteros no inicializados

Punteros que apuntan a saber qué...

Un puntero no inicializado contiene una dirección desconocida

```
int *punt; // No inicializado
*punt = 12; // ¿A qué dato se está asignando el valor?
```

¿Dirección de la zona de datos del programa?

¡Podemos modificar inadvertidamente un dato del programa!

¿Dirección de la zona de código del programa?

¡Podemos modificar el código del propio programa!

¿Dirección de la zona de código del sistema operativo?

¡Podemos modificar el código del propio S.O.!

→ Consecuencias imprevisibles (*cuelgue*)

(Los S.O. modernos protegen bien la memoria)

Un valor seguro : NULL

Punteros que no apuntan a nada

Inicializando los punteros a `NULL` podemos detectar errores:

```
int *punt = NULL;
...
*punt = 13;
```

`punt` ha sido inicializado a `NULL`: ¡No apunta a nada!

Si no apunta a nada, ¿¿¿qué significa `*punt`??? No tiene sentido

→ ERROR: *¡Acceso a un dato a través de un puntero nulo!*

Error de ejecución, lo que ciertamente no es bueno

Pero sabemos cuál ha sido el problema, lo que es mucho

Sabemos dónde y qué buscar para depurar

Apuntando al mismo dato

Copia de punteros

Al copiar un puntero en otro, ambos apuntarán al mismo dato:

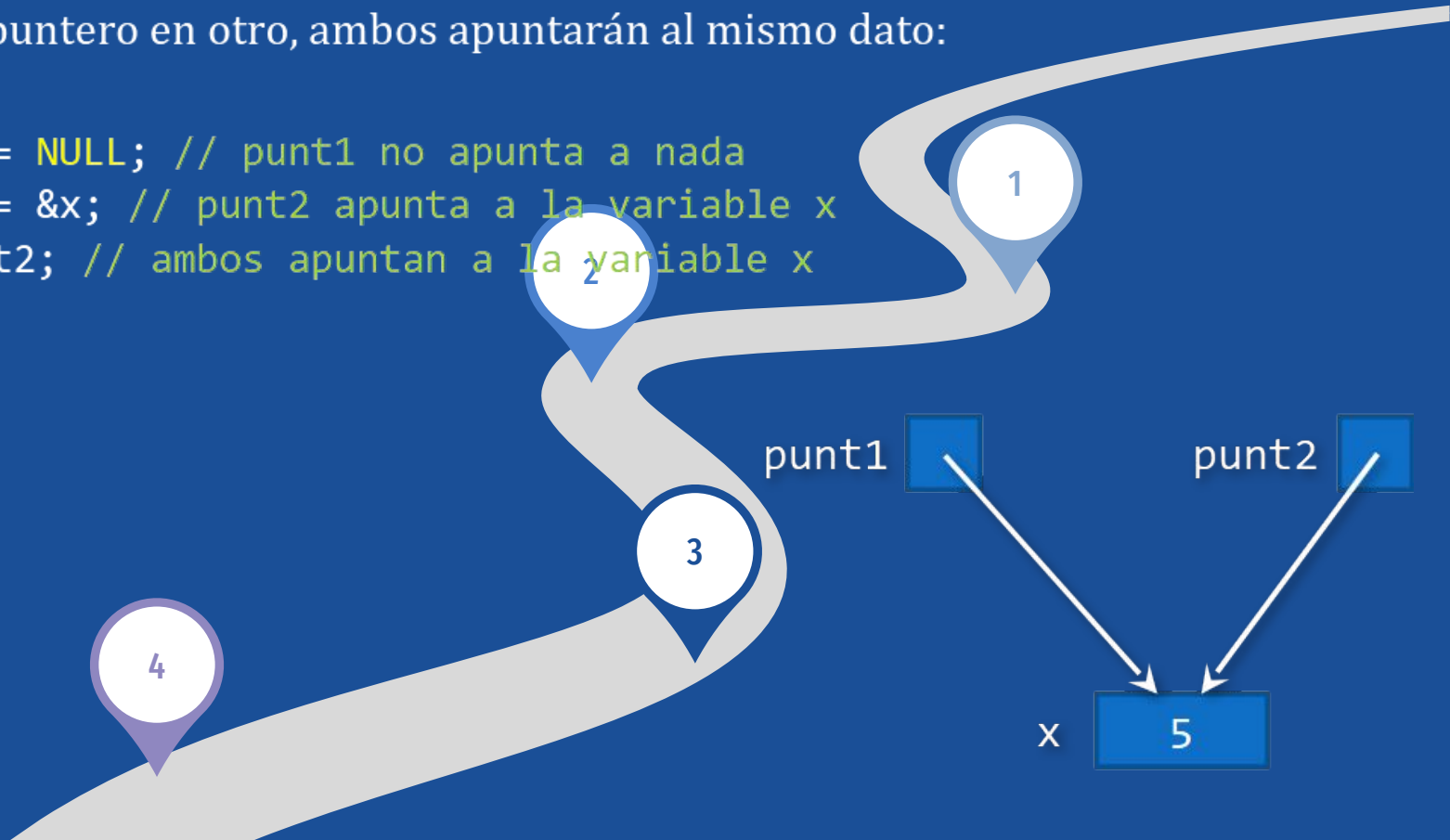
```
int x = 5;
int *punt1 = NULL; // punt1 no apunta a nada
int *punt2 = &x; // punt2 apunta a la variable x
```

Apuntando al mismo dato

Copia de punteros

Al copiar un puntero en otro, ambos apuntarán al mismo dato:

```
int x = 5;
int *punt1 = NULL; // punt1 no apunta a nada
int *punt2 = &x; // punt2 apunta a la variable x
punt1 = punt2; // ambos apuntan a la variable x
```



Copia de punteros

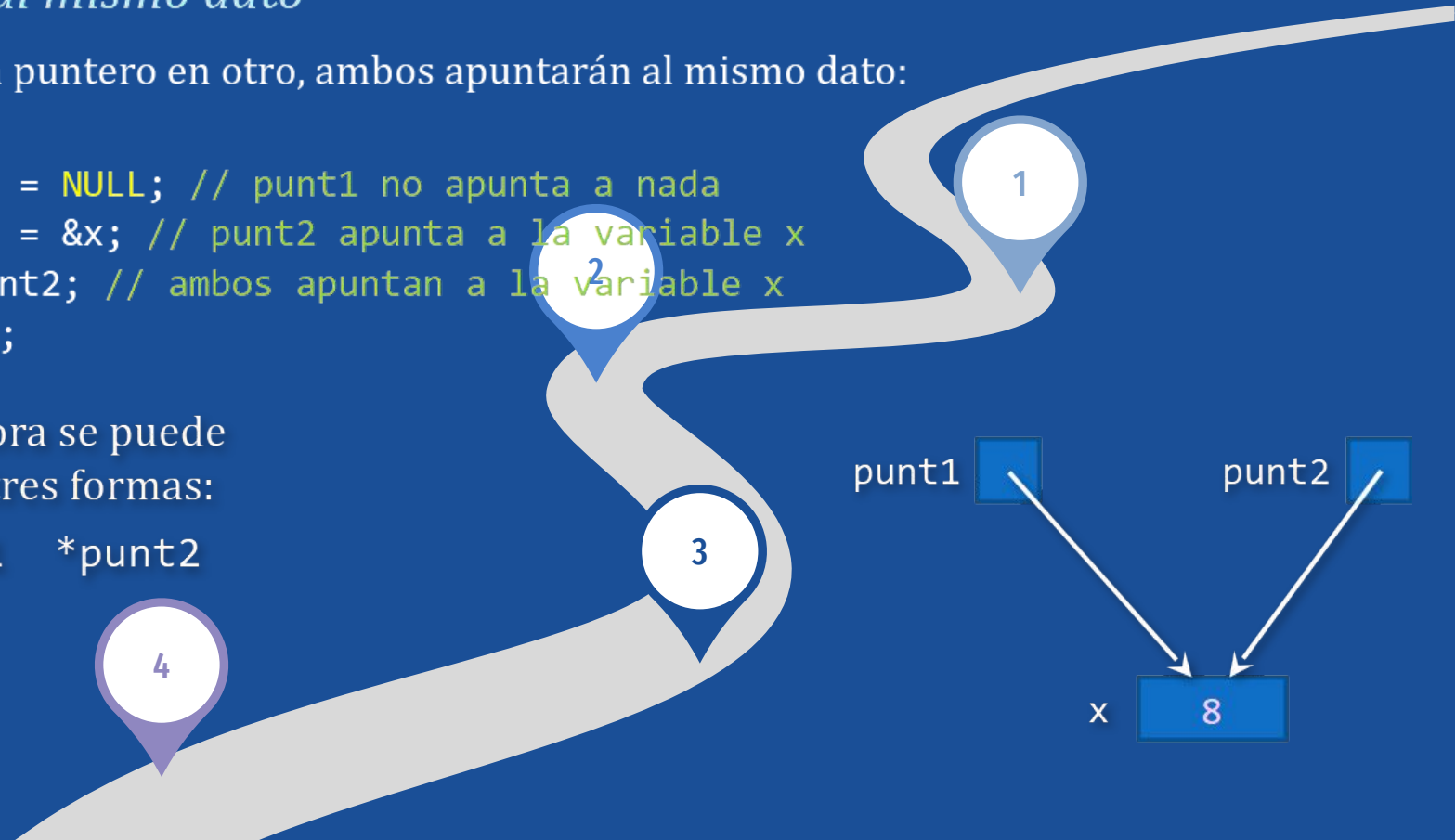
Apuntando al mismo dato

Al copiar un puntero en otro, ambos apuntarán al mismo dato:

```
int x = 5;
int *punt1 = NULL; // punt1 no apunta a nada
int *punt2 = &x; // punt2 apunta a la variable x
punt1 = punt2; // ambos apuntan a la variable x
*punt1 = 8;
```

Al dato x ahora se puede acceder de tres formas:

x *punt1 *punt2



Tipos de punteros

Declaración de tipos puntero

Declaramos tipos para los punteros con distintos tipos base:

```
typedef int *tIntPtr;
typedef char *tCharPtr;
typedef double *tDoublePtr;

int entero = 5;
tIntPtr puntI = &entero;
char caracter = 'C';
tCharPtr puntC = &caracter;
double real = 5.23;
tDoublePtr puntD = &real;

cout << *puntI << "  " << *puntC << "  " << *puntD << endl;
```

```
Int *p;
p = &i;
Cout << *p << endl;
```

Con **puntero* podemos hacer lo que con otros datos del tipo base

Tipos de punteros

Acceso a estructuras a través de punteros

Los punteros pueden apuntar también a estructuras:

```
typedef struct {
   int codigo;
   string nombre;
   double sueldo;
} tRegistro;
tRegistro registro;
typedef tRegistro *tRegistroPtr;
tRegistroPtr puntero = &registro;
```

Operador flecha (->):

Acceso a los campos a través de un puntero sin usar el operador *

```
puntero->codigo     puntero->nombre     puntero->sueldo
```

```
puntero->…  (*puntero).…
```

Punteros a estructuras

Acceso a estructuras a través de punteros

```
typedef struct {
   int codigo;
   string nombre;
   double sueldo;
} tRegistro;
tRegistro registro;
typedef tRegistro *tRegistroPtr;
tRegistroPtr puntero = &registro;
registro.codigo = 12345;
registro.nombre = "Javier";
registro.sueldo = 95000;
cout << puntero->codigo << "   " << puntero->nombre
     << "   " << puntero->sueldo << endl;

puntero->codigo  (*puntero).codigo    *puntero.codigo
```

`puntero` sería una estructura con campo `codigo` de tipo puntero

Punteros y el modificador const

Punteros a constantes y punteros constantes

El efecto del modificador de acceso `const` depende de su sitio:

`const tipo *puntero;` Puntero a una constante

`tipo *const puntero;` Puntero constante

Punteros a constantes:

```
typedef const int *tIntCtePtr; // Puntero a constante
int entero1 = 5, entero2 = 13;
tIntCtePtr punt_a_cte = &entero1;

(*punt_a_cte)++; // ERROR: ¡Dato no modificable!
punt_a_cte = &entero2; // OK: El puntero no es cte.
```

Punteros Punteros y el modificador const

Punteros a constantes y punteros constantes

El efecto del modificador de acceso `const` depende de su sitio:

`const tipo *puntero;` Puntero a una constante

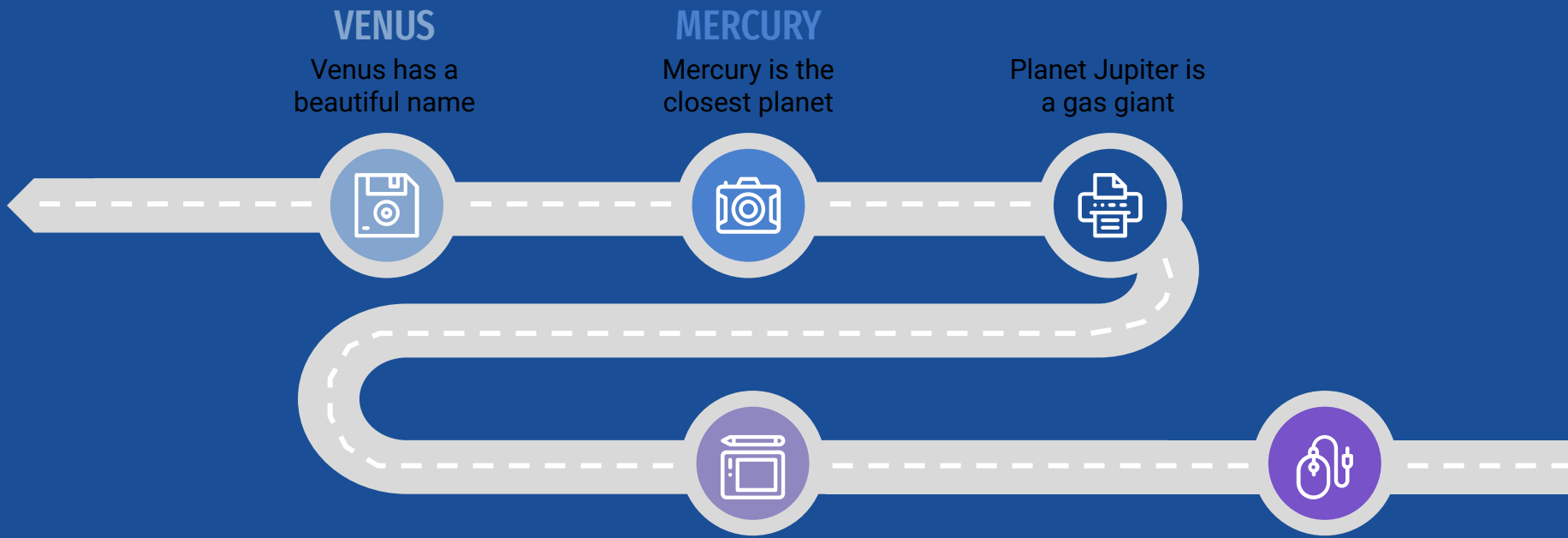
`tipo *const puntero;` Puntero constante

Punteros constantes:

```
typedef int *const tIntPtrCte; // Puntero constante
int entero1 = 5, entero2 = 13;
tIntPtrCte punt_cte = &entero1;

(*punt_cte)++; // OK: El puntero no apunta a cte.
punt_cte = &entero2; // ERROR: ¡Puntero constante!
```

Product Roadmap Infographics



Product Roadmap Infographics

MERCURY

Mercury is the closest planet to the Sun and the smallest one

A

VENUS

Venus has a beautiful name and is the second planet

B

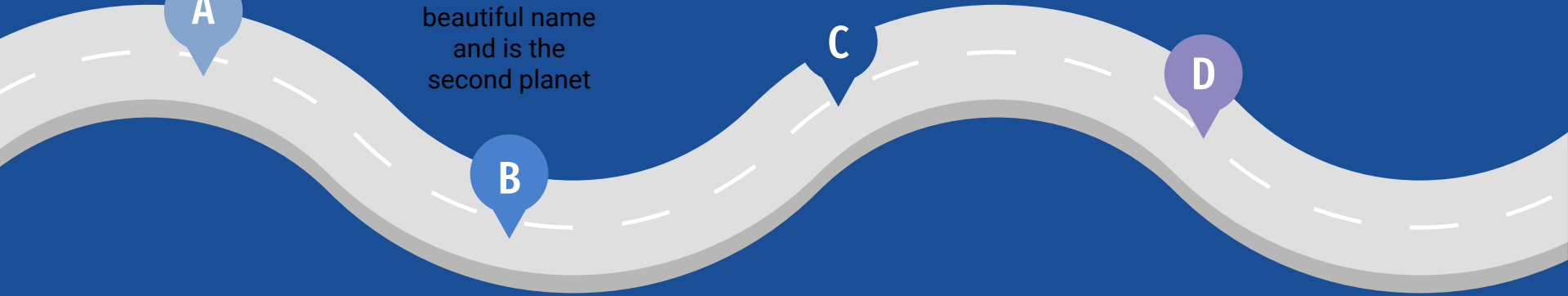
Despite being red, Mars is a cold place full of iron oxide dust

C

JUPITER

Jupiter is a gas giant and the biggest planet in the Solar System

D



Paso de Parámetros

Paso de parámetros por referencia o variable

En el lenguaje C no hay mecanismo de paso por referencia (&)

Sólo se pueden pasar parámetros por valor

¿Cómo se simula el paso por referencia? Por medio de punteros:

```
void incrementa(int *punt);
```

```
Ejemplo:
void incrementa(int *punt) {
   (*punt)++;
}
...
int entero = 5;
incrementa(&entero);
cout << entero << endl;
```

Mostrará 6 en la consola

Paso de Parámetros

Paso de parámetros por referencia o variable

```
int entero = 5;
incrementa(&entero);
```

entero 5

punt recibe la dirección de entero

```
void incrementa(int *punt) {
  (*punt)++;
}
```

punt

entero 6

```
cout << entero << endl;
```

entero 6

Paso de Parámetros

Paso de parámetros por referencia o variable

¿Cuál es el equivalente en C a este prototipo de C++?

```
void foo(int &param1, double &param2, char &param3);
```

Prototipo equivalente:

```
void foo(int *param1, double *param2, char *param3);
```

```
void foo(int *param1, double *param2, char *param3) {
  // Al primer argumento se accede con *param1
  // Al segundo argumento se accede con *param2
  // Al tercer argumento se accede con *param3
}
```

¿Cómo se llamaría?

```
int entero; double real; char caracter;
//...
foo(&entero, &real, &caracter);
```