Datos puntero y estructuras dinámicas de datos

Lección 7

Índice

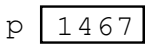
- Datos puntero y datos dinámicos
- Estructuras de datos recursivas: representación mediante punteros y datos dinámicos
- Punteros y datos dinámicos en C++

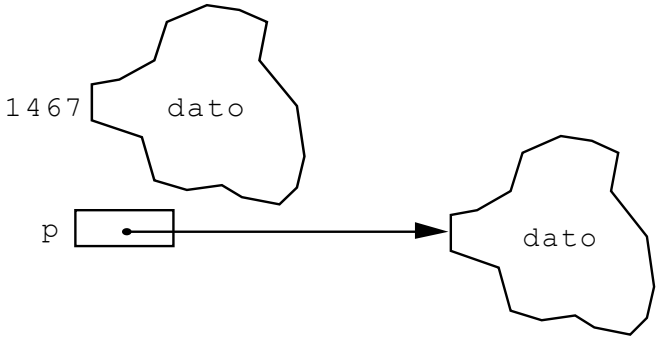
LECTURA MUY RECOMENDADA:
http://cslibrary.stanford.edu/102/PointersAndMemory.pdf

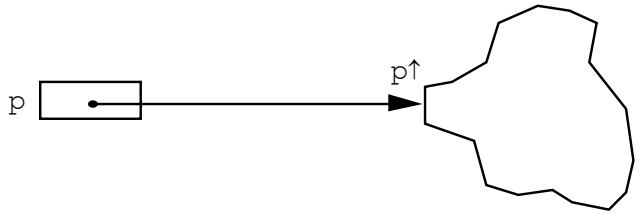
Datos estáticos vs Datos dinámicos

- **datos estáticos (datos creados en memoria estática** o *stack memory*)**:**
  - los almacenados en constantes, variables (o parámetros) declaradas al principio de los algoritmos (su declaración les da un nombre y un tipo)
  - el "nombre" identifica al dato en memoria (zona de memoria reservada para almacenarlo)
  - existen en memoria sólo desde su declaración hasta el final del bloque de código en el que son declaradas (*ámbito*), y al acabar la ejecución del bloque se devuelve al sistema la memoria que ocupaban
- **datos dinámicos (datos creados en memoria dinámica** o *heap memory*)**:**
  - son creados (se les reserva memoria) dinámicamente durante la ejecución, cuando sean necesarios, y destruidos cuando ya no hacen falta (se libera la memoria ocupada)

    → uso EFICIENTE de la memoria disponible
  - no tienen nombre (identificador), se referencian y manipulan mediante punteros
    - Un puntero es un dato cuyo valor es la dirección en memoria de otro dato



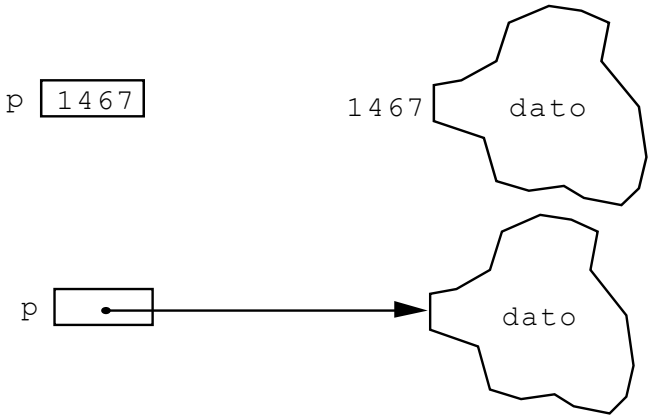


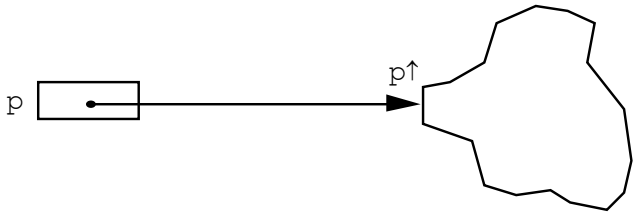


p↑ es "el dato apuntado por el puntero p"

Datos estáticos vs Datos dinámicos

- Punteros:
  - Un puntero es un dato cuyo valor es la dirección en memoria de otro determinado dato (la dirección almacenada será transparente al programador)





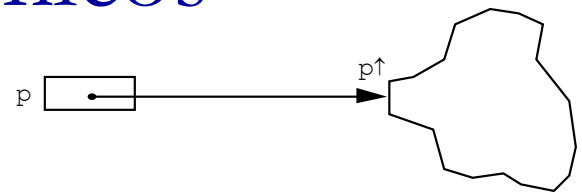
p↑ es "el dato apuntado por el puntero p"

NO CONFUNDIR el PUNTERO con el DATO apuntado por el puntero:

- Una cosa es **declarar el puntero** (se reserva memoria solo para el puntero)
- Otra cosa **es reservar memoria dinámica** para un dato
- Para usar el DATO creado en memoria dinámica habrá **que tener un puntero** que **apunte** a dicho dato

Datos estáticos vs Datos dinámicos

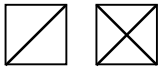
- Declaración de punteros:



  - Crearemos *tipos puntero* (punteros fuertemente tipados):
    - Cada tipo de punteros servirá para apuntar a *un* tipo concreto de datos
    - Podremos declarar tipos punteros para cualquier tipo predefinido o definido con anterioridad
    - Tendremos variables y parámetros de tipos puntero

{definición de tipos y variables punteros en pseudocódigo:}
Tipos tx = ...
...
tp_punt_a_tx = ↑tx
...
Variables p,q: tp_punt_a_tx p [?] q [?]

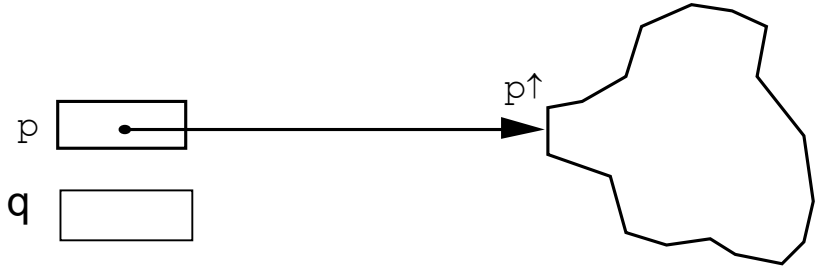
{Declarar la variable puntero NO le asigna un valor}

  - Valor especial, válido para cualquier puntero de cualquier tipo: ***NIL*** ***{Representación gráfica de puntero con valor NIL:***  ***}***
    - Significa que el puntero no apunta a ningún sitio
    - Intentar acceder al dato apuntado por un puntero con valor NIL provocará un error en tiempo de ejecución

q:=NIL; q↑ {ERROR!!!!!!}

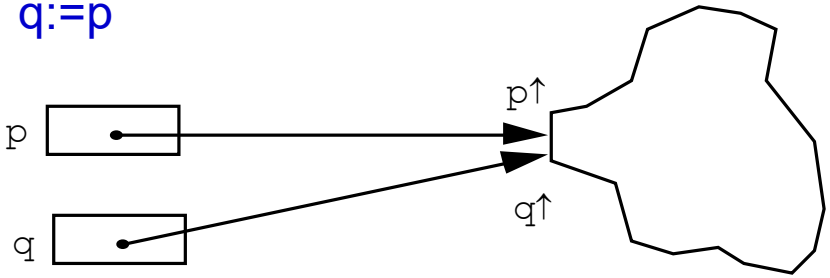
Datos estáticos vs Datos dinámicos

- Operaciones con punteros:
  - Asignación (:=) entre punteros del mismo tipo:
    - hace que apunten al mismo dato en memoria
    - a cualquier puntero se le puede asignar el valor NIL *{q:=NIL}*
  - Comparación (=) entre punteros del mismo tipo:
    - comprueba si apuntan al mismo dato en memoria
      - También será cierto si ambos punteros tienen valor NIL
    - También disponible operación ≠



{Si hacemos: }

q:=p



{DESPUES:
p↑ y q↑ se pueden usar para acceder al mismo dato (lo comparten).
Si comprobamos p=q será cierto porque ambos están apuntando al mismo dato, en cualquier otro caso será falso, salvo que ambos punteros tengan el valor NIL }

Datos estáticos vs Datos dinámicos

- Trabajando con datos dinámicos:
  - **Creación de dato dinámico: reserva de memoria**

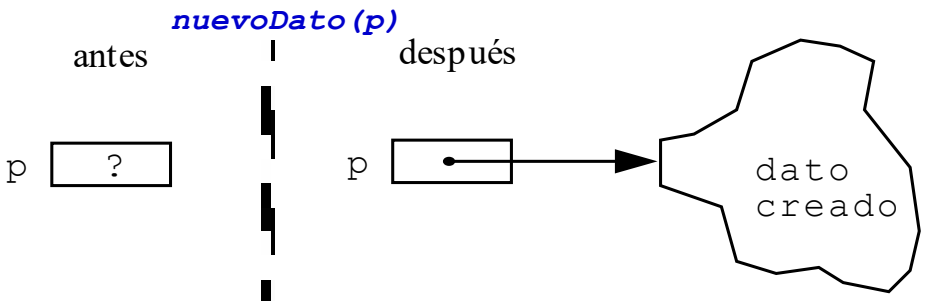
`nuevoDato(p)`

  - Esta instrucción reserva espacio en memoria dinámica para alojar un dato del tipo de los apuntados por el puntero dado
  - Devuelve la dirección de memoria reservada, dejándola en el puntero (el puntero queda apuntando al nuevo dato creado)

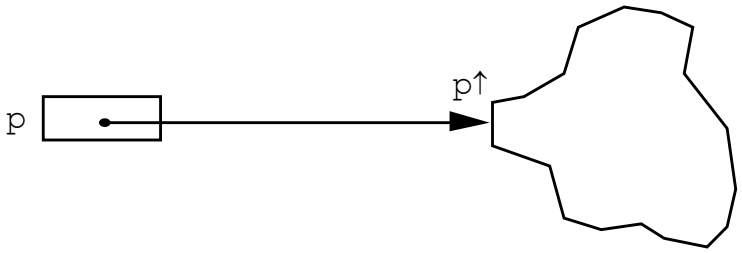
```
Tipos  tx = ...
       ...
       tp_punt_a_tx = ↑tx
       ...
Variables
p,q: tp_punt_a_tx

{Declarar el puntero NO le asigna un valor}
```

```
{Declarar el
puntero NO
reserva memoria
para el dato
apuntado}
```



  - **Manipulación** del dato creado dinámicamente



p↑ es el dato apuntado por el puntero p

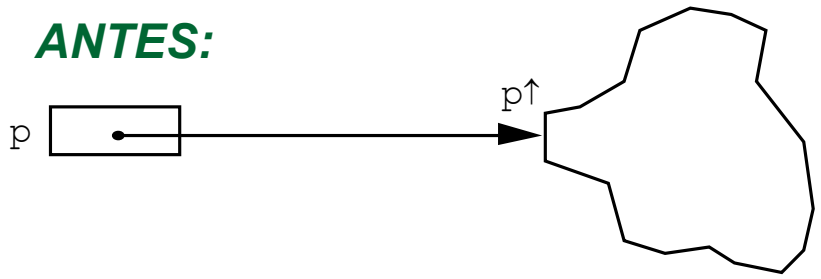
Datos estáticos vs Datos dinámicos

- Trabajando con datos dinámicos:
  - El programador **es responsable de** no ocupar más memoria de la necesaria, y de no agotar la memoria disponible con *basura (= memoria dinámica ocupada pero inaccesible)*
  - **Liberación de la memoria** dinámica ocupada cuando ya no sea necesaria (acción opuesta a `nuevoDato`)

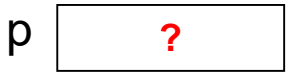
`disponer(p)`

- Esta instrucción libera la memoria ocupada por el dato apuntado por p, y deja p con valor indefinido (no está controlado y por tanto no utilizarlo)

ANTES:



DESPUES de disponer(p):



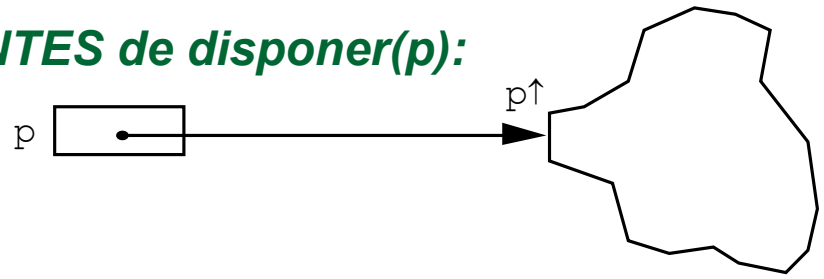
p↑ es el dato apuntado por el puntero p

Datos estáticos vs Datos dinámicos

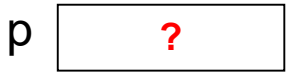
Liberación de la memoria **`disponer(p)`**

- problemas?

ANTES de disponer(p):



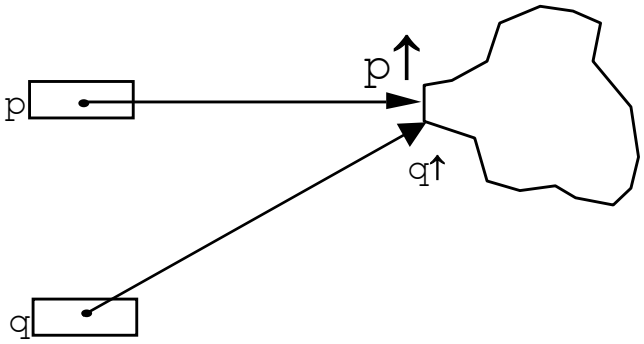
DESPUES de disponer(p):



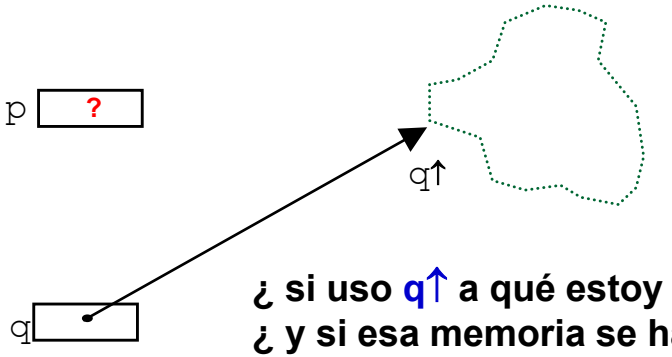
¿ si uso p↑ a qué parte de la memoria estoy accediendo ?

- Y si .....

ANTES de disponer(p):

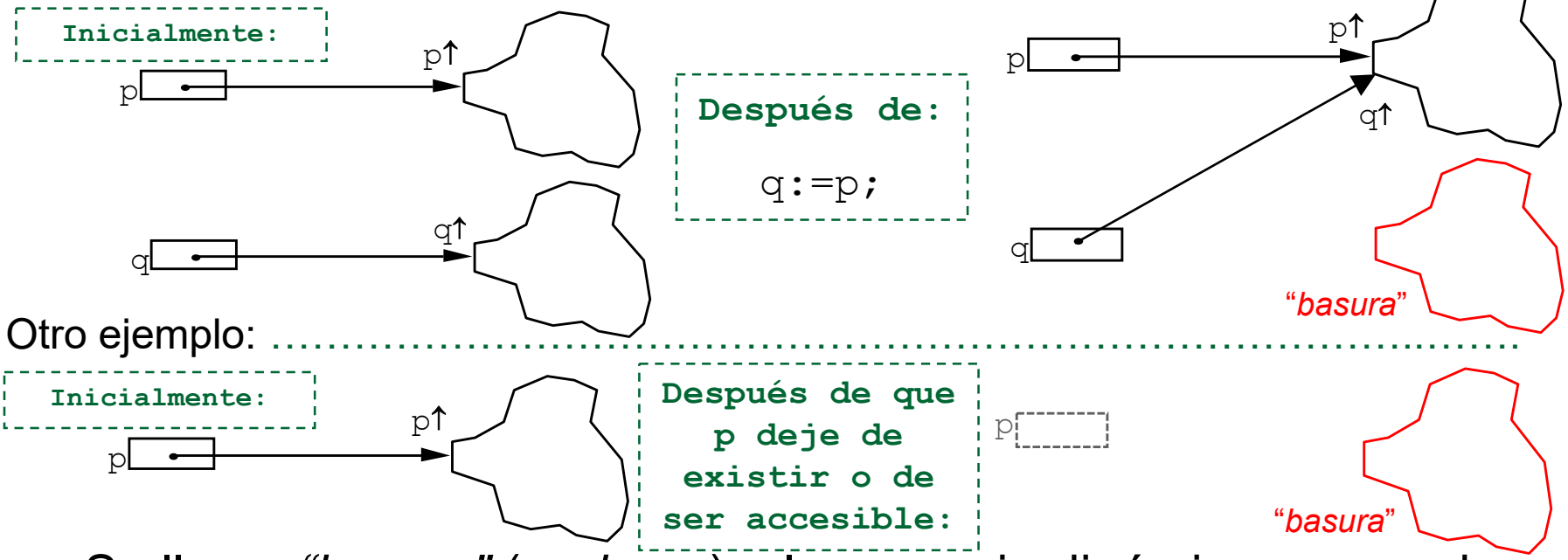


DESPUES de disponer(p):



¿ si uso q↑ a qué estoy accediendo ?
¿ y si esa memoria se ha reservado ya para otros datos ?

Recolección de basura



- Se llama *"basura"* (*garbage*) a la memoria dinámica reservada para datos, pero que ha quedado inaccesible (ningún puntero *"apunta"* a ella)
- Se llama "*recolección de basura*" a la recuperación, manual o automática, de la memoria dinámica
  - La instrucción **`disponer`** es la que permite al programador responsabilizarse de la recolección (manual) de basura (**debe hacerlo antes de que pase a ser basura**).
  - Hay lenguajes que permiten la recolección manual (por ejemplo: C, C++, Ada, y el seudocódigo) y otros que no (ejemplo: Java).

Datos estáticos vs Datos dinámicos

- Pautas a seguir:

a) *Manipular punteros no inicializados puede llevar a leer o escribir en zonas de memoria no deseadas, y a errores de ejecución muy difíciles de detectar*
   - No dejar nunca variables punteros con valor no controlado o indefinido:
     - Inicializar siempre los punteros tras declararlos
     - Tras utilizar un puntero con *disponer* para liberar memoria, asignarle valor antes de volver a usarlo
   - Asegurarse de que todo puntero tiene un valor controlado (apunta a un dato útil) o tiene valor NIL

b) *Intentar acceder al dato apuntado por un puntero que tiene valor NIL provocará un error de ejecución (runtime)*
   - No utilizar nunca un puntero para acceder al dato apuntado, sin comprobar antes que el puntero no tiene el valor NIL

c) *No utilizar más punteros de los necesarios para apuntar a cada dato*
   - Compartir datos es una de las ventajas del uso de punteros
   - Tener mucho cuidado cuando varios punteros comparten un dato

d) *Dibujar, dibujar, dibujar…*
   - Dibujar los punteros y las estructuras de datos con las que trabajamos, y hacerlo lo más fielmente posible (y paso a paso!), nos ayudará a pensar, escribir el código, y encontrar errores

Usos de punteros y memoria dinámica

- Usaremos punteros para:
  - Crear datos dinámicamente, y manipularlos
  - Definir y utilizar estructuras de datos dinámicas:
    - Estructuras que pueden variar (en tamaño y forma) durante la ejecución
  - Definir y manipular estructuras recursivas:
    - Estructuras que se definen en base a si mismas

Estructuras de Datos Recursivas

- Estructuras recursivas serían aquellas que en su definición incluyesen un dato de su mismo tipo (directa o indirectamente)

```
tipo cadena = registro
                esVacía:booleano
                {y por si no es vacía...}
                dato:carácter {el 1º de la cadena}
                resto:cadena (*)
              freg
```

  - Problema: cómo sabría el compilador o interprete cuánta memoria se deberá asignar a una variable de ese tipo?

  (*) ¡No lo permitimos!

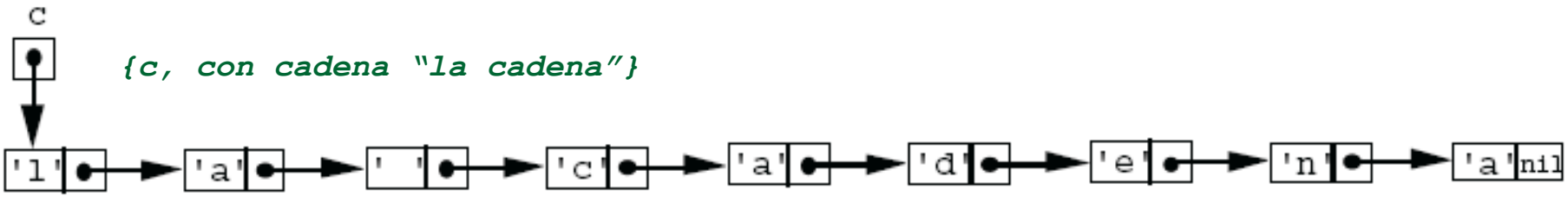
Estructuras de Datos Recursivas

- Se definen utilizando punteros a diferentes partes de la estructura
  - El tamaño y forma de la estructura podrá variar en ejecución (dinámica)

```
tipos cadena = ↑cad;
      cad = registro
              primero:carácter;
              resto:cadena
            freg
```

```
c,a:cadena;
```

{a, con cadena vacía}



Representación de la cadena: «la cadena».

Punteros y Datos Dinámicos en C++

- Las variables de tipo puntero se crean para poder apuntar a datos de un tipo concreto (fuertemente tipados)

  nombreTipo * nombreVariablePuntero

```
int a; //crea variable a de tipo int
int* ptA; //crea variable ptA, puntero a datos de tipo int.
/* Los espacios alrededor del * no son significativos,
 son equivalentes:   int* ptA;  int * ptA;  int *ptA;
*/
int* ptB, b;// crea variable b de tipo int, y
            // crea variable ptB puntero a datos de tipo int
int *c,*d; //crea variables c y d, ambas punteros a datos de tipo int

struct Persona {
  string nombre;
  int edad;
};
Persona p1,p2;// crea variables p1 y p2 ambas de tipo Persona
Persona* p; Persona* q; {o bien: Persona *p, *q;}
// crea variables p y q ambas punteros a datos de tipo Persona
```

- Declarar el puntero NO le asigna valor p ¿?
- A todo puntero, de cualquier tipo, se le puede asignar el valor especial nullptr (también se puede usar: 0, o el valor NULL como en C)

```
q=nullptr; p= nullptr;
```

q p

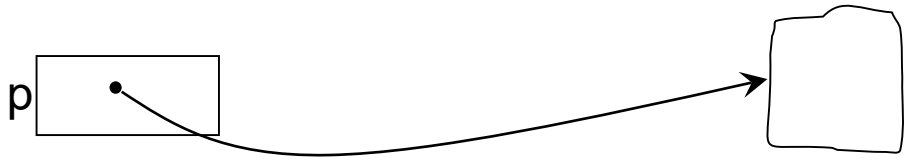
Punteros y Datos Dinámicos en C++

- Declarar el puntero NO reserva memoria dinámica para el dato apuntado
- Para **reservar memoria** para el dato apuntado:

```
punteroADato= new  TipoDato
```

  - El puntero utilizado queda asignado con la dirección de memoria que se reserva

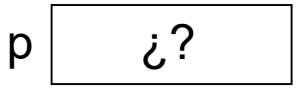
```
Persona* p;
...
p = new Persona;
//o bien:
// Persona* p= new Persona;
```

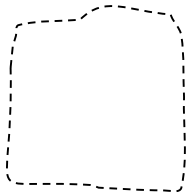


- Para **liberar la memoria** ocupada por el dato apuntado:

```
delete punteroADato
```

```
delete p;
```





  - si el puntero `p` utilizado (`delete p`) tiene valor nullptr, no ocurre nada

- Para gestionar (ocupar y liberar) memoria dinámica en C se utilizan funciones de librería (malloc, … free). Por compatibilidad con C están disponibles en C++, pero no son totalmente compatibles con new y delete → no mezclar ambas formas de gestionar la memoria dinámica.

Punteros y Datos Dinámicos en C++

- Sobre la asignación de punteros:
  1) La orden `new` devuelve la dirección de la memoria que se reserva, y normalmente se la asignamos a un puntero
  2) Se puede asignar a un puntero el valor de otro puntero, es decir, se le asigna la misma dirección que tiene otro puntero (quedan apuntando al mismo dato)

```
int* z;  int* t; ...  t=z;
```

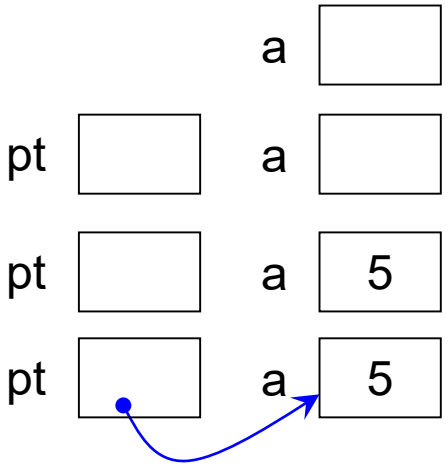
  3) Se puede asignar a un puntero la dirección de memoria en la que está un dato, utilizando el operador **&** , que se lee como "*la dirección de*" (*address-of operator*, no existe en seudocódigo ni existe en todos los lenguajes de programación)

```
int a;

int* pt;

a=5;

pt= &a;
```



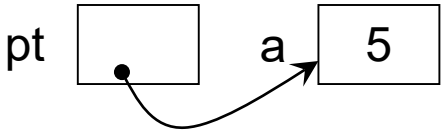
Punteros y Datos Dinámicos en C++

- Acceso al dato apuntado por el puntero (*dereference operator*):
  - Si p es el puntero, *"el dato apuntado por"* p es ***p**

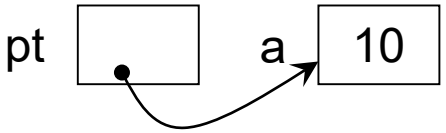
*No confundir con el * de la declaración de puntero*

  - Manipulación del dato apuntado por el puntero p, utilizando *p:

```
pt= &a;
```

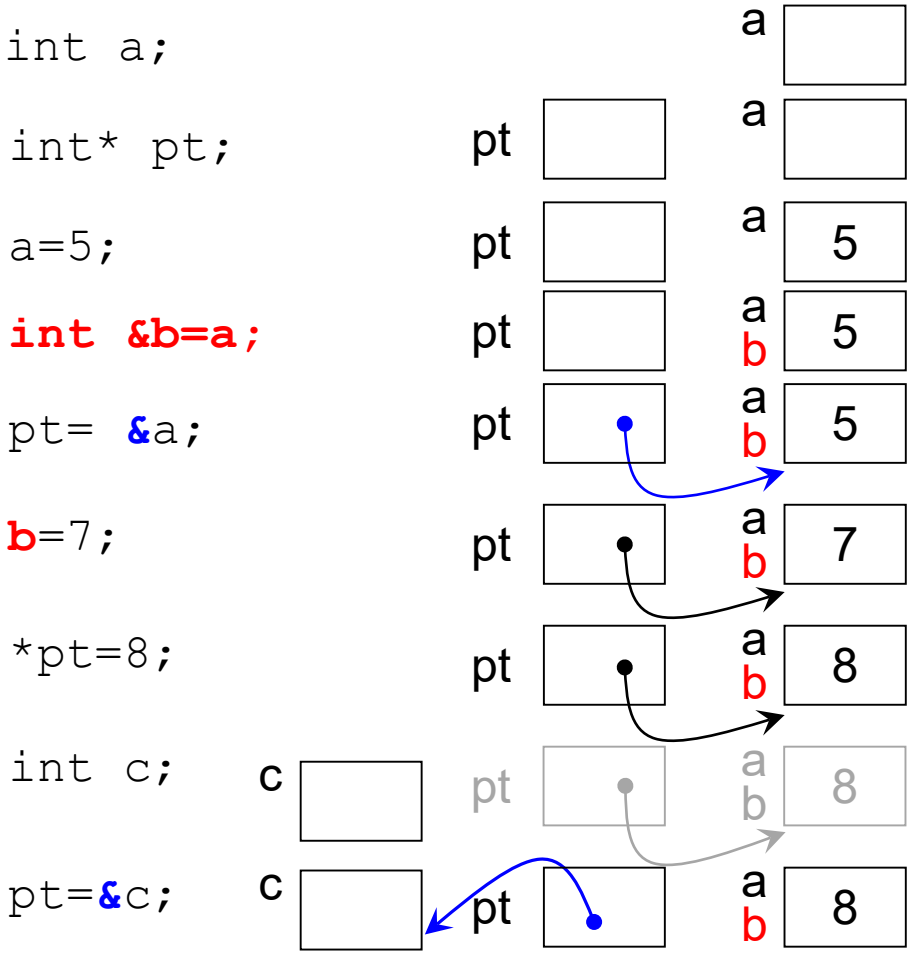



```
*pt= 10;
```



Punteros y Datos Dinámicos en C++

- Sobre la asignación de punteros:
  3) Se puede asignar a un puntero la dirección de memoria en la que está un dato, utilizando el operador **&** , que se lee como "*la dirección de*" (*address-of operator*, no existe en seudocódigo ni existe en todos los lenguajes de programación)



¡CUIDADO!

en C++ punteros y referencias NO son lo mismo

No confundir: `int *pt=&a;`

con: `int &b=a;`

{Crea un *alias* o referencia *b* para referenciar a la variable de tipo *int* llamada *a* *(que debe haberse declarado antes).* Es obligatorio asignarle una variable a *b* en su declaración.

Y *b* ya no podrá cambiar para *referenciar* a otra variable (cualquier asignación a *b* será una asignación a la zona de memoria también llamada *a*).}

¡ LOS PUNTEROS NO SON *ALIAS* !

Los punteros pueden reasignarse y utilizarse para apuntar a otro dato, siempre que sea del mismo tipo al que apunta el puntero

Para implementar estructuras de datos dinámicas usamos punteros, no alias.

Punteros y Datos Dinámicos en C++

- Acceso al dato apuntado por el puntero (*dereference operator*):

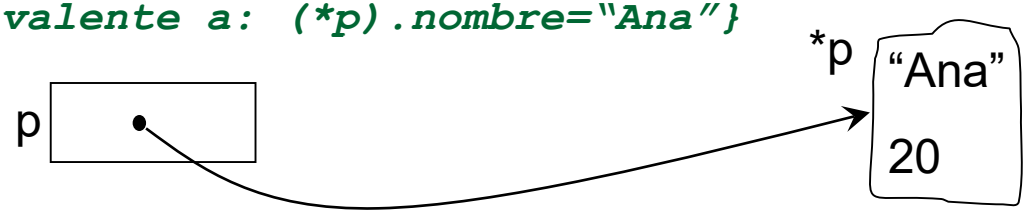
  – Cuando el tipo de dato apuntado tiene campos o miembros (struct, clases…) C++ ofrece el operador **->** para acceder directamente al campo desde el puntero (no existe en seudocódigo)

```
struct Persona {
        string nombre;
        int edad;
};

 Persona* p = new Persona;
```


```
 p->nombre="Ana";  {equivalente a: (*p).nombre="Ana"}

 p->edad=20;
```

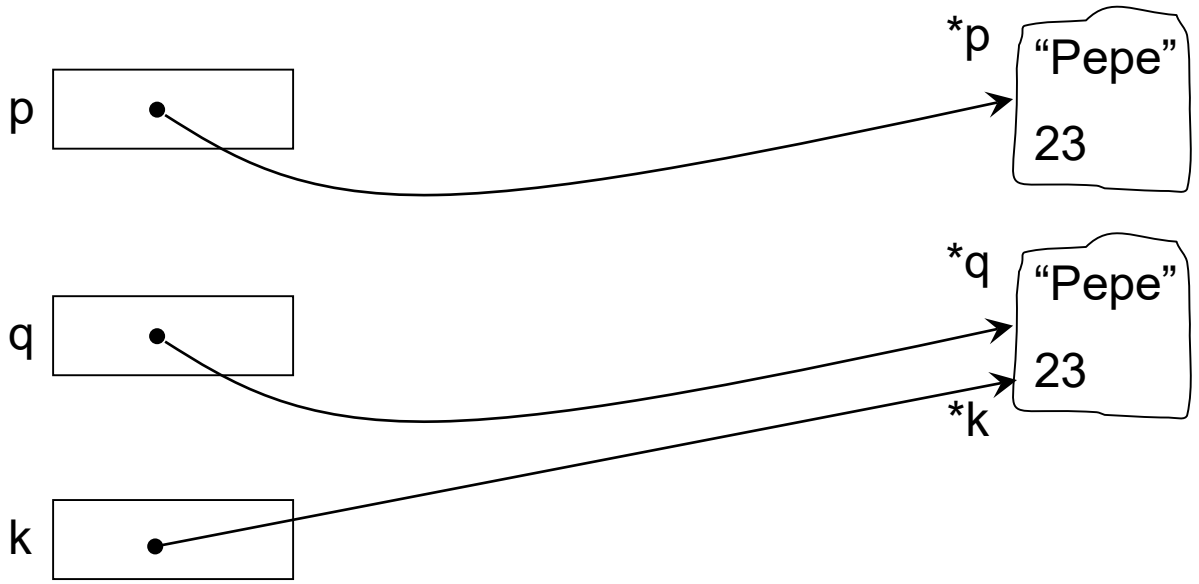



```
{Ojo: *p.nombre  equivaldría a *(p.nombre) }
```

Punteros y Datos Dinámicos en C++

- Comparación de punteros:
  - La comparación de punteros (==) comprueba si apuntan al mismo dato, no si los datos a los que apuntan tienen el mismo contenido

```
Persona* p, *q, *k;
p = new Persona;...
q= new Persona; ...
...
k=q;
```




```
¿ p == q ?  Es falso
¿ q == k ? Es verdad
```

  - Cualquier puntero puede compararse con el valor nullptr o NULL

```
¿ p==nullptr ? es falso
```

Estructuras de Datos Recursivas en C++

- En C++ una estructura de datos recursiva se define de forma similar a como se define en el seudocódigo:

```
struct Cadena {
    char elemento;
    Cadena* resto;
}
```

Punteros y Datos Dinámicos en C++

- Ejemplo

Seudocódigo:

```
Tipo Persona = registro
                 nombre : cadena;
                 edad   : entero;
               freg;
     Puntero_Persona = ↑Persona;
Variable
  p1, p3: Persona;
  p, q : Puntero_Persona;
  . . .
Principio . . .
  nuevoDato(p);
  p↑.nombre:="Pepe";
  p↑.edad := 23;

  disponer(p);

  p := nil;
  . . .
fin
```

C++:

```
struct Persona {
   string nombre;
   int edad;
};

Persona p1,p3;

Persona* p; Persona* q;
{o bien: Persona *p, *q;}

p = new Persona;
p->nombre="Pepe"; {equivalente a:
  (*p).nombre="Pepe"}
p->edad=23;

delete p;

p = nullptr;
 {por compatibilidad con C además de
  nullptr también se admite usar la
  constante NULL}
```

Paso de parámetros en C++

- Estudiado en programación I:
  - *"los parámetros cuyo valor es la referencia o dirección de memoria de un dato se denominan parámetros por referencia"*
- Cuándo utilizar parámetros por referencia y no parámetros por valor:
  - Para tener parámetros de *entrada/salida*, o parámetros de *salida*
  - Por eficiencia: para evitar el coste de copiar los datos (parámetros entrada)
    - Parámetros de entrada que no sean de tipos básicos o primitivos, los declararemos como *parámetros constantes por referencia*

Paso de parámetros en C++

- En C el paso de parámetros es por valor, y el programador tiene que implementar el paso por referencia usando punteros
- Ejemplo: intercambiar los valores de dos números, al estilo C

```
void intercambiarC (int* num1, int* num2){
     int numAux=*num1;
     *num1=*num2;
     *num2=numAux;
}
```

Este código es válido tanto en C como en C++

- Uso:

```
int c,d;     int *e,*f;
...
/* ... Reservar memoria para 2 datos int, y apuntar a ellos
con e y con f ... */
...
 c=10;  d=20;  *e=5;  *f=8;
//pasando las direcciones de los enteros a intercambiar:
intercambiarC(&c,&d);
//pasando dos punteros a los enteros a intercambiar:
intercambiarC(e,f);
/*liberar la memoria dinámica apuntada desde e y f...*/
```

Paso de parámetros en C++

- En C++ se ha incorporado el paso de parámetros por referencia
  - El programador indica que el parámetro se pasa por referencia (**&**)
  - Se simplifica el código
- Ejemplo: intercambiar los valores de dos números, en C++

```
void intercambiar (int& num1, int& num2){
    // num1 y num2 son de tipo int
    int numAux=num1;
    num1=num2;
    num2=numAux;
}
```

Este código es válido en C++

- Uso:

```
int a,b;
...
a=2;  b=3;
//se pasan los dos datos int a intercambiar:
intercambiar(a,b);
```

Paso de parámetros en C++

- Por eficiencia, los parámetros de entrada de tipo no básico o primitivo, serán *parámetros constantes por referencia*

Este código es válido en C++

- Ejemplo:

Se lee "p1 es una constante de tipo persona, que se pasa por referencia"

```
struct Persona{
    string nombre;
    int edad;
};

void Intentar (const Persona& p1, Persona& p3){
   //p1 y p3 son de tipo Persona
   p3.nombre="John Doe"; //OK
   //No se pueden hacer cambios a la Persona p1 por ser
   //constante:
   p1.nombre="Juan Nadie"; //Error de compilación
}
```

VISITAR: http://cslibrary.stanford.edu/106/ y http://cslibrary.stanford.edu/104/

MUY RECOMENDADO→ http://cslibrary.stanford.edu/102/PointersAndMemory.pdf

Pointer Fun with

Binky

by Nick Parlante

This is document 104 in the Stanford CS Education Library — please see cslibrary.stanford.edu for this video, its associated documents, and other free educational materials.

Carpe Post Meridiem!

Reglas básicas uso de punteros

1. Punteros versus datos apuntados

a) Primero se crea el puntero
   - ANTES deberemos haber declarado el tipo al que pertenece el puntero, para poder así crear punteros de ese tipo…..
   - Tipo del puntero no es lo mismo que el tipo del dato apuntado

b) Crear el puntero sólo reserva espacio de memoria para el puntero, NO para el dato al que apuntará

c) ¡NO OLVIDAR HACER QUE EL PUNTERO APUNTE A UN DATO! (DON'T FORGET TO SET UP THE POINTEE!)

ó

a) Puede crearse un nuevo dato reservándole memoria (new, NuevoDato…), y hacer que el puntero apunte al dato recién creado,

b) Asignaremos el puntero para que apunte al mismo dato apuntado por otro (asignación entre punteros)

Reglas básicas uso de punteros

2. Acceso al dato apuntado (en el video: dereference)

a) A partir del puntero se accede al dato apuntado (sintaxis dependiente del lenguaje)

b) Intentar acceder al dato apuntado por un puntero que tiene valor *null* o *nil* → error de ejecución

c) Manipular punteros no inicializados puede llevar a leer o escribir en zonas de memoria no deseadas, y a errores de ejecución muy difíciles de detectar

3. Asignación y comparación entre punteros

1. La asignación entre punteros hace que apunten al mismo sitio o dato, no afecta a los datos apuntados
2. La comparación entre punteros sólo comprueba si están apuntando al mismo sitio