

Tema 1:

Introducción

GRADO EN INGENIERÍA DEL SOFTWARE

Salvador Sánchez y Sergio Caverio



Universidad
Rey Juan Carlos

Grado en Ingeniería del Software

Contenidos

- Abstracción
- Tipos de datos
- Memoria dinámica y punteros
- Implementación en Pascal

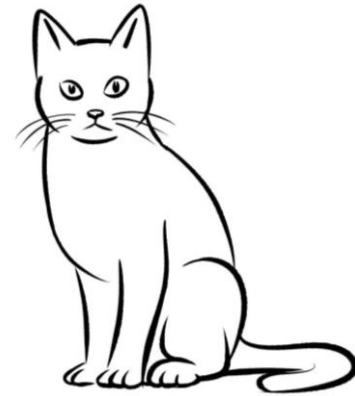
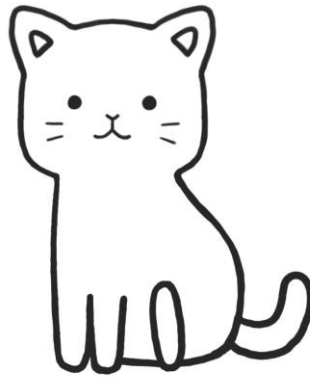
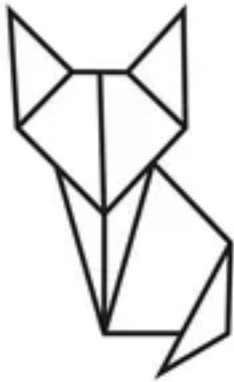
ABSTRACCIÓN

Definición de abstracción

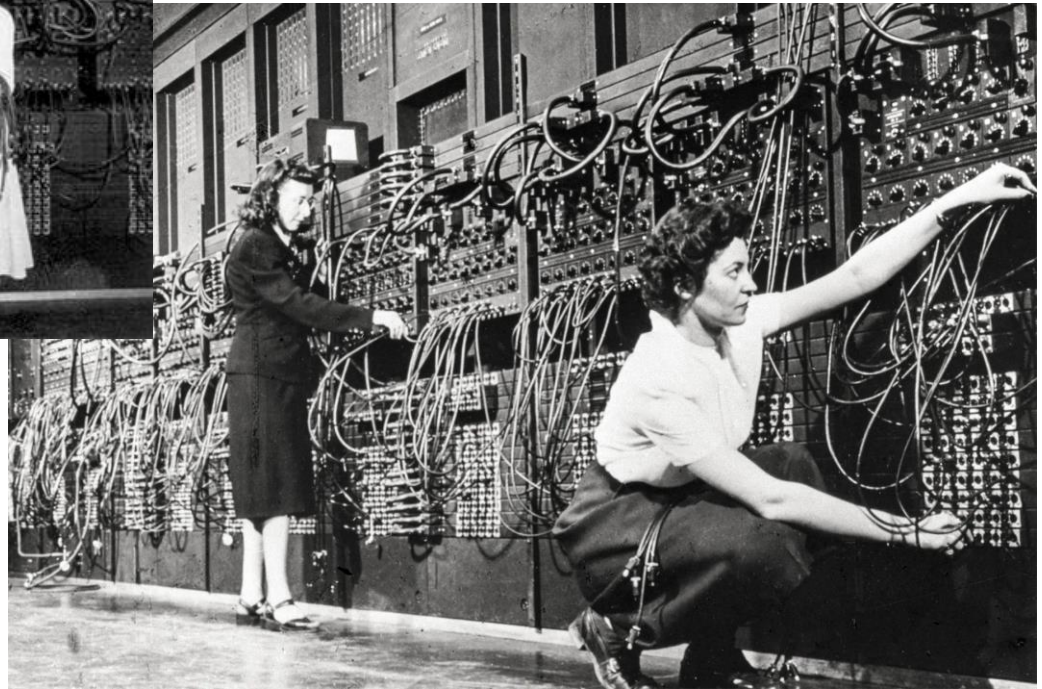
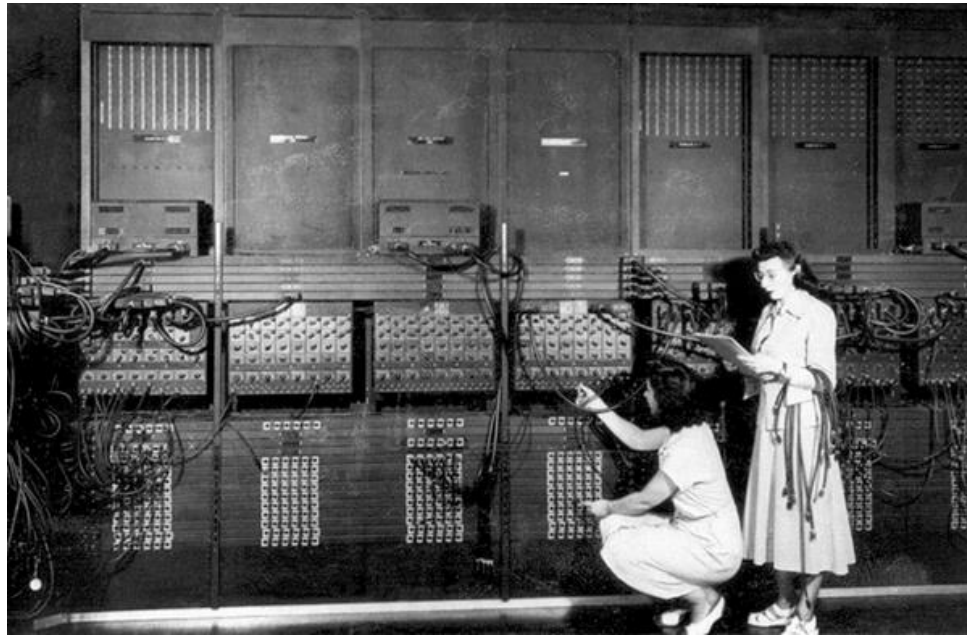
Definición: *“Proceso mental por el que el ser humano extrae las características esenciales de algo, ignorando los detalles superfluos”*

- Es primordial para modelar el mundo real sin perderse en detalles y pormenores que no resultan esenciales.
- Veamos unos ejemplos...

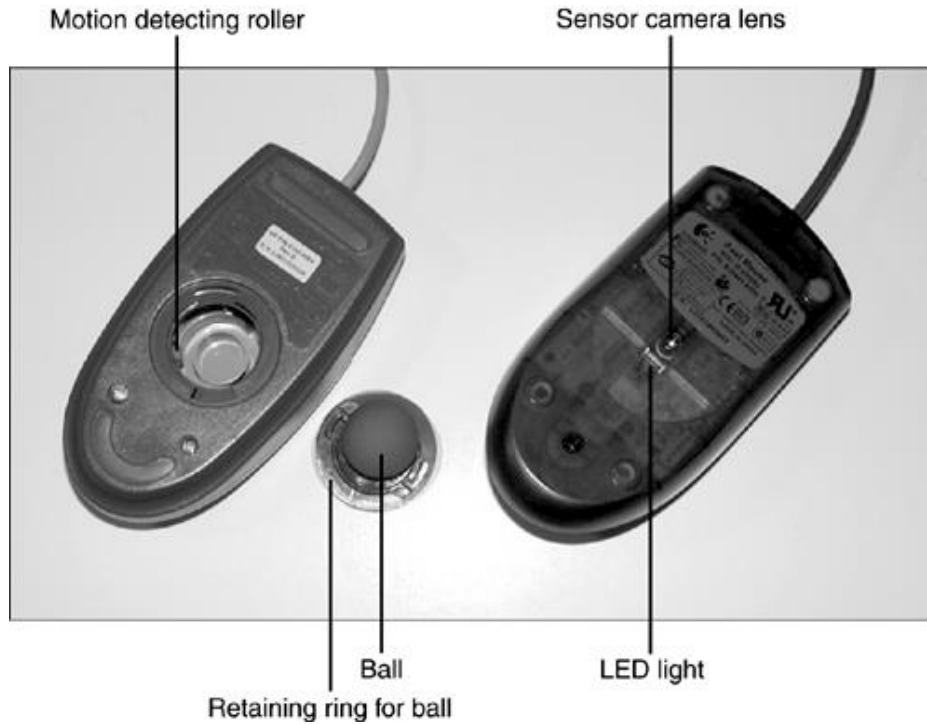
Abstracción: ejemplos



Abstracción: ejemplos



Abstracción: ejemplos



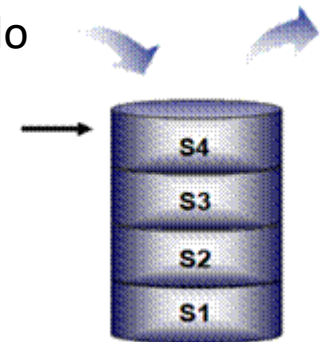
Conceptos relacionados

- **Caja negra:** estudio del comportamiento de un sistema obviando su funcionamiento interno y centrándose únicamente en el exterior observable.
 - Métodos en ingeniería eléctrica (Cauer, 1940s), donde se analizaban circuitos por la respuesta que producían a ciertas entradas y no por su estructura interna (caja que no puede abrirse)
 - Ross Ashby generalizó este concepto aplicándolo a la cibernética (1958) afirmando que todo sistema complejo puede ser estudiado como una máquina, cuyo interior se ignora.



Conceptos relacionados

- **Encapsulamiento:** proceso por el que se ocultan los detalles de las características de una abstracción.
 - En programación resulta esencial para reutilizar código.
 - Ocultando los detalles de cómo está hecho un módulo/programa pero conociendo cómo operarlo, se puede utilizar en cualquier otro programa (incluso modificando su interior).
 - Se oculta también el estado interno
 - Base para impedir utilizar el objeto de modos no deseados: solo mediante las operaciones definidas



Conceptos relacionados

- **Interfaz:** Conjunto de elementos mediante los cuales puede interactuarse con un objeto que es visto como una caja negra.
 - Medio para que los objetos se comuniquen entre sí.
- **Contrato:** metáfora según la cual los elementos de un sistema software colaboran sobre la base de obligaciones y beneficios mutuos y públicamente declarados, de modo similar a un contrato (B. Meyer).
 - Diseñador: pone a disposición (publica en la interfaz) varios componentes y oculta el resto. Se despreocupa de posibles malos usos
 - Usuario: utiliza los componentes que le garantizan unas ciertas salidas, centrándose en resolver su problema
 - Permite cambiar la implementación sin alterar el uso
 - Se emplea la **encapsulación** para evitar el acceso a elementos que no forman parte del contrato (no publicados en la interfaz).



Abstracción y conceptos relacionados

- **Ejemplo:** Un coche automático

- Funciones principales (lo que se ofrece al conductor):

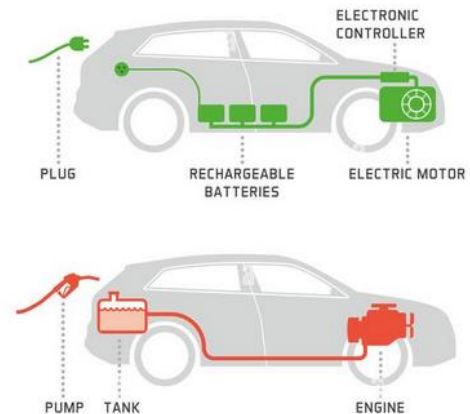
- Girar, acelerar y frenar

- Interfaces:

- Girar → Volante
- Acelerar, frenar → Pedales

- Funcionamiento interno (implementación):

- Diesel/eléctrico/híbrido, sistemas de frenos, tracción delantera/trasera, dirección eléctrica/hidráulica, etc.
- Todo esto se oculta **deliberadamente** para que cualquier conductor pueda conducir cualquier coche.



Diseño de abstracciones

Pensemos en la interfaz, contrato y diferentes implementaciones de...

- Una linterna
- Una placa de cristal para cocinar (con un fuego)
- Un bombo de números de lotería
 - Un bombo de premios de la lotería
- Un contador
- Un conjunto
 - Genérico / Específico
- Una cola de personas
- Una tabla de récords

Tipos de datos

Tipo de datos

- Clasificación de los tipos de datos:
- A) Predefinidos: uso común
 - Tipos “simples”: Integer, Real, Boolean, Char...
 - Tipos “estructurados”: String
- B) Definidos por el usuario: mediante herramientas del lenguaje
 - Simples: subrango, enumerado, puntero
 - Estructurados: Conjunto, Array, Registro, Archivo

Free Pascal como ejemplo

integer types

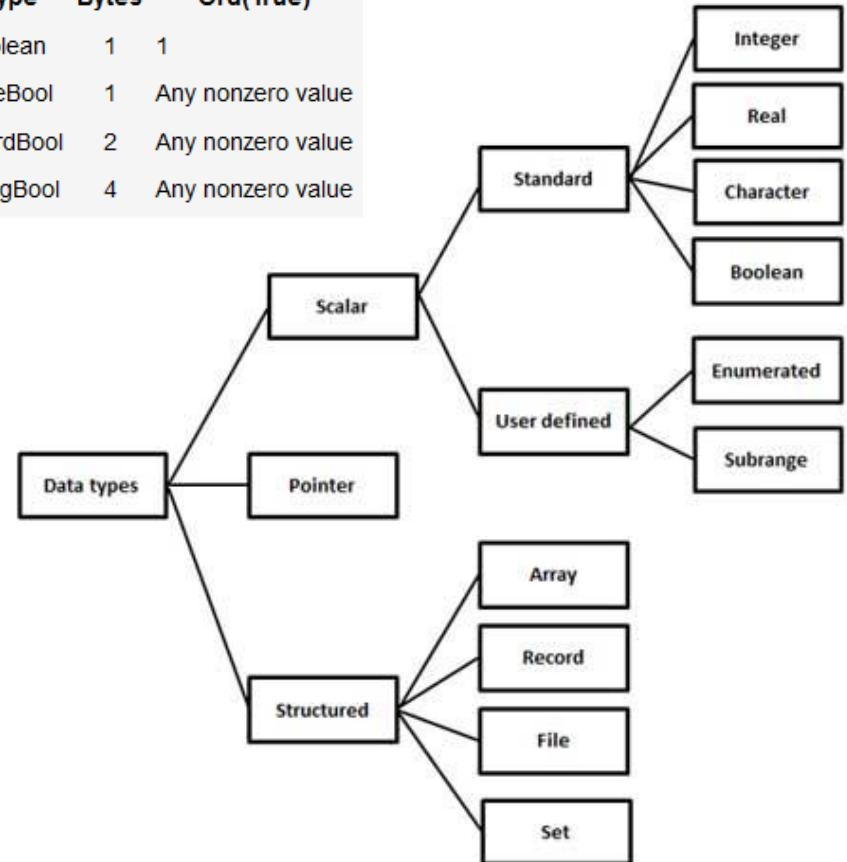
Type	Range	Bytes
Byte	0 .. 255	1
Shortint	-128 .. 127	1
Smallint	-32768 .. 32767	2
Word	0 .. 65535	2
Integer	smallint or longint	2 or 4
Cardinal	longword	4
Longint	-2147483648 .. 2147483647	4
Longword	0..4294967295	4
Int64	-9223372036854775808 .. 9223372036854775807	8
QWord	0 .. 18446744073709551615	8

real types

Type	Range	Significant digits	Bytes
Real	platform dependent	???	4 or 8
Single	1.5E-45 .. 3.4E38	7-8	4
Double	5.0E-324 .. 1.7E308	15-16	8
Extended*	1.9E-4932 .. 1.1E4932	19-20	10
Comp	-2E64+1 .. 2E63-1	19-20	8
Currency	-922337203685477.5808 .. 922337203685477.5807		8

boolean types

Type	Bytes	Ord(True)
Boolean	1	1
ByteBool	1	Any nonzero value
WordBool	2	Any nonzero value
LongBool	4	Any nonzero value



Especificación de tipos simples

- Los mecanismos de construcción de tipos de los lenguajes vistos hasta el momento tienen dos limitaciones importantes:
 - Solo podemos modelar estructura, no comportamiento
 - Lo resolveremos en POO con las clases (*Asignatura Programación Orientada Objetos*)
 - Tenemos que limitarnos a los tipos que nos ofrece el lenguaje, y así quedan fuera diccionarios, pilas, colas, listas, etc.
 - Realmente **NO**. Los punteros van a ayudarnos a superar esta limitación.

Tipo de datos como abstracción

- Los tipos de datos pueden verse como abstracciones donde cada elemento del tipo está sujeto a un contrato
- Ejemplo: booleanos
 - **Nombre:** boolean
 - **Valores posibles (estado interno):** true, false
 - **Operaciones:** and, or, not
 - **Uso de otros tipos:** no. Solo booleanos

Tipo de datos como abstracción

- Ejemplo: array de enteros
 - **Valores posibles:** infinitos
 - **Operaciones:** cambiar elemento, consultar elemento, consultar tamaño, ...
 - **Uso de otros tipos:** sí: enteros.

Especificación de un tipo de datos

- La **especificación** es aquello que se permite al cliente (quien va a utilizar el tipo de datos) saber sobre el tipo
 - El contrato, en una palabra
 - Dos partes: **signatura** y **axioma**
- **Signatura:**
 - Nombre del tipo: ej. integer, boolean, etc.
 - Nombres de las operaciones: +, -, and, or, etc.
 - Argumentos y resultados de las operaciones
- **Axioma:** detalle de cómo se comportan las operaciones
(más sobre todo esto pronto)

Memoria dinámica

Objetivos

- Conocer el concepto de “puntero”
- Entender la gestión dinámica de memoria
- Manejar estructuras estáticas y dinámicas en memoria a través de punteros
- Crear y destruir estructuras dinámicas en memoria



Me breezing
thru Pascal
in school



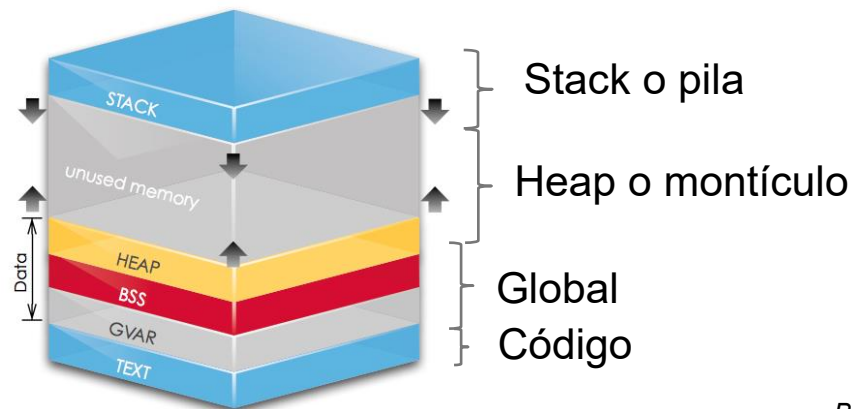
We start
learning
pointers

Estructura de la memoria

- Las **estructuras estáticas** (por ejemplo, array) presentan rigideces que no pueden superarse con la gestión de memoria vista hasta ahora:
 - Cambiar la disposición de los elementos dentro de la estructura estática es, a veces, costoso.
 - Ej: Colocar el último elemento al comienzo del array.
 - No se puede cambiar su tamaño durante la ejecución del programa
 - A menudo es necesario acomodar (redimensionar) una estructura de datos a un número de elementos desconocido en compilación
 - Además, hay otros factores importantes a tener en cuenta sobre el uso de la memoria en los procesos.

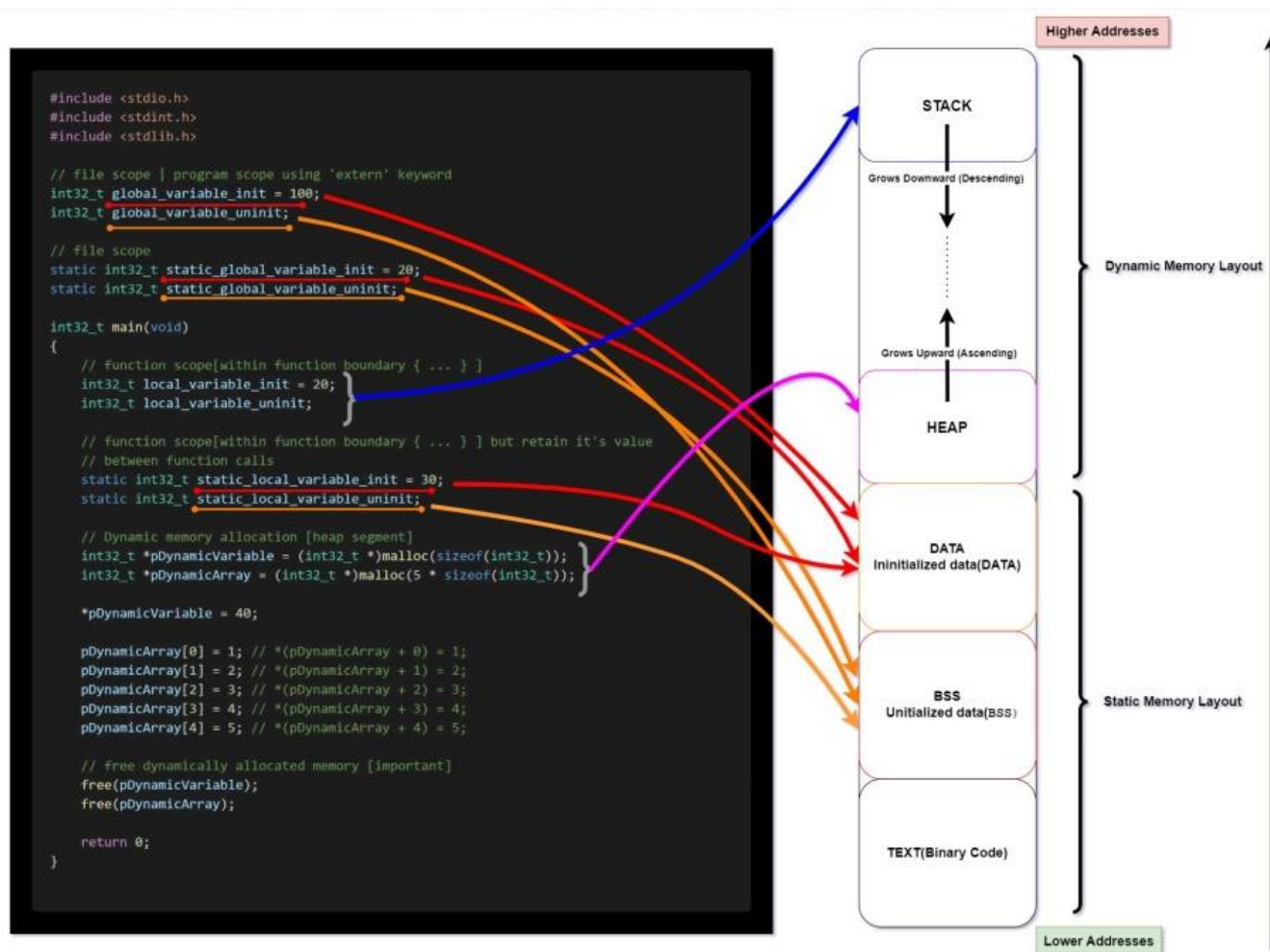
Estructura de la memoria

- El espacio de memoria en un sistema está descompuesto de forma general en 4 bloques con tamaños diversos
 - Segmento de código: asignación **automática**
 - Variables globales: asignación **automática**
 - *Stack* o pila de memoria: asignación **automática**
 - *Heap* o montículo de memoria: asignación **manual**



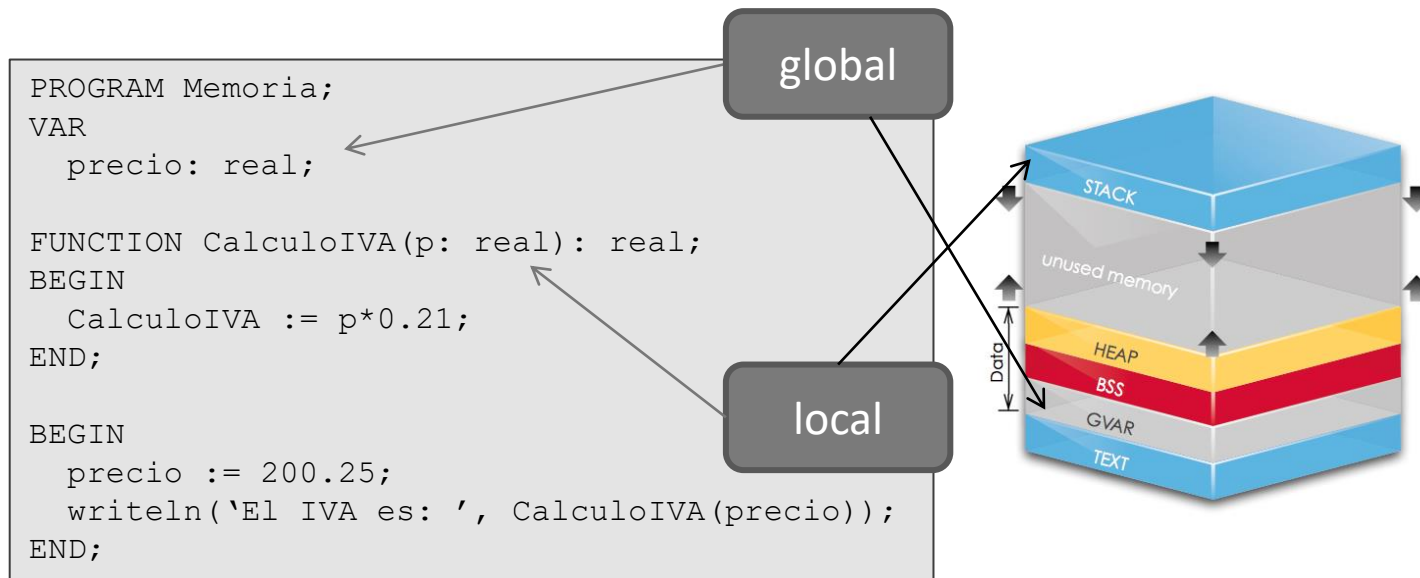
BSS: variables globales y static sin inicializar

Estructura de la memoria



Estructura de la memoria

- La memoria local a los subprogramas se gestiona en la pila de memoria
 - Cada proceso de un programa tiene su propia pila de memoria, por lo que en general la pila tiene un tamaño muy limitado
- La memoria dinámica se gestiona en un bloque muy grande de memoria (*heap*/montículo)



Definición del problema

- En muchos problemas no se conoce en tiempo de diseño cuánta memoria necesitaremos ni cómo se va a organizar
- **Solución:** definir y organizar esa memoria en tiempo de ejecución
 - Para ello, se utilizan estructuras de memoria dinámica.
- La gestión de memoria dinámica se realiza a través de variables capaces de guardar direcciones de memoria: **punteros**

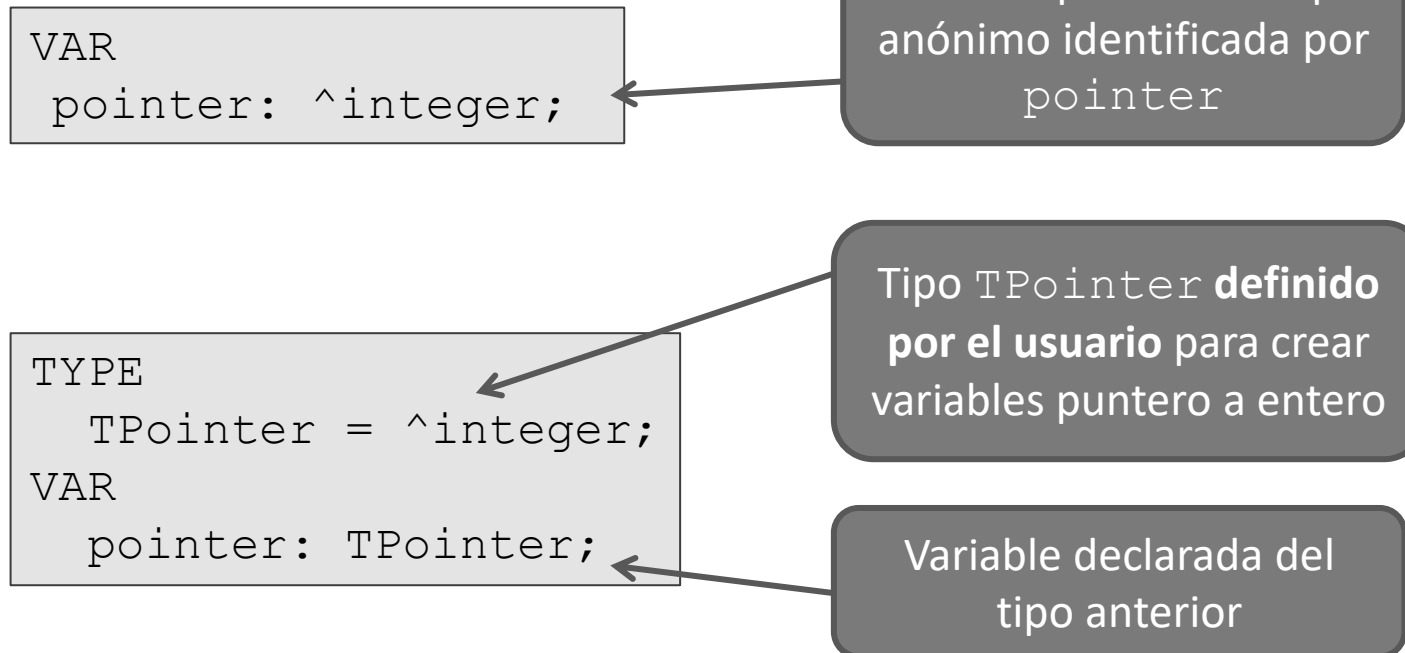
¿Qué es eso de ...?

- **Memoria dinámica:** Memoria en la que se puede reservar espacio en tiempo de ejecución.
 - El *heap* es el bloque del espacio direccionable de memoria dedicado para la memoria dinámica.
- **Estructuras de datos dinámicas:** Colección de elementos (nodos) que se crean o destruyen en tiempo de ejecución.
- **Variables Dinámicas:** Posiciones de memoria reservadas en tiempo de ejecución.

Punteros en Pascal

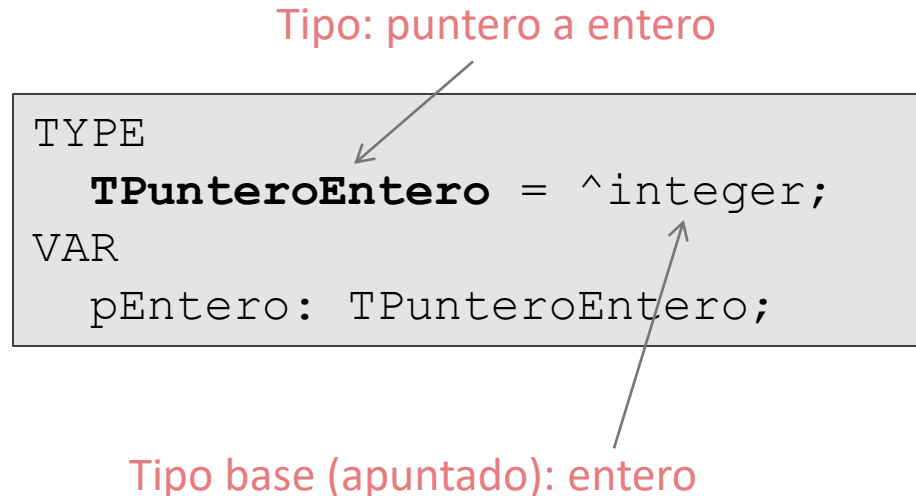
Punteros

- Una variable puntero (igual que otras variables) se puede declarar con tipo anónimo, o como tipo definido por el usuario.



Punteros

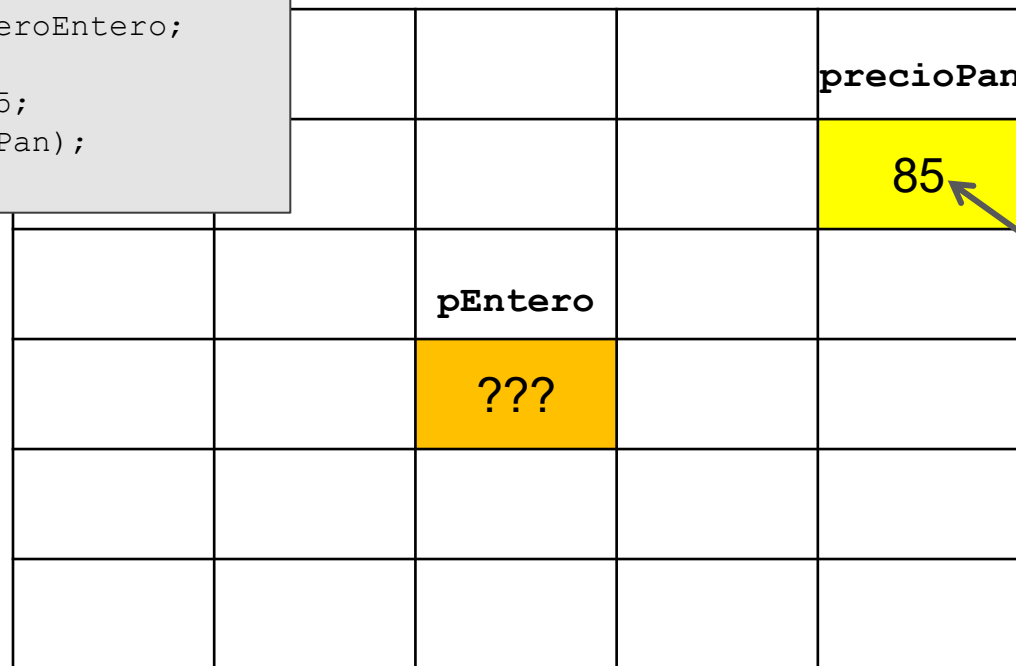
- Un tipo puntero se puede usar para declarar variables de ese tipo



- Una variable puntero almacena una dirección de memoria donde guardar un valor del “tipo base” del puntero
 - Ej: puntero a entero guarda una dirección donde se guarda un entero

Punteros: representación en memoria

```
TYPE
  TPrecio = integer;
  TPunteroEntero = ^integer;
VAR
  precioPan: TPrecio;
  pEntero: TPunteroEntero;
BEGIN
  precioPan := 85;
  writeln(precioPan);
END.
```



Dirección \$3\$12
identificada
como
`precioPan`

Valor (85) que
guarda
`precioPan`

Operaciones con punteros

Operador @ (Referencia)

- Obtención de la dirección de memoria de una variable

```
VAR
  pEntero: ^integer;
  precioPan: integer;
BEGIN
  precioPan := 85;
  pEntero   := @precioPan;
  ...
```

Operador ^ (Des-referencia)

- Acceso al valor de la variable apuntada desde el puntero

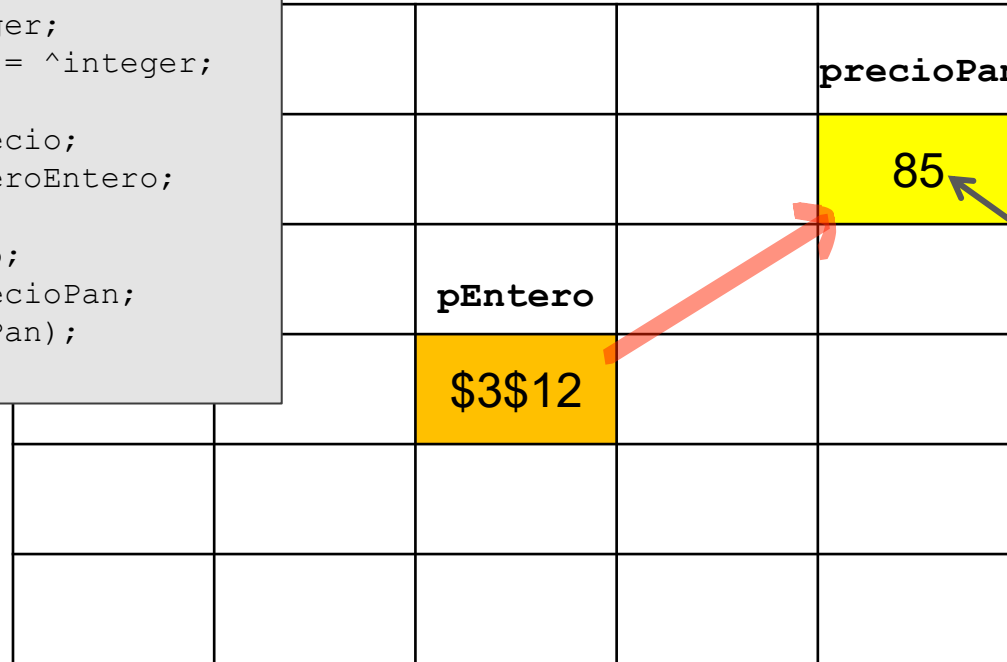
```
...
pEntero^ := 100;
writeln(precioPan); {imprime 100}
```

pEntero^ y
precioPan son
sinónimos



Punteros

```
TYPE
  TPrecio = integer;
  TPunteroEntero = ^integer;
VAR
  precioPan: TPrecio;
  pEntero: TPunteroEntero;
BEGIN
  precioPan := 85;
  pEntero := @precioPan;
  writeln(precioPan);
END.
```



Dirección \$3\$12
identificada
como
precioPan

Valor (85) que
guarda
precioPan

Si **pEntero** contiene el valor \$3\$12, y esa es la dirección de memoria de la variable **precioPan**, se dice que **pEntero** apunta a **precioPan**

Operación new

Procedimiento por el que se **reserva un espacio de memoria dinámica**.

- No siempre es obligatorio reservar memoria para utilizar un puntero (ver ejemplos anteriores)
- **Sintaxis:**

```
new (variablePuntero) ;
```
- **Tras la instrucción** la variable puntero que se pasa como parámetro se deja apuntando a dicho espacio

new: ejemplo (1/2)

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;

VAR
  pRegistro: tPtrRegistro;

BEGIN
  ...
  new(pRegistro);
  ...
END.
```

Declaramos la variable estática (se reserva memoria en tiempo de compilación) puntero `pRegistro`



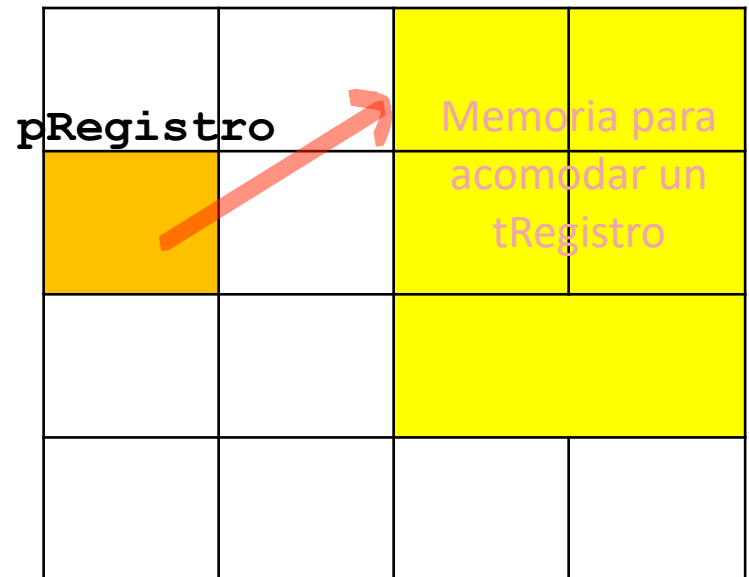
new: ejemplo (2/2)

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;

VAR
  pRegistro: tPtrRegistro;

BEGIN
  ...
  new(pRegistro);
  ...
END.
```

new reserva memoria (para guardar un registro de tipo **tRegistro**) en tiempo de ejecución (dinámica) y el puntero que le pasamos (**pRegistro**) se deja apuntando a esa porción de memoria reservada.



Operación **dispose**

Procedimiento por el que **se libera memoria dinámica**.

- No siempre es obligatorio reservar memoria para utilizar un puntero (ver ejemplos anteriores)
- **Sintaxis:**

```
dispose (variablePuntero) ;
```

Tras ejecutarse, el espacio de memoria apuntado por la variable puntero queda marcado como libre y está disponible para posteriores asignaciones de mi programa o de otros

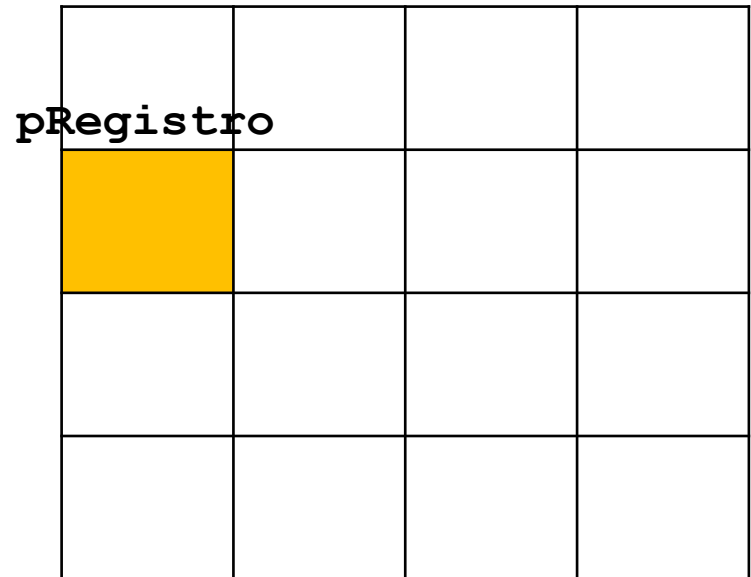
dispose: ejemplo (1/3)

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;

VAR
  pRegistro: tPtrRegistro;

BEGIN
  ...
  new(pRegistro);
  ... {Utilización de pRegistro} ...
  dispose(pRegistro);
  ...
END.
```

Declaramos la variable estática (se reserva memoria en tiempo de compilación) puntero `pRegistro`



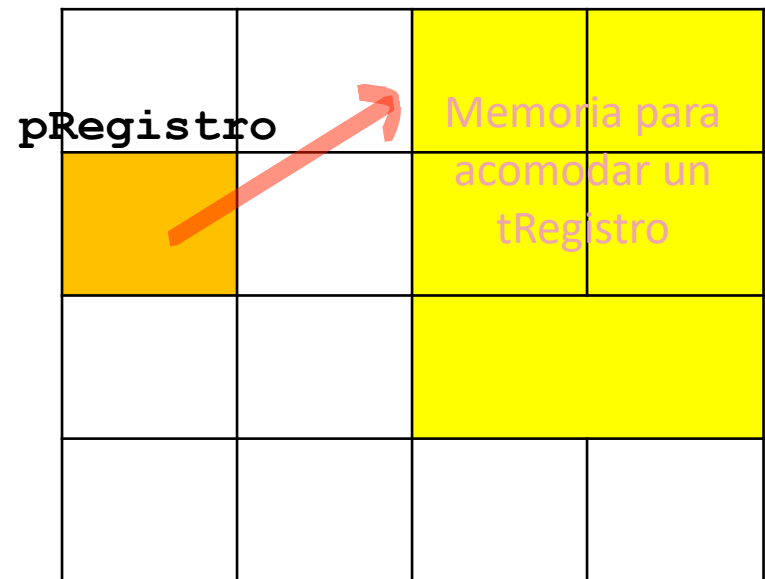
dispose: ejemplo (2/3)

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;

VAR
  pRegistro: tPtrRegistro;

BEGIN
  ...
  new(pRegistro);
  ... {Utilización de pRegistro} ...
  dispose(pRegistro);
  ...
END.
```

new reserva memoria (para guardar un registro de tipo **tRegistro**) en tiempo de ejecución (dinámica) y el puntero que le pasamos (**pRegistro**) se deja apuntando a esa porción de memoria reservada.



dispose: ejemplo (3/3)

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;

VAR
  pRegistro: tPtrRegistro;

BEGIN
  ...
  new(pRegistro);
  ... {Utilización de pRegistro} ...
  dispose(pRegistro);
  ...
END.
```

dispose libera la memoria apuntada por el puntero.



¿Por qué hay que liberar memoria?

- El *heap* tiene una capacidad finita de espacio reservado para crear memoria dinámica.
- No liberar memoria produce problemas graves que pueden afectar a otros programas:
 - Si el montículo y la pila se encuentran, pueden sobrescribirse (corromperse) partes de la memoria
 - Bloqueo del SO
 - Casos menos graves: *Memory leaks*
- Solución moderna:
 - Recolector basura → no en Pascal ☹️

¿Por qué hay que liberar memoria?



Your PC ran into a problem and needs to restart. We're just collecting some error info, and then we'll restart for you. (0% complete)

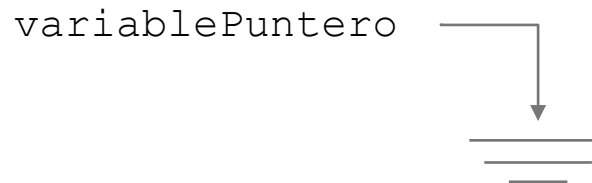
If you'd like to know more, you can search online later for this error: **MEMORY_MANAGEMENT**

Puntero nulo: **NIL**

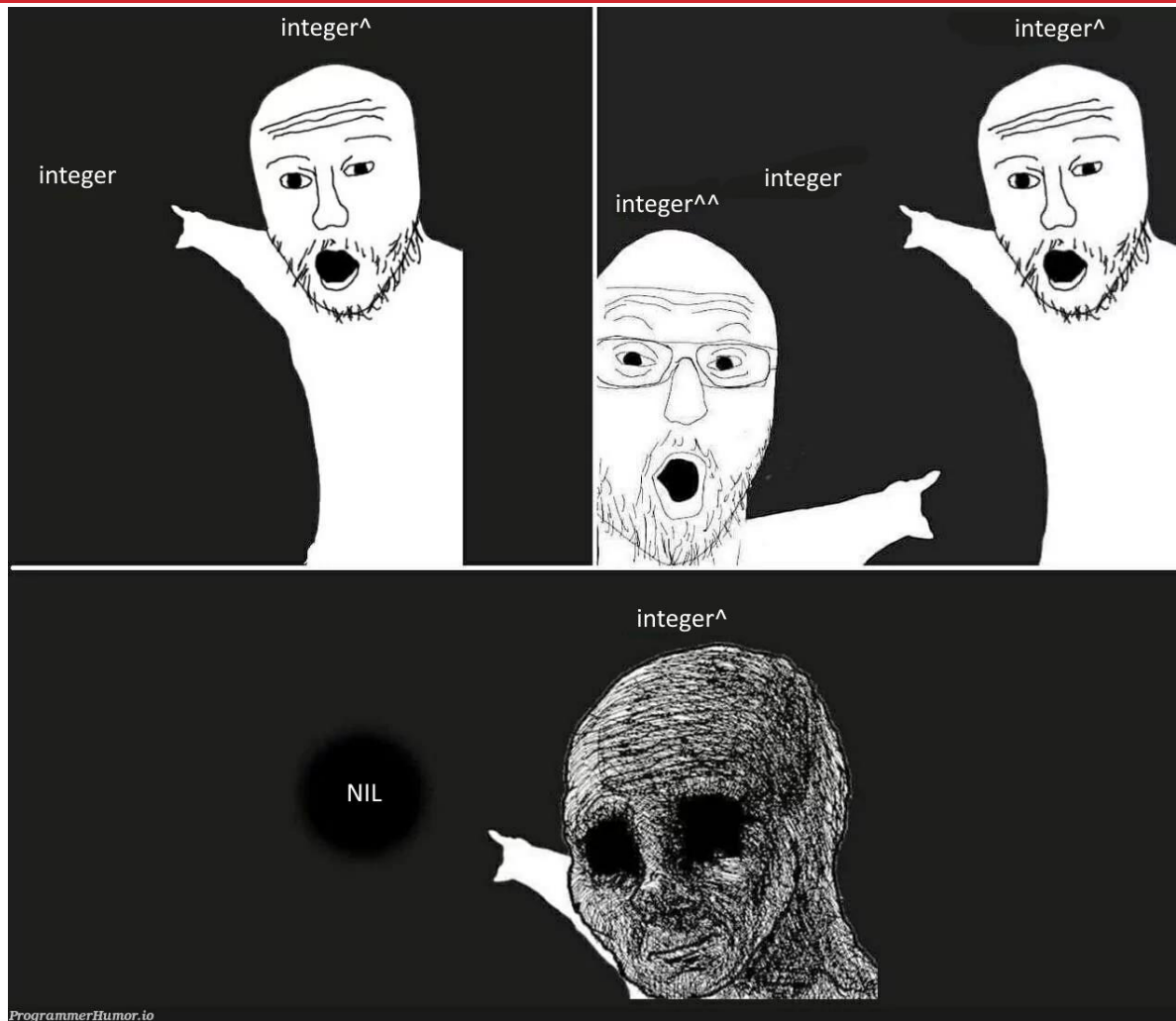
- **NIL** (Puntero nulo) es una constante de tipo puntero
- Una variable puntero puesta a **NIL**, indica que dicha variable no apunta a ninguna posición de memoria
 - Algo así como “apunta a un valor seguro”

- **Sintaxis:**

```
variablePuntero := NIL;
```



Puntero nulo: NIL



NIL: ejemplo (1/4)

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;

VAR
  pRegistro: tPtrRegistro;

BEGIN
  ...
  new(pRegistro);
  ... {Utilización de pRegistro} ...
  dispose(pRegistro);
  pRegistro := NIL;
  ...
END.
```

Declaramos la variable estática (se reserva memoria en tiempo de compilación) puntero `pRegistro`



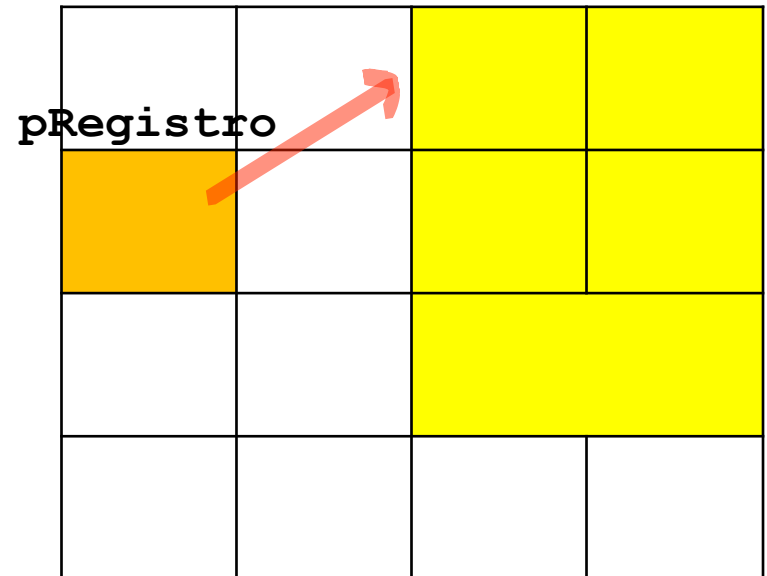
NIL: ejemplo (2/4)

new reserva memoria (para guardar un registro de tipo **tRegistro**) en tiempo de ejecución (dinámica) y el puntero que le pasamos (**pRegistro**) se deja apuntando a esa porción reservada .

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;

VAR
  pRegistro: tPtrRegistro;

BEGIN
  ...
  new(pRegistro);
  ... {Utilización de pRegistro} ...
  dispose(pRegistro);
  pRegistro := NIL;
  ...
END.
```



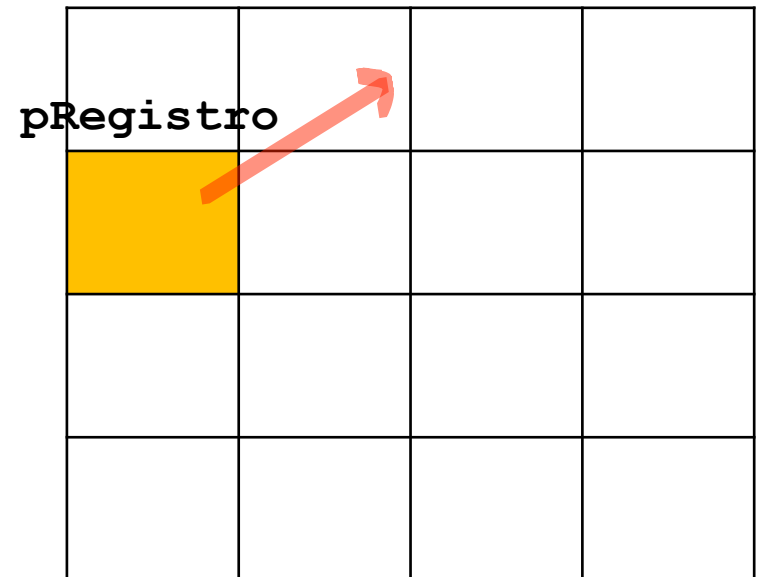
NIL: ejemplo (3/4)

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;

VAR
  pRegistro: tPtrRegistro;

BEGIN
  ...
  new(pRegistro);
  ... {Utilización de pRegistro} ...
  dispose(pRegistro);
  pRegistro := NIL;
  ...
END.
```

dispose libera la memoria
apuntada por el puntero.



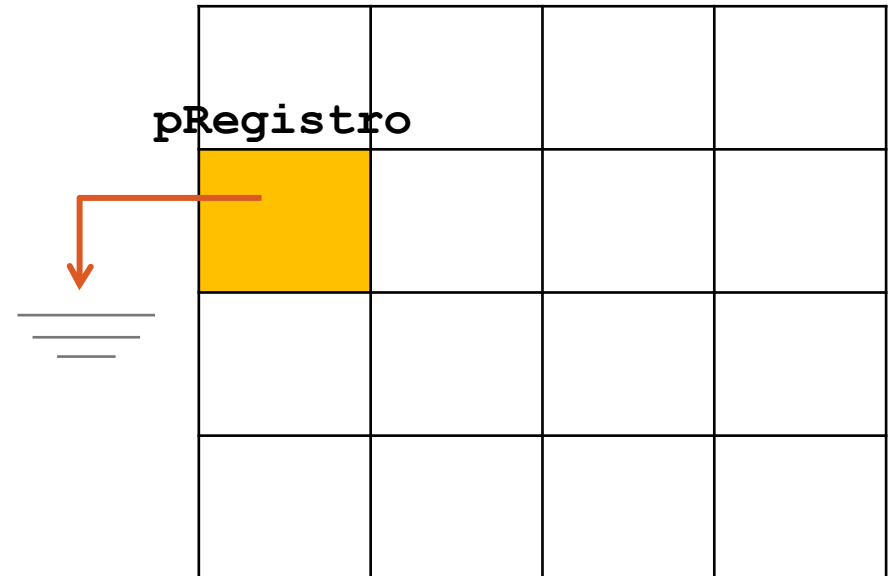
NIL: ejemplo (4/4)

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;

VAR
  pRegistro: tPtrRegistro;

BEGIN
  ...
  new(pRegistro);
  ... {Utilización de pRegistro} ...
  dispose(pRegistro);
  pRegistro := NIL;
  ...
END.
```

asignar **NIL** al puntero, indica que no apunta a ninguna dirección de memoria.



Efecto de omitir el **dispose**

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;

VAR
  pRegistro: tPtrRegistro;

BEGIN
  ...
  new(pRegistro);
  ... {Utilización de pRegistro} ...
  pRegistro := NIL;
  ...
END.
```

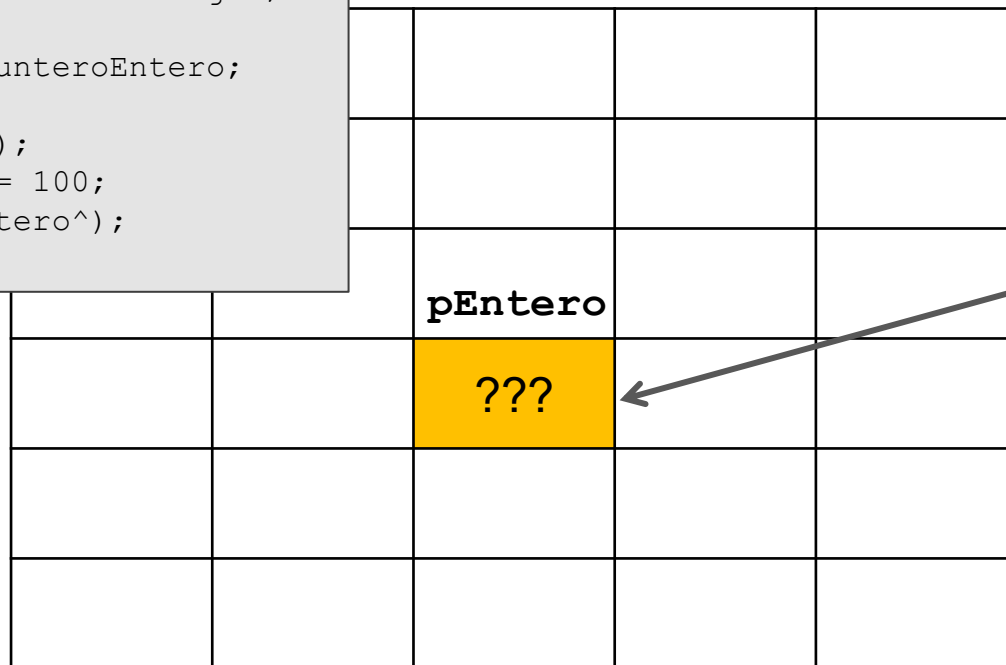
Si, en el código anterior, se omite la liberación de memoria (mediante **dispose**), el resultado es que se mantiene reservada una porción de memoria innecesaria, ya que se ha desapuntado con la asignación de NIL a pRegistro.

pRegistro



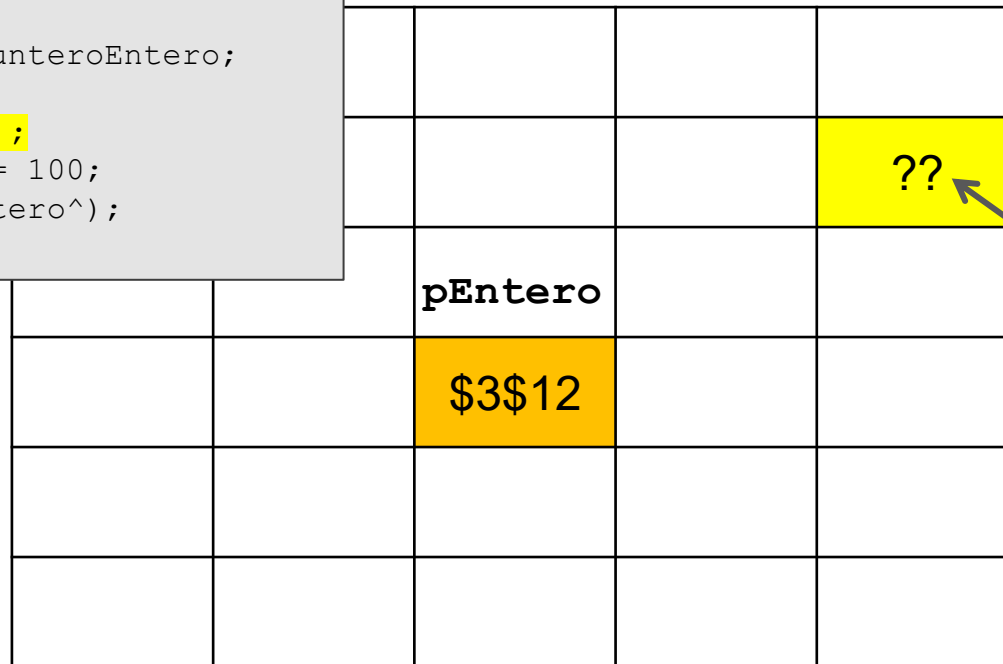
Punteros: inicialización

```
TYPE
  TPrecio = integer;
  TPunteroEntero = ^integer;
VAR
  pEntero: TPunteroEntero;
BEGIN
  new(pEntero);
  pEntero^ := 100;
  writeln(pEntero^);
END.
```



Punteros: inicialización

```
TYPE
  TPrecio = integer;
  TPunteroEntero = ^integer;
VAR
  pEntero: TPunteroEntero;
BEGIN
  new(pEntero);
  pEntero^ := 100;
  writeln(pEntero^);
END.
```

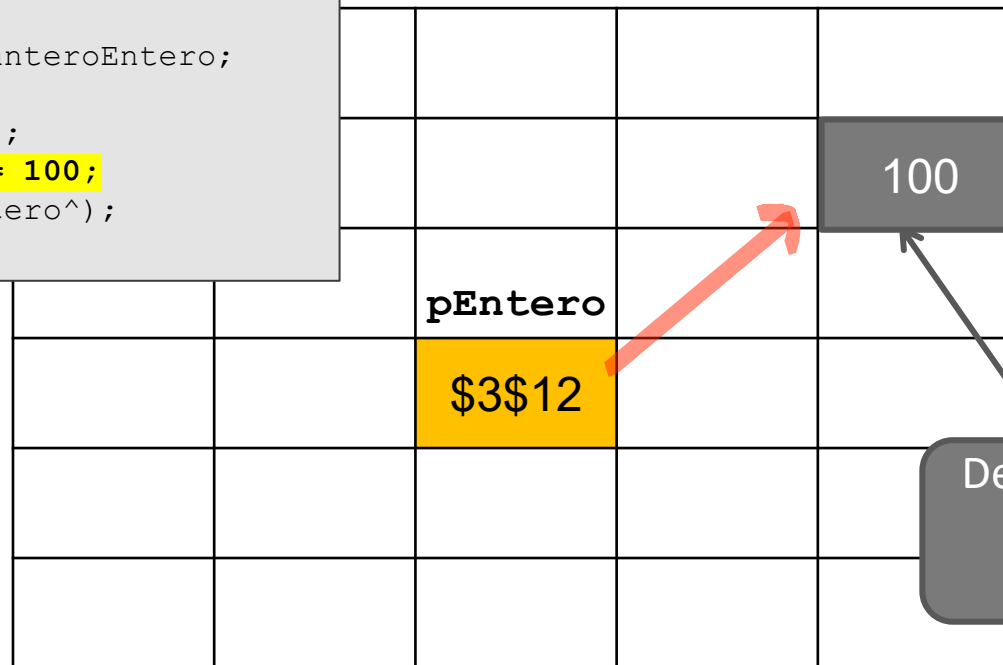


Dirección
\$3\$12 creada
por new

Valor basura

Punteros: inicialización

```
TYPE
  TPrecio = integer;
  TPunteroEntero = ^integer;
VAR
  pEntero: TPunteroEntero;
BEGIN
  new(pEntero);
  pEntero^ := 100;
  writeln(pEntero^);
END.
```



Operador[^]: Ejemplo (1/3)

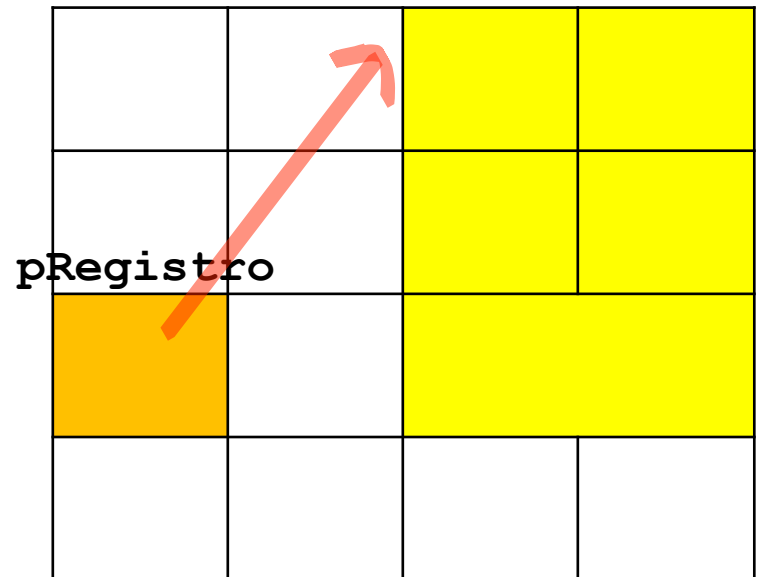
```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;
VAR
  pRegistro: tPtrRegistro;
BEGIN
  ...
  new(pRegistro);
  pRegistro^.iniciales := 'ASM';
  pRegistro^.identificacion := 2345;
  ...
END.
```



Operador[^]: Ejemplo (2/3)

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;
VAR
  pRegistro: tPtrRegistro;
BEGIN
  ...
  new(pRegistro);
  pRegistro^.iniciales := 'ASM';
  pRegistro^.identificacion := 2345;
  ...
END.
```

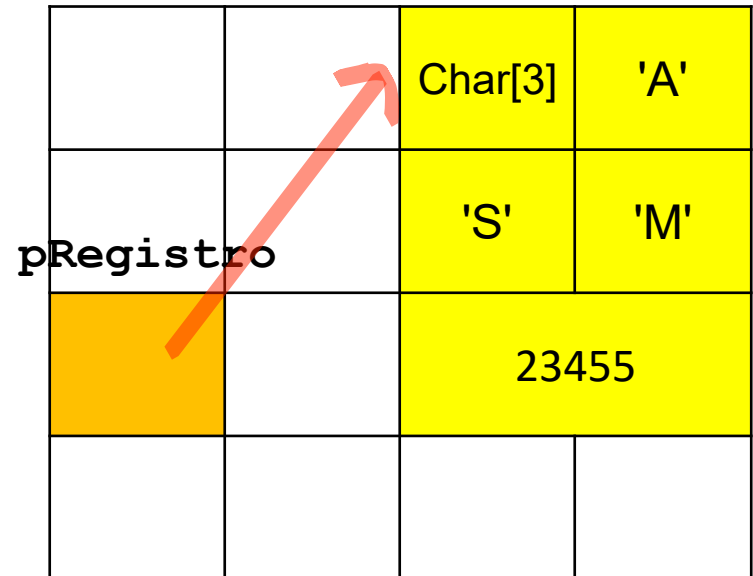
Reservamos en tiempo de ejecución un bloque de memoria para un registro.



Operador^: Ejemplo (3/3)

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;
VAR
  pRegistro: tPtrRegistro;
BEGIN
  ...
  new(pRegistro);
  pRegistro^.iniciales := 'ASM';
  pRegistro^.identificacion := 2345;
  ...
END.
```

Apuntamos con el puntero el bloque de memoria del registro. Tenemos accesible la información del registro a través del puntero.



Operaciones con punteros

- Operaciones permitidas:
 - Asignación ($:=$)
 - Comparación ($=$ y $<>$)
- Para realizar estas operaciones los operandos han de ser **punteros a variables del mismo** tipo o **NIL**.
- Los punteros no se pueden leer ni escribir directamente
 - No está permitido utilizar `write/read/etc.`

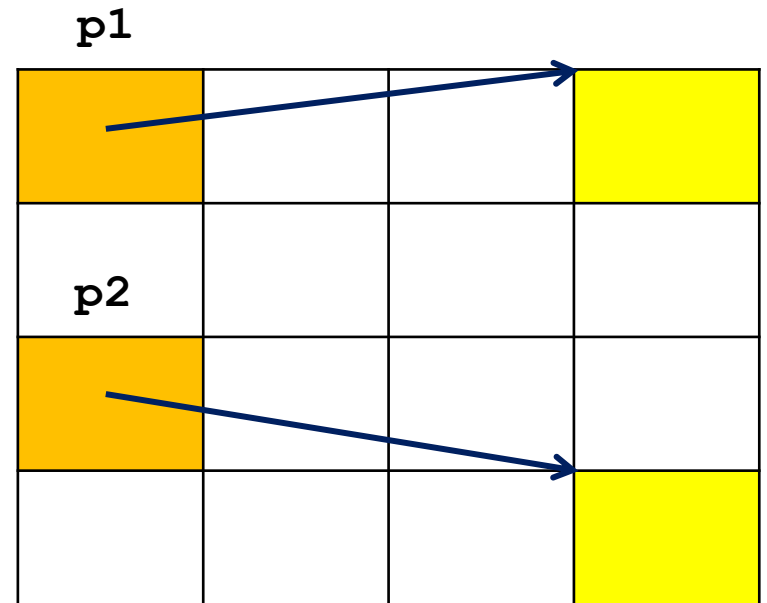
Operaciones: ejemplos (1)

```
TYPE
  ptrACharacter = ^char;
VAR
  p1, p2: ptrACharacter;
BEGIN
  ...
END.
```

p1			
??			
p2			
??			

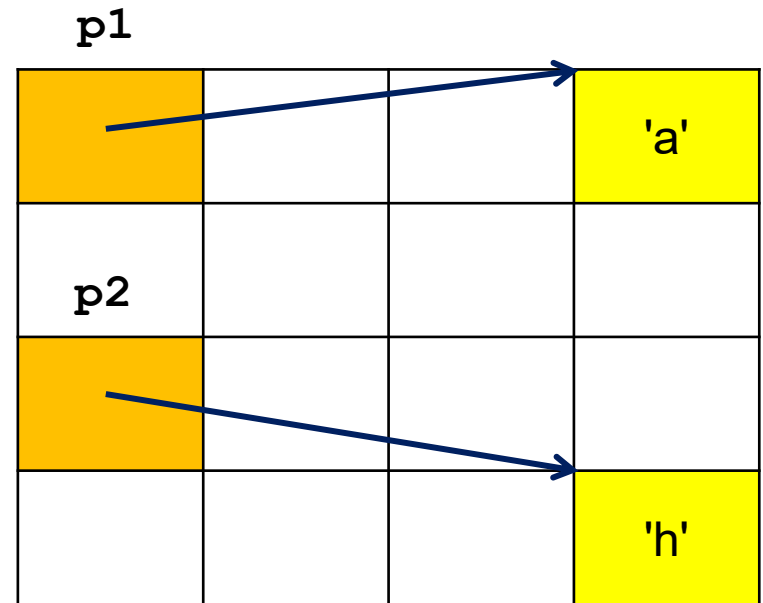
Operaciones: ejemplos (1)

```
TYPE
  ptrACharacter = ^char;
VAR
  p1, p2: ptrACharacter;
BEGIN
  new(p1);
  new(p2);
  ...
END.
```



Operaciones: ejemplos (1)

```
TYPE
  ptrACharacter = ^char;
VAR
  p1,p2: ptrACharacter;
BEGIN
  new(p1);
  new(p2);
  p1^:='a';
  p2^:='h';
  ...
END.
```

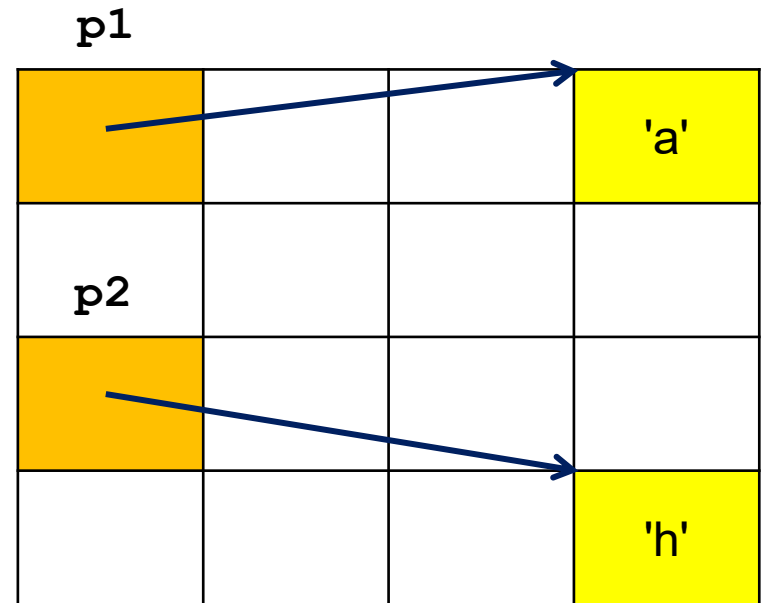


Operaciones: ejemplos (1)

```
TYPE
  ptrACharacter = ^char;
VAR
  p1,p2: ptrACharacter;
BEGIN
  new(p1);
  new(p2);
  p1^:='a';
  p2^:='h';
  ...
END.
```

En este punto...

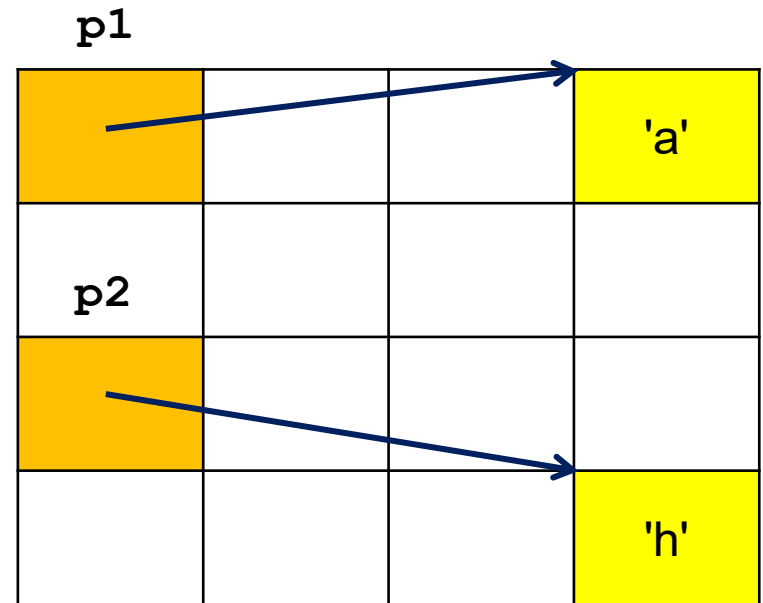
p1 = p2 --> TRUE OR FALSE?
p1 <> p2 --> TRUE OR FALSE?
p1^ = p2^ --> TRUE OR FALSE?



Operaciones: ejemplos (1)

```
TYPE
  ptrACharacter = ^char;
VAR
  p1,p2: ptrACharacter;
BEGIN
  new(p1);
  new(p2);
  p1^:='a';
  p2^:='h';
  ...
END.
```

p1 = p2 --> FALSE
p1 <> p2 --> TRUE
p1^ = p2^ --> FALSE

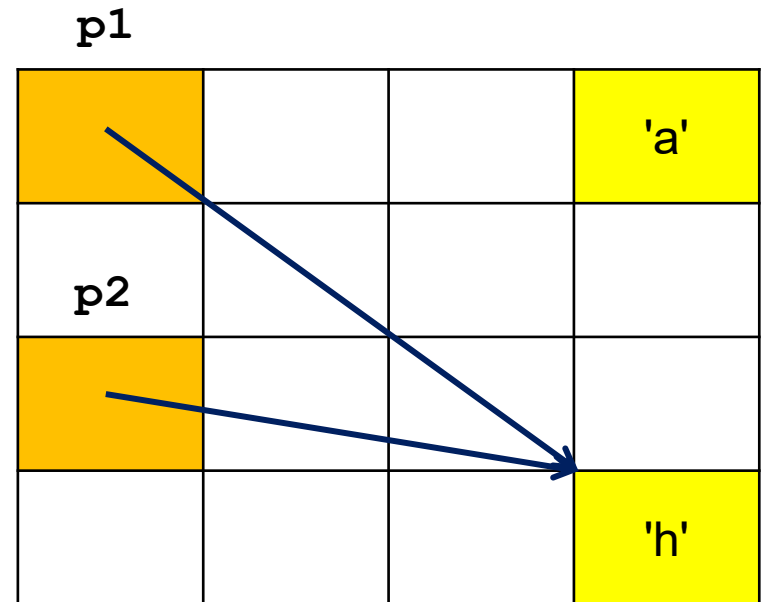


Operaciones: ejemplos (2)

```
TYPE
  ptrACharacter = ^char;
VAR
  p1,p2: ptrACharacter;
BEGIN
  new(p1);
  new(p2);
  p1^:='a';
  p2^:='h';
  p1 := p2;
  ...
END.
```

Tras esta asignación...

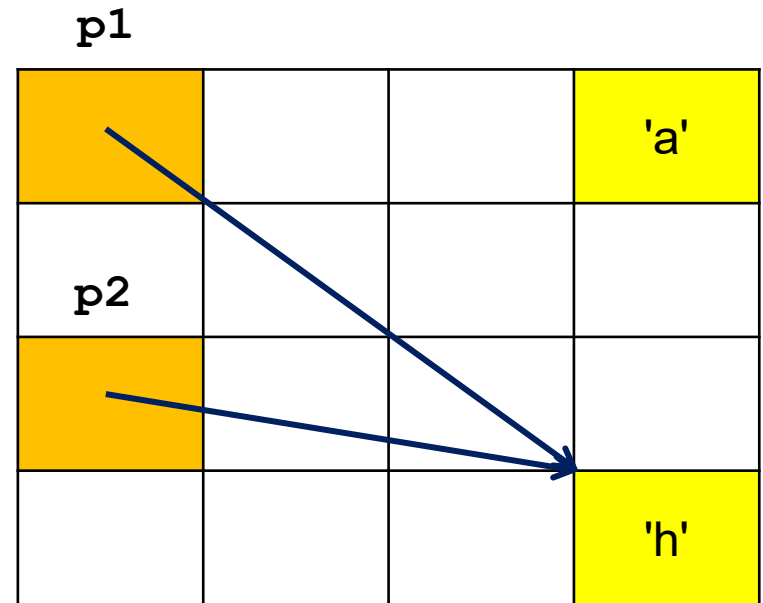
p1 = p2 --> ?
p1 <> p2 --> ?
p1^ = p2^ --> ?



Operaciones: ejemplos (2)

```
TYPE
  ptrACharacter = ^char;
VAR
  p1,p2: ptrACharacter;
BEGIN
  new(p1);
  new(p2);
  p1^:='a';
  p2^:='h';
  p1 := p2;
  ...
END.
```

$p1 = p2 \rightarrow \text{TRUE}$
 $p1 \neq p2 \rightarrow \text{FALSE}$
 $p1^ = p2^ \rightarrow \text{TRUE}$

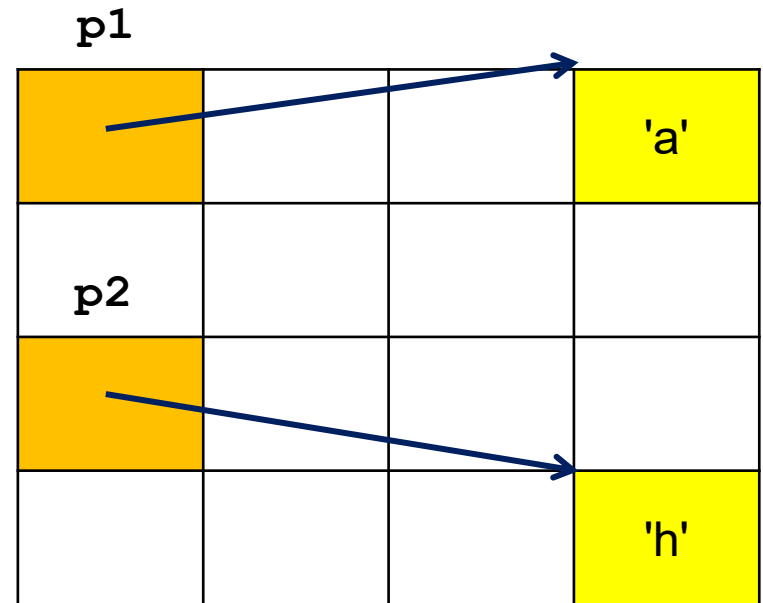


Operaciones: ejemplos (3)

```
TYPE
  ptrACharacter = ^char;
VAR
  p1,p2: ptrACharacter;
BEGIN
  new(p1);
  new(p2);
  p1^:='a';
  p2^:='h';
  p1^ := p2^;
  ...
END.
```

Tras esta asignación...

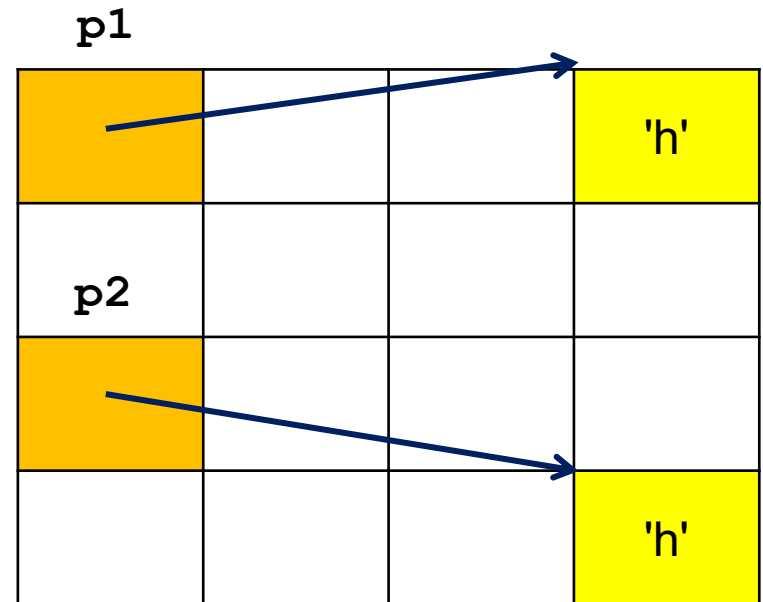
p1 = p2 --> TRUE OR FALSE?
p1 <> p2 --> TRUE OR FALSE?
p1^ = p2^ --> TRUE OR FALSE?



Operaciones: ejemplos (3)

```
TYPE
  ptrACharacter = ^char;
VAR
  p1,p2: ptrACharacter;
BEGIN
  new(p1);
  new(p2);
  p1^:='a';
  p2^:='h';
  p1^ := p2^;
  ...
END.
```

p1 = p2 --> FALSE
p1 <> p2 --> TRUE
p1^ = p2^ --> TRUE



Procedimientos y funciones

- Los punteros...
 - Pueden ser parámetros de un subprograma, tanto por valor como por referencia
 - Se pueden devolver como resultado de una función.
- Aunque f sea una función que devuelva un puntero a un registro, no se permite:

```
f (parámetrosReales) ^ .campoDelReg;
```

- La sintaxis correcta es:

```
varPuntero:=f (parámetrosReales) ;
```

- y posteriormente sí se puede acceder al campo:

```
varPuntero ^ .campoDelReg;
```

Paso por valor de punteros

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;
VAR
  pRegistro: tPtrRegistro;

PROCEDURE miProc(dato: tPtrRegistro);
BEGIN
  ...
END;

BEGIN
  new(pRegistro);
  pRegistro^.iniciales := 'ASM';
  pRegistro^.identificacion := 23455;
  miProc(pRegistro);
  ...
END.
```

El procedimiento recoge una copia del puntero, que apunta al mismo lugar que el original.



Cualquier cambio en esa información se verá reflejada en la información apuntada por el puntero original.

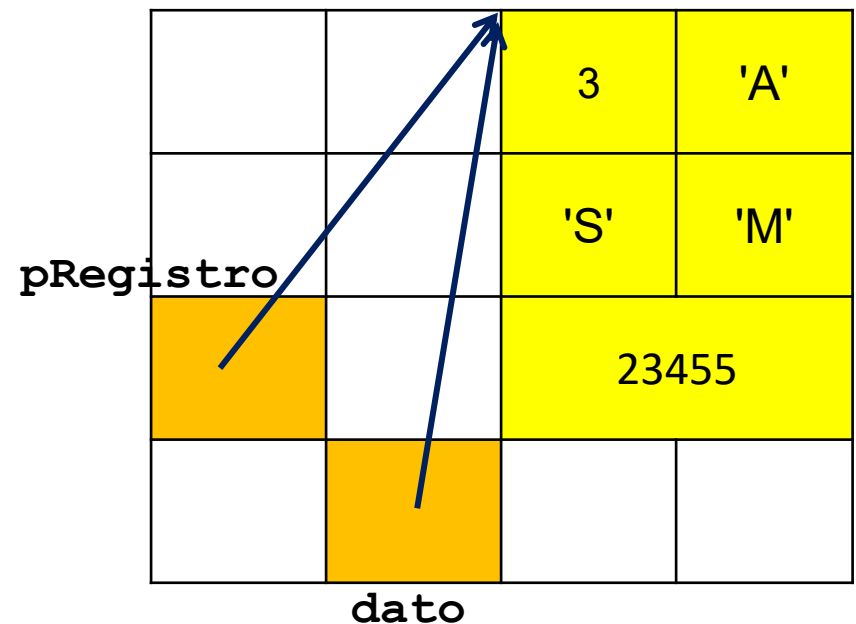
Paso por valor de punteros

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;
VAR
  pRegistro: tPtrRegistro;

PROCEDURE miProc(dato: tPtrRegistro);
BEGIN
  ...
END;

BEGIN
  new(pRegistro);
  pRegistro^.iniciales:='ASM';
  pRegistro^.identificacion:=23455;
  miProc(pRegistro);
  ...
END.
```

Cualquier cambio en esa información se verá reflejada en la información apuntada por el puntero original.



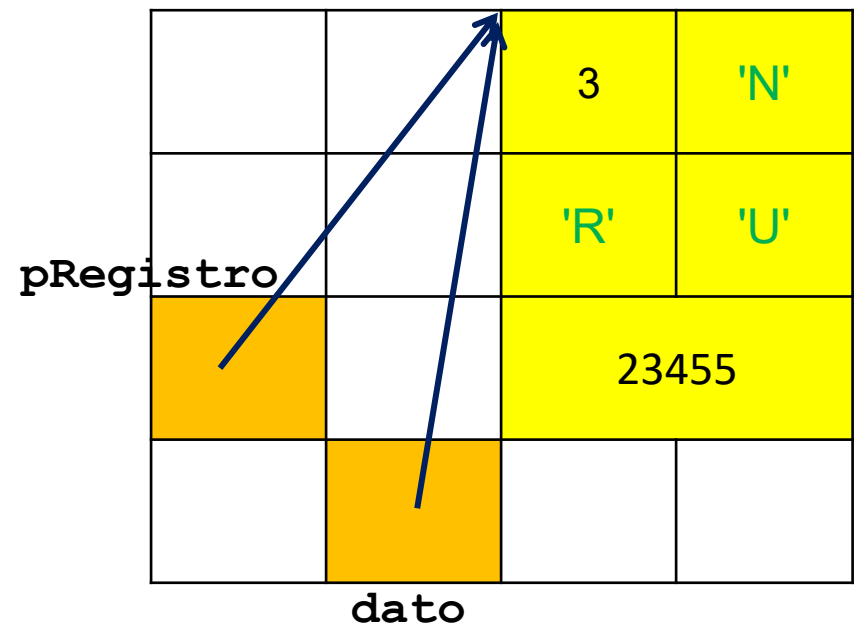
Paso por valor de punteros

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;
VAR
  pRegistro: tPtrRegistro;

PROCEDURE miProc(dato: tPtrRegistro);
BEGIN
  dato^.iniciales := 'NRU';
END;

BEGIN
  new(pRegistro);
  pRegistro^.iniciales := 'ASM';
  pRegistro^.identificacion := 23455;
  miProc(pRegistro);
  ...
END.
```

Cualquier cambio en esa información se verá reflejada en la información apuntada por el puntero original.



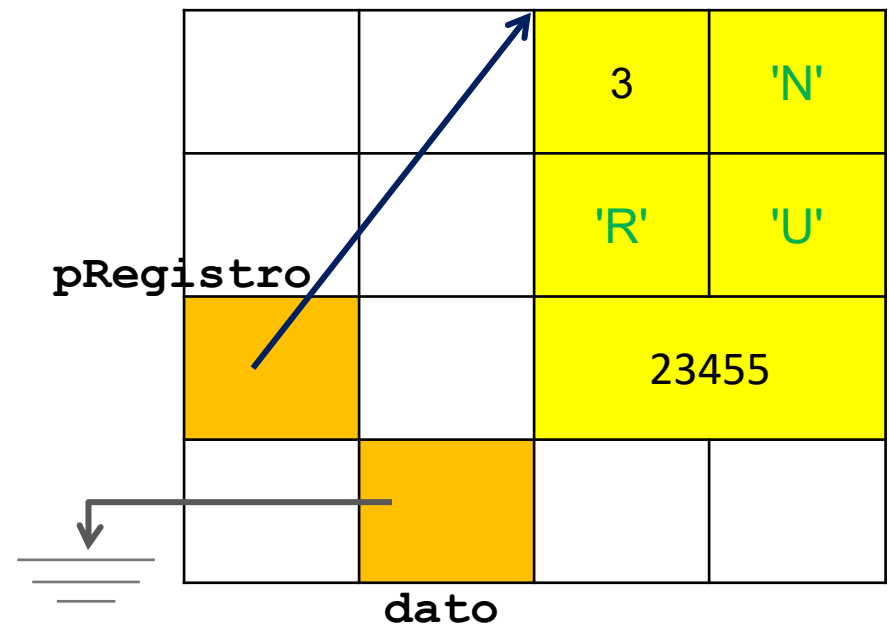
Paso por valor de punteros

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;
VAR
  pRegistro: tPtrRegistro;

PROCEDURE miProc(dato: tPtrRegistro);
BEGIN
  dato^.iniciales := 'NRU';
  dato := NIL;
END;

BEGIN
  new(pRegistro);
  pRegistro^.iniciales := 'ASM';
  pRegistro^.identificacion := 23455;
  miProc(pRegistro);
  ...
END.
```

Cualquier cambio en esa información se verá reflejada en la información apuntada por el puntero original **pero no en el puntero original**



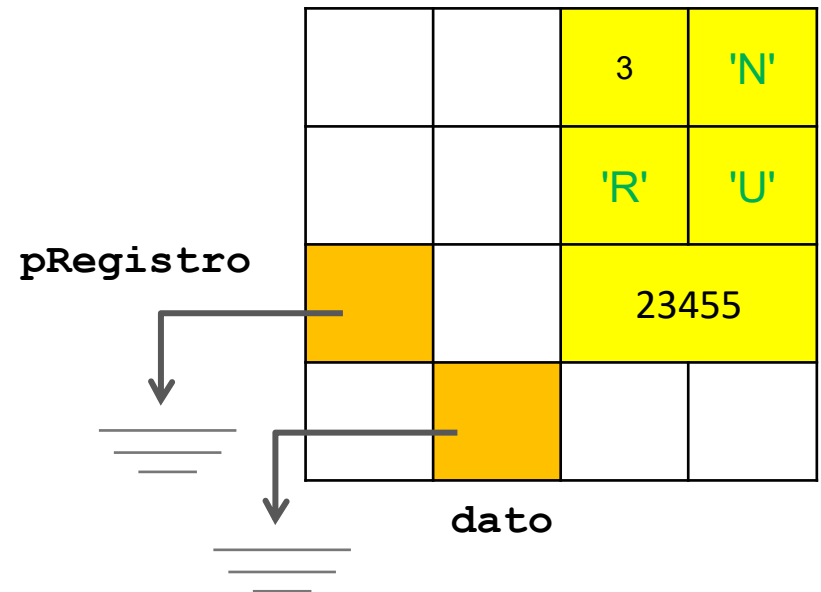
Paso por referencia de punteros

```
TYPE
  tIniciales = string[3];
  tRegistro = RECORD
    iniciales: tIniciales;
    identificacion: Integer;
  END;
  tPtrRegistro = ^tRegistro;
VAR
  pRegistro: tPtrRegistro;

PROCEDURE miProc(var dato: tPtrRegistro);
BEGIN
  dato^.iniciales := 'NRU';
  dato := NIL;
END;

BEGIN
  new(pRegistro);
  pRegistro^.iniciales := 'ASM';
  pRegistro^.identificacion := 23455;
  miProc(pRegistro);
  ...
END.
```

Cualquier cambio en esa información se verá reflejado en la información apuntada por el puntero original y en el propio puntero



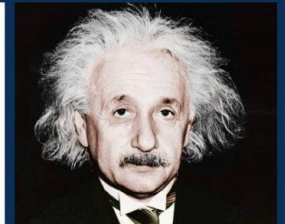
Resumen

- Los punteros son una herramienta **FUNDAMENTAL** porque permiten gestionar memoria dinámicamente (en tiempo de ejecución): crear, modificar y liberar.
- **variablePuntero** (lo que apunta) \neq **variablePuntero^** (lo apuntado)
- Para crear información dinámicamente usar **new**.
- Después de llamar a **new**, el valor al que apunta el puntero es indefinido. Para definirlo es necesario asignarle un valor.
- Es imprescindible liberar el espacio de memoria dinámica mediante **dispose** cuando no se vaya a utilizar más.
- El valor **nil**, válido para punteros de cualquier tipo base, permite apuntar a un “lugar seguro”.

Pointers



Pointers to
pointers



Pointers to
pointers to
pointers



Pointers to
pointers to
pointers to
pointers

