

Punteros e introducción a estructuras dinámicas

Introducción

Hasta este momento hemos visto dos tipos de variables: globales y locales; las primeras pertenecen al algoritmo/programa principal y existen desde que éste se inicia hasta que finaliza. Las variables locales pertenecen a funciones y subrutinas y sólo existen durante la ejecución del subprograma; por esa razón, las variables locales se denominan automáticas (son creadas y destruidas por el procesador) y las globales estáticas (existen durante toda la ejecución del algoritmo sin poder crearse ni destruirse).

Hay ocasiones en las que puede resultar interesante crear y destruir variables de forma dinámica. Por ejemplo, imaginemos un programa para manejar fichas de alumnos, si quisiéramos implementar dicho programa con los conocimientos que tenemos hasta el momento deberíamos emplear un array de registros para almacenar las diferentes fichas; la cuestión que se plantea es la siguiente: ¿qué tamaño debería tener dicho array? Si lo declaramos demasiado pequeño podría darse el caso de que fuera imposible manejar todas las fichas existentes, si lo declaramos demasiado grande estaríamos desperdiciando memoria. La solución radicaría en poder definir algo que se ajustase en todo momento al tamaño necesario para lo cual haría falta crear y destruir registros (fichas de alumnos) en tiempo de ejecución.

Las variables cuya creación y destrucción no coincide con el inicio y el fin de un algoritmo o subprograma y que, por tanto, es controlada por el diseñador del algoritmo se denominan variables dinámicas; la mayor parte de lenguajes de programación modernos, incluido FORTRAN 90, pueden manipular de una u otra forma este tipo de variables.

En este capítulo presentaremos los conceptos básicos necesarios para entender las variables dinámicas así como la forma en que FORTRAN implementa dichas variables.

Punteros

La herramienta básica para la creación de variables dinámicas son los punteros o apuntadores. Un puntero es un tipo simple (como los enteros, reales o caracteres) pero que siempre está asociado a otro tipo; por ejemplo, podemos tener punteros a enteros, punteros a reales, punteros a registros, etc.

Una forma sencilla de entender los punteros es imaginarlos como variables que almacenan direcciones de memoria; de aquí se deducen una serie de hechos:

1. El puntero es un medio para acceder al contenido “real” apuntado por la variable.
2. La declaración de una variable de tipo puntero no implica que exista un contenido apuntado por la misma.
3. Debido a esto, es necesario crear y destruir explícitamente el contenido apuntado por las variables de tipo puntero.

Se puede establecer una analogía entre los punteros y las direcciones postales:

1. Una dirección postal permite acceder al lugar físico indicado mediante la propia dirección.
2. El hecho de disponer de una dirección postal no garantiza que el lugar realmente exista; por ejemplo, “742 Evergreen Terrace, Springfield (USA)” es una dirección postal pero no existe ningún lugar real que responda a la misma.

Para declarar variables de tipo puntero en la notación algorítmica se utiliza la sintaxis siguiente:

variable \in puntero a tipo

Por ejemplo,

```
preal  $\in$  puntero a real
pentero  $\in$  puntero a entero
palumno  $\in$  puntero a alumno
```

Una vez un puntero ha sido declarado pueden llevarse a cabo dos acciones básicas sobre el mismo: asignarle memoria para almacenar un dato del tipo al que apunta y eliminar la memoria que tiene asignada (y con ella el dato apuntado); para ello se emplean las acciones `crear` y `destruir` con la siguiente sintaxis:

```
crear (variable puntero)
destruir (variable puntero)
```

Por ejemplo, en las líneas siguientes se va a reservar memoria para los punteros anteriormente declarados:

```
crear (preal)
crear (pentero)
crear (palumno)
```

A una variable de tipo puntero es posible asignarle el contenido de otra variable de tipo puntero; para hacer esto es necesario que la variable a la que vamos a asignar un valor no apunte a ningún dato pues, en ese caso, el dato sería inaccesible después de la asignación. Esto es fácil de entender si continuamos con el ejemplo de las direcciones postales; imaginemos que tenemos tarjetas de cartón para apuntar direcciones, siempre podremos copiar la dirección de una tarjeta en otra, sin embargo, si la tarjeta sobre la que escribimos ya tenía una dirección la perderemos y ya no podremos acceder al lugar que indicaba. Así pues, sólo deberíamos asignar un puntero a otro si el puntero asignado no tiene memoria reservada.

Una vez se ha reservado memoria para una variable de tipo puntero es posible almacenar un dato del tipo apuntado en dicho espacio de memoria; la sintaxis que se emplea para llevar a cabo esta acción es la siguiente:

variable de tipo puntero \uparrow \leftarrow expresión del tipo apuntado

Por ejemplo, las siguientes sentencias asignan valores a las variables referenciadas por los punteros anteriormente declarados:

```
preal $\uparrow$   $\leftarrow$  3.1416
pentero $\uparrow$   $\leftarrow$  5
```

A continuación se presenta un ejemplo sencillo que ilustrará lo visto hasta el momento:

```
variables
  x, y  $\in$  a entero
  pa, pb  $\in$  puntero a entero

inicio
  x  $\leftarrow$  1
  y  $\leftarrow$  2

  crear(pa)

  pa $\uparrow$   $\leftarrow$  10
  pb  $\leftarrow$  pa

  escribir x + y
  escribir pa $\uparrow$  + pb $\uparrow$ 

  destruir (pa)
fin
```

Como se puede ver en el algoritmo anterior se han declarado dos variables de tipo entero (x e y) y dos variables de tipo puntero a entero (pa y pb); a las dos primeras se les asignan los valores 1 y 2 mientras que para el primero de los punteros se reserva memoria y se asigna a la variable referenciada el valor 10 mientras que la segunda, pb , se hace apuntar a pa (accediendo, por tanto, al valor de dicha variable, 10). Posteriormente, se emplean las variables enteras y las variables referenciadas por punteros a entero en sendas expresiones para, finalmente, liberar la memoria ocupada por una de las variables de tipo puntero. En dicho algoritmo se puede apreciar un aspecto muy importante, por cada sentencia de reserva de memoria para un puntero debe existir una sentencia de liberación de memoria.

En la figura siguiente se muestra la evolución de las variables empleadas por el algoritmo así como la creación y destrucción de variables dinámicas referenciadas por los punteros.

Variables del algoritmo		Inicio
pa		x \leftarrow 1
pb		y \leftarrow 2
x		
y		

Variables del algoritmo		inicio
pa		x \leftarrow 1
pb		y \leftarrow 2
x	1	
y		

Variables del algoritmo

pa	
pb	
x	1
y	2

```

inicio
  x ← 1
  y ← 2

```

Variables del algoritmo

pa		→	
pb			
x	1		
y	2		

```

crear(pa)
pa↑ ← 10
pb ← pa

```

Variables del algoritmo

pa		→	10
pb			
x	1		
y	2		

```

crear(pa)
pa↑ ← 10
pb ← pa

```

Variables del algoritmo

pa		→	10
pb		→	↑
x	1		
y	2		

```

crear(pa)
pa↑ ← 10
pb ← pa

```

Variables del algoritmo

pa			
pb		→	⚡
x	1		
y	2		

```

destruir (pa)

```

Obsérvese cómo al liberar la memoria asignada a `pa` la variable `pb` continua apuntando a la posición de memoria anteriormente asignada a dicha variable; tras la destrucción de la variable referenciada dicha posición de memoria no contiene ningún dato válido por lo que la variable `pb` apunta a lo que puede considerarse como un dato “basura”.

De aquí se deduce que cuando a un puntero se asigna el contenido de otro puntero hay que cuidar que el puntero no termine referenciando posiciones sin datos válidos; para lograr esto existe una constante que siempre se puede asignar a una variable de tipo puntero indicando que el puntero no referencia ninguna posición de memoria, nos referimos a `NIL`. De esta forma, en el algoritmo anterior una vez se ha destruido el puntero habría que ejecutar:

```
pb ← NIL
```

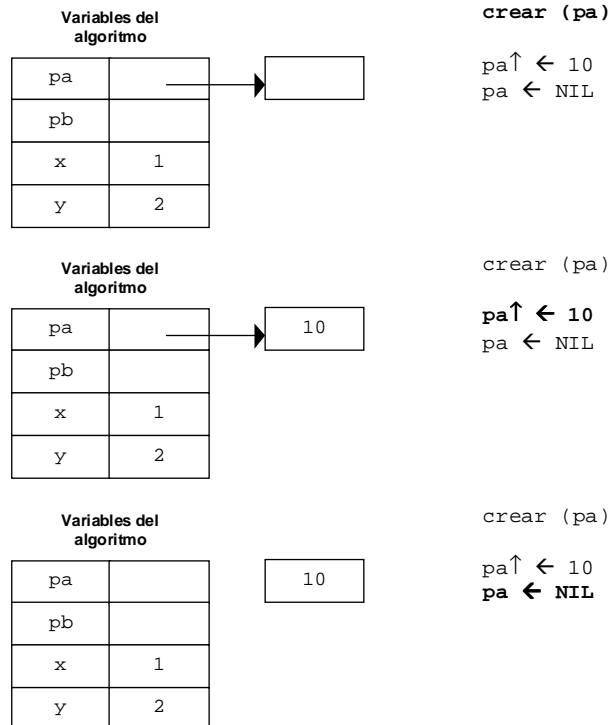
En referencia a `NIL` es necesario tener en cuenta que el hecho de asignar este valor a un puntero que tiene asignada una posición de memoria no libera este espacio sino que simple y llanamente lo hace inaccesible desperdiçándolo. Supongamos que en el algoritmo anterior se sustituye la sentencia

```
destruir (pa)
```

por esta otra

```
pa ← NIL
```

El resultado se muestra en la siguiente secuencia:



Como se puede ver, el resultado puede parecer equivalente (de hecho el programador no percibiría ningún “problema”), sin embargo, se está desperdiciando memoria puesto que una vez que hemos perdido todos los apuntadores a una posición de memoria es imposible liberarla. De esto se deduce otra práctica que es necesario tener en cuenta a la hora de trabajar con punteros en nuestros algoritmos: **nunca se pueden perder todos los punteros a una posición de memoria.**

Si nos atenemos a las siguientes normas nunca deberíamos tener problemas con los punteros:

1. Por cada sentencia de reserva de memoria tiene que haber una sentencia de liberación.
2. Para asignar un puntero a otro, el puntero asignado no debe tener asignada memoria previamente.
3. Para asignar un valor a la memoria referenciada por un puntero es necesario que se haya reservado memoria previamente.
4. Cuando la memoria apuntada por un puntero ha desaparecido se debe asignar el valor `NIL` al puntero.
5. Nunca se pueden perder todas las referencias a memoria reservada pues entonces ya no se podría liberar.

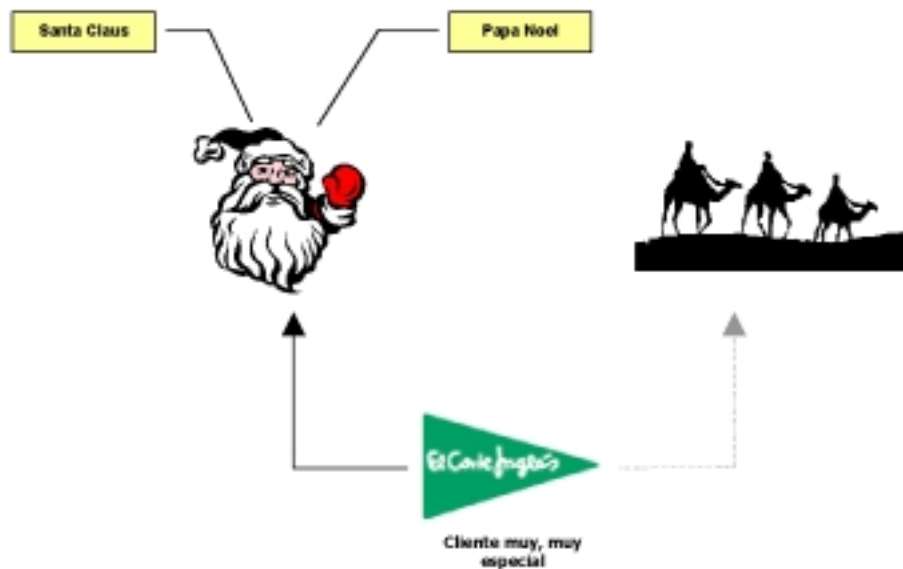
Referencias

Las referencias tienen una cierta similitud con los punteros, para entenderlas mejor nos referiremos a ellas como “alias” puesto que se trata de variables que aunque apuntan a una variable (he aquí el parecido a los punteros) pueden ser utilizadas como si se tratase directamente con la variable apuntada (en esto no se parecen en nada a los punteros).

Las referencias poseen una serie de propiedades que las hacen diferentes de los punteros:

- Los punteros pueden declararse y no reservarse memoria para los mismos (es decir, no apuntar a ninguna parte); sin embargo, las referencias deben inicializarse en el momento de crearlas.
- La memoria asignada a un puntero puede destruirse en cualquier momento; sin embargo, la variable referenciada por una referencia no puede ser destruida.
- Es posible cambiar en cualquier momento la variable apuntada por un puntero; sin embargo, la variable “apuntada” por una referencia (indicada en el momento de crear la variable) no puede modificarse.

En la figura siguiente se proporciona un ejemplo sencillo que servirá para ilustrar las diferencias entre referencias y punteros.



En la figura anterior aparecen dos “objetos”: un alegre lapón que reparte juguetes en Nochebuena y un grupo de magos de Oriente que reparten juguetes en fechas más tardías; ambos “objetos” son referenciados por unos conocidos grandes almacenes que los consideran grandes clientes.

Como se aprecia en la ilustración los grandes almacenes pueden considerar (probablemente en función de la fecha) a uno u otro “objeto” como cliente especial; en ese sentido “cliente muy, muy especial” sería un puntero a comprador.

Por otro lado, obsérvese que el alegre lapón es apuntado, además de por los grandes almacenes, por dos etiquetas: “Santa Claus” y “Papa Noel”; dichas etiquetas son referencias (alias) puesto que no pueden ser creadas sin asignarles un “objeto” (“Papa Noel” no existe por sí mismo, sino como alias de “Santa Claus”) y todo lo que se haga a una de las referencias se le hace a la otra.

Esto último quedará más claro mediante el siguiente ejemplo: supongamos que los grandes almacenes consideran a “Santa Claus” como un “cliente muy, muy especial”, en ese caso está claro que “Papa Noel” también será un gran cliente. Sin embargo, un espíritu malicioso podría decidir secuestrar a “Papá Noel” con lo cual está claro que:

1. “Santa Claus” es secuestrado puesto que se trata del mismo “objeto”.
2. Los grandes almacenes se quedan sin su gran cliente pero, como se trata de un puntero, pueden decidir referenciar otro objeto, por ejemplo, a los magos de Oriente.

No todos los lenguajes de programación utilizan referencias, de hecho nuestra notación algorítmica no las empleará; la razón de esta breve introducción a las referencias es muy simple: FORTRAN 90 no dispone de un tipo puntero exactamente igual al descrito antes sino que, en ciertos aspectos, se comporta como las referencias. En la sección siguiente se describirá la sintaxis para la declaración de variables de tipo “puntero” en FORTRAN y se pondrán ejemplos ilustrativos sobre su manejo.

“Punteros” en FORTRAN 90

Mediante la palabra reservada `pointer` es posible utilizar los “punteros” o “referencias” de FORTRAN 90; la denominación de los mismos de una u otra forma no nos interesa demasiado (aunque propiamente deberían considerárselos referencias en lugar de punteros), lo único que debemos tener en mente es que su utilización nos permitirá crear y destruir de manera dinámica variables y, con ellas, estructuras dinámicas de datos (más adelante veremos qué son y en qué consisten las estructuras dinámicas de datos).

Para declarar variables con el atributo `pointer` en FORTRAN 90 se utiliza la sintaxis siguiente:

```
tipo, pointer :: variable
```

Por ejemplo,

```
real, pointer :: preal
integer, pointer :: pentero
type(alumno), pointer :: palumno
```

Una vez un puntero ha sido declarado pueden llevarse a cabo las dos acciones básicas de asignación y liberación de memoria; para ello se emplean las funciones `allocate` y `deallocate` con la siguiente sintaxis:

```
allocate (variable pointer)
deallocate (variable pointer)
```

Por ejemplo, en las líneas siguientes se va a reservar memoria para los `pointer` anteriormente declarados:

```
allocate (preal)
allocate (pentero)
allocate (palumno)
```

A diferencia de en la notación algorítmica **a una variable con el atributo `pointer` NO es posible asignarle el contenido de otra variable de tipo puntero mediante el operador asignación**; para hacer que una variable de este tipo “apunte” al mismo objetivo que otro puntero se emplea el operador `=>`. Por ejemplo, lo que en la notación algorítmica se hacía como:

```
preal_1 ← preal_2
```

En FORTRAN 90 se representa así:

```
preal_1 => preal_2 ! Así, sí
```

Y nunca de esta forma:

```
preal_1 = preal_2 ! Así, NO
```

¿Por qué razón no puede utilizarse de esta forma el operador de asignación con los punteros de FORTRAN? Porque dicho operador se emplea para asignar un valor a la memoria apuntada por el puntero; es decir, lo que en la notación algorítmica se expresaba de esta forma:

```
variable puntero↑ ← expresión del tipo apuntado
```

En FORTRAN 90, se traduce a:

```
variable pointer = expresión del tipo apuntado
```

Por ejemplo,

```
preal↑ ← 3.141592
pentero↑ ← 5
```

Se traduciría como:

```
preal = 3.141592
pentero = 5
```

Al igual que en la notación algorítmica, **para asignar un valor a la memoria asignada a un `pointer`, lógicamente es necesario haber reservado memoria previamente para dicho `pointer`**.

Así pues, en FORTRAN la asignación a punteros se lleva a cabo mediante el operador `=>` y la asignación a memoria referenciada por punteros mediante el operador `=`.

A continuación mostraremos el algoritmo de la sección anterior traducido a FORTRAN 90:

```
program programa
implicit none
integer, pointer :: pa,pb
integer x,y

x=1
y=2

allocate(pa)

pa = 10
pb => pa

print *, x + y
print *, pa + pb

deallocate(pa)
end
```

Como se puede apreciar en este código, los `pointer FORTRAN` se comportan como punteros por lo que respecta a la asignación (`allocate`) y liberación (`deallocate`) de memoria; en todos los demás aspectos se comportan como el tipo al que apuntan, esto es, como referencias. Para comprenderlo mejor, en la secuencia siguiente se muestra cómo evoluciona el estado de las variables en el programa:

Variables del algoritmo	
pa	
pb	
x	
y	

```
program programa
implicit none
integer, pointer :: pa,pb
integer x,y

x=1
y=2
```

Variables del algoritmo	
pa	
pb	
x	1
y	


```
program programa
implicit none
integer, pointer :: pa,pb
integer x,y

x=1
y=2
```

Variables del algoritmo	
pa	
pb	
x	1
y	2


```
program programa
implicit none
integer, pointer :: pa,pb
integer x,y

x=1
y=2
```

Variables del algoritmo	
pa	
pb	
x	1
y	2


```
allocate(pa)
```

```
pa = 10
pb => pa
```

Variables del algoritmo	
pa	
pb	
x	1
y	2

```
allocate(pa)
```

```
pa = 10
pb => pa
```

Variables del algoritmo	
pa	
pb	
x	1
y	2

```
allocate(pa)
```

```
pa = 10
pb => pa
```

Variables del algoritmo	
pa	
pb	
x	1
y	2

```
print *, x + y
print *, pa + pb
```

```
deallocate(pa)
end
```

A la vista de la figura anterior puede parecer que dado que `pb` está asociado a `pa` (es un alias de dicho puntero), al liberar la memoria de `pa` también se liberaría (puesto que es la misma) la memoria de `pb`. Sin embargo, dependiendo del compilador pueden darse diversas opciones no definidas por el lenguaje; por tanto, de manera análoga a como se hacía en nuestra notación algorítmica es necesario hacer que un “puntero” que apuntaba a una zona de memoria recién liberada apunte al equivalente FORTRAN de `NIL`; para ello existe la función `nullify`:

```
nullify (variable pointer)
```

En el código anterior, una vez liberada la memoria apuntada por `pa`, se ejecutaría la instrucción

```
nullify (pb)
```

También es cierto aquí que el hecho de hacer que un pointer apunte al equivalente de `NIL` no significa que la memoria haya sido liberada, sólo que es inaccesible (lo cual no es bueno). Para terminar con los punteros nulos tan sólo resta decir que al no existir una constante como `NIL` para saber si un pointer apunta a algo o no se dispone de la función booleana:

```
associated (variable pointer)
```

De manera análoga a los punteros de la notación algorítmica también tenemos unas “reglas” que nos permitirán trabajar con los pointer FORTRAN de manera relativamente segura:

Si nos atenemos a las siguientes normas nunca deberíamos tener problemas con los punteros:

1. Por cada sentencia `allocate` tiene que haber una sentencia `deallocate`.
2. Para asociar un pointer a otro (mediante el operador `=>`) el pointer asignado no debe tener asignada memoria previamente.
3. Para asignar un valor a la memoria referenciada por un pointer (mediante el operador `=`) es necesario que se haya reservado memoria previamente.
4. Cuando la memoria apuntada por un pointer ha desaparecido se debe aplicar la función `nullify` a dicho puntero.
5. Nunca se pueden perder todas las referencias a memoria reservada pues entonces ya no se podría liberar.

Ejemplo de la utilización de punteros en la notación algorítmica y pointer en FORTRAN

En este último apartado se presentará una estructura de datos muy sencilla: una cola simplemente enlazada. Una cola está constituida por una serie de elementos (0 ó más) todos del mismo tipo que almacenan un dato (que puede ser tan complejo como queramos) y un apuntador al siguiente elemento en la cola. Las operaciones que existen en la cola son fundamentalmente dos: insertar un elemento en la cola y eliminar un elemento de la cola.

Para facilitar la comprensión del ejemplo supondremos que es una cola en una caja de supermercado, los elementos almacenarán el nombre del cliente y las operaciones se denominarán “ponerse a la cola” y “atender cliente”. El ejemplo se proporcionará tanto en su versión pseudocódigo como una versión FORTRAN 90, a la luz de este ejemplo se podrá apreciar cómo los pointer de FORTRAN aunque aúnan características de punteros y referencias son de muy fácil comprensión.

Una vez expuestos los ejemplos se explica de forma gráfica el funcionamiento de los subprogramas de inserción y borrado de elementos dentro de la cola.

Cola de supermercado en notación algorítmica

```
tipos
  cliente = tupla
    nombre ∈ carácter
    siguientecliente ∈ puntero a cliente
  fin tupla

variables
  colaSuper ∈ puntero a cliente
  nombre ∈ carácter

inicio
  colaSuper ← NIL

  mientras nombre ≠ '.' hacer
    escribir 'Deme su nombre (pulse . para finalizar):'
    leer nombre
    si nombre ≠ '.' entonces
      ponerseCola (nombre, colaSuper)
    fin si
  fin mientras
```



```

mientras nombre ≠ '' hacer
  nombre ← atenderCliente (colaSuper)
  escribir nombre
fin mientras

accion ponerseCola (nombreCliente ∈ caracter, cola ∈ puntero a cliente)
inicio
  si cola = NIL entonces
    crear (cola)
    cola↑.nombre ← nombreCliente
    cola↑.siguienteCliente ← NIL
  si no
    llamar ponerseCola (nombreCliente, cola↑.siguienteCliente)
  fin si
fin accion

carácter funcion atenderCliente (cola ∈ puntero a cliente)
variables
  cursor ∈ puntero a cliente
inicio
  si cola = NIL entonces
    atenderCliente ← ''
  si no
    cursor = cola↑.siguienteCliente
    atenderCliente ← cola↑.nombre
    destruir (cola)
    cola ← cursor
  fin si
fin funcion
fin

```

Cola de supermercado en FORTRAN 90

```

program ejemploCola
  implicit none

  ! Se define el tipo base de la cola
  !
  type cliente
    character*64 nombre
    type(cliente), pointer :: siguienteCliente
  end type cliente

  ! La cola es un puntero al tipo base
  !
  type(cliente), pointer :: colaSuper
  character*64 nombre

  ! Se inicializa la cola
  !
  nullify (colaSuper)

  do while (nombre/='.')
    print *, 'Deme su nombre (pulse . para finalizar):'
    read *, nombre
    if (nombre/='.') then
      call ponerseCola (nombre, colaSuper)
    end if
  end do

  do while (nombre/='')
    nombre = atenderCliente (colaSuper)
    print *, nombre
  end do

  ! Subprogramas utilizados por el programa principal
  !
  contains

  ! Subrutina recursiva para añadir al final de la cola
  !
  recursive subroutine ponerseCola (nombreCliente, cola)
    implicit none

    character*64, nombreCliente
    type(cliente), pointer :: cola

```

```

    if (.not.associated cola)) then
        allocate (cola)
        cola%nombre=nombreCliente
        nullify (cola%siguienteCliente)
    else
        call ponerseCola (nombreCliente, cola%siguienteCliente)
    end if
end subroutine ponerseCola

! Función que retorna el nombre del primer cliente y lo elimina de la cola
!
character*64 function atenderCliente (cola)
    implicit none

    type(cliente), pointer :: cola, cursor

    if (.not.associated (cola)) then
        atenderCliente = ''
    else
        cursor => cola%siguienteCliente
        atenderCliente = cola%nombre
        deallocate (cola)
        cola => cursor
    end if
end function atenderCliente
end

```

Funcionamiento de los subprogramas **ponerseCola** y **atenderCliente**

Subprograma ponerseCola

Este subprograma sirve para insertar un nuevo elemento en una cola; la inserción en una cola siempre es al final de la misma por lo que hay que recorrerla completamente hasta el final y, una vez allí, crear un nuevo elemento al que se obligue a enlazar con la cola antigua. Este problema, como todos los problemas iterativos, puede resolverse de forma muy elegante empleando recursividad:

1. Suponemos que ya existe un algoritmo que permite insertar elementos al final de una cola.
2. Existe un caso muy simple que no precisa dicho algoritmo: una cola vacía; para añadir un elemento a una cola vacía simplemente hay que crear el elemento y hacer que la cola apunte a dicho elemento.
3. El caso general entonces se producirá cuando la cola no esté vacía, en ese caso podemos considerar una cola como un elemento seguido por una cola más corta; así pues, aplicamos el algoritmo recursivamente sobre esta cola más corta (que terminará estando vacía).

De ahí el ya conocido algoritmo:

```

accion ponerseCola(nombreCliente € caracter, cola € puntero a cliente)
inicio
    si cola = NIL entonces
        crear (cola)
        cola↑.nombre ← nombreCliente
        cola↑.siguienteCliente ← NIL
    si no
        llamar ponerseCola (nombreCliente, cola↑.siguienteCliente)
    fin si
fin accion

```

```

recursive subroutine ponerseCola (nombreCliente, cola)
    implicit none

    character*64, nombreCliente
    type(cliente), pointer :: cola

    if (.not.associated (cola)) then
        allocate (cola)
        cola%nombre=nombreCliente
        nullify (cola%siguienteCliente)
    else
        call ponerseCola (nombreCliente, cola%siguienteCliente)
    end if
end subroutine ponerseCola

```

A continuación se muestra gráficamente el proceso de inserción del elemento 'Pepe' en una cola vacía (se presenta sólo el pseudocódigo, el código FORTRAN equivalente se muestra en la figura anterior):

cola → NIL

cola →

--	--

cola →

'Pepe'	
--------	--

cola →

'Pepe'	
--------	--

 → NIL

```
si cola = NIL entonces
  crear (cola)
  cola↑.nombre ← nombreCliente
  cola↑.siguienteCliente ← NIL
```

```
si cola = NIL entonces
  crear (cola)
  cola↑.nombre ← nombreCliente
  cola↑.siguienteCliente ← NIL
```

```
si cola = NIL entonces
  crear (cola)
  cola↑.nombre ← nombreCliente
  cola↑.siguienteCliente ← NIL
```

```
si cola = NIL entonces
  crear (cola)
  cola↑.nombre ← nombreCliente
  cola↑.siguienteCliente ← NIL
```

La siguiente secuencia muestra el proceso de inserción del elemento 'Chus' en una cola con 2 elementos:

cola →

'Pepe'	
--------	--

--	--

 → NIL

cola' →

'María'	
---------	--

 → NIL

cola'' → NIL

cola'' →

--	--

cola'' →

Chus	
------	--

cola'' →

'Chus'	
--------	--

 → NIL

cola' →

'María'	
---------	--

--	--

 → NIL

cola →

'Pepe'	
--------	--

--	--

 →

'María'	
---------	--

--	--

 →

'Chus'	
--------	--

 → NIL

```
si cola = NIL entonces
  ...
  si no
    llamar ponerseCola (nombreCliente,cola↑.siguienteCliente)
  fin si
```

```
si cola = NIL entonces
  ...
  si no
    llamar ponerseCola (nombreCliente,cola↑.siguienteCliente)
  fin si
```

```
si cola = NIL entonces
  crear (cola)
  cola↑.nombre ← nombreCliente
  cola↑.siguienteCliente ← NIL
```

```
si cola = NIL entonces
  crear (cola)
  cola↑.nombre ← nombreCliente
  cola↑.siguienteCliente ← NIL
```

```
si cola = NIL entonces
  crear (cola)
  cola↑.nombre ← nombreCliente
  cola↑.siguienteCliente ← NIL
```

```
si cola = NIL entonces
  crear (cola)
  cola↑.nombre ← nombreCliente
  cola↑.siguienteCliente ← NIL
```

```
si cola = NIL entonces
  ...
  si no
    llamar ponerseCola (nombreCliente,cola↑.siguienteCliente)
  fin si
```

```
si cola = NIL entonces
  ...
  si no
    llamar ponerseCola (nombreCliente,cola↑.siguienteCliente)
  fin si
```

Subprograma atenderCliente

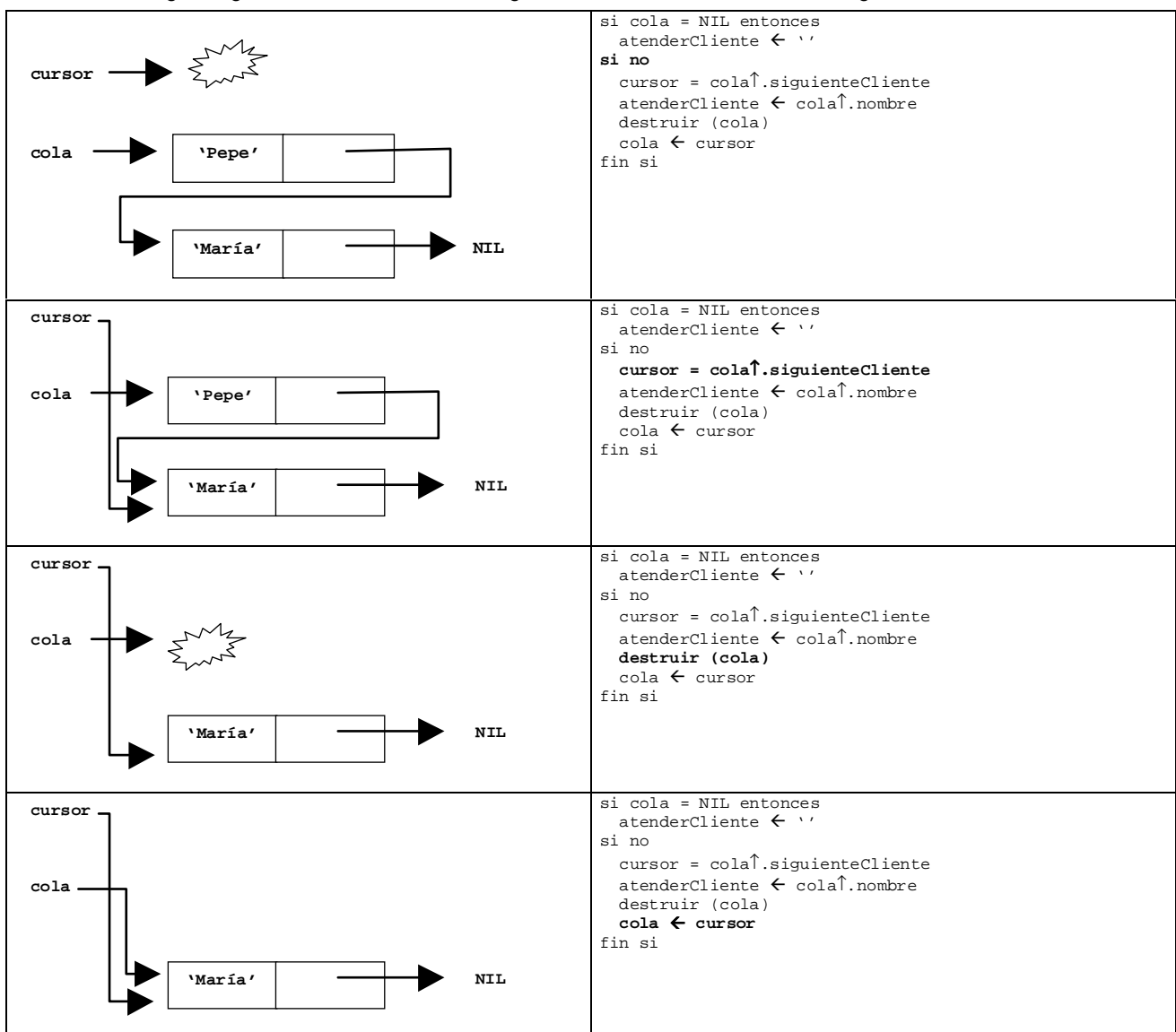
Esta función realiza dos acciones: por un lado, toma el nombre del primer elemento de la cola y lo retorna como valor de la función; por otro, elimina este elemento de la cabeza de la cola. Dado que trabaja siempre con la cabeza de la cola sólo hay dos casos posibles: la cola está vacía y hay un elemento en la cabeza de la cola; el primer caso es trivial por lo que sólo se mostrará el segundo.

```
carácter funcion atenderCliente (cola ∈ puntero a cliente)
variables
  cursor ∈ puntero a cliente
inicio
  si cola = NIL entonces
    atenderCliente ← ''
  si no
    cursor = cola↑.siguienteCliente
    atenderCliente ← cola↑.nombre
    destruir (cola)
    cola ← cursor
  fin si
fin funcion
```

```
character*64 function atenderCliente (cola)
  implicit none
  type(cliente), pointer :: cola, cursor

  if (.not.associated(cola)) then
    atenderCliente = ''
  else
    cursor => cola%siguienteCliente
    atenderCliente = cola%nombre
    deallocate(cola)
    cola => cursor
  end if
end function atenderCliente
```

En la figura siguiente se muestra de forma gráfica el funcionamiento de dicho algoritmo con una cola no vacía.



Resumen

1. Frente a las variables globales (estáticas) y locales (automáticas) existe un tercer tipo denominadas variables dinámicas puesto que son creadas y destruidas en tiempo de ejecución.
2. Las variables dinámicas son la base para construir estructuras dinámicas de datos (como colas, pilas, árboles, etc.)
3. Los punteros permiten crear y destruir variables dinámicas; un puntero es una variable que almacena una dirección de memoria (por tanto, “apunta” a la dirección ocupada por otra variable).
4. Los punteros presentan una serie de características:
 - Un puntero es un medio para acceder al contenido “real” apuntado por la variable.
 - La declaración de una variable de tipo puntero no implica que exista un contenido apuntado por la misma.
 - Es necesario crear y destruir explícitamente el contenido apuntado por las variables de tipo puntero.
5. Para declarar variables de tipo puntero en la notación algorítmica se utiliza la siguiente sintaxis:
`variable ∈ puntero a tipo`
6. Una vez un puntero ha sido declarado puede asignársele memoria y eliminar la memoria asignada; para ello se emplean las acciones `crear` y `destruir`.
7. Para trabajar con punteros es necesario tener en cuenta las siguientes reglas:
 - Por cada sentencia de reserva de memoria tiene que haber una sentencia de liberación.
 - Para asignar un puntero a otro, el puntero asignado no debe tener asignada memoria previamente.
 - Para asignar un valor a la memoria referenciada por un puntero es necesario que se haya reservado memoria previamente.
 - Cuando la memoria apuntada por un puntero ha desaparecido se debe asignar el valor `NIL` al puntero.
 - Nunca se pueden perder todas las referencias a memoria reservada pues entonces ya no se puede liberar.
8. Las referencias se parecen a los punteros aunque, en realidad, son alias ya que, aunque apuntan a una variable, se utilizan como si se tratase directamente con la variable apuntada.
9. Hay varias propiedades que diferencian a una referencia de un puntero:
 - Los punteros pueden declararse y no reservarse memoria para los mismos; las referencias deben inicializarse en el momento de su creación.
 - La memoria asignada a un puntero puede destruirse en cualquier momento; sin embargo, la variable referenciada por una referencia no puede ser destruida.
 - Siempre se puede cambiar la variable apuntada por un puntero; en cambio, no se puede cambiar el “objetivo” de una referencia.
10. FORTRAN 90 dispone de la palabra reservada `pointer`; esta palabra permite modificar el comportamiento de una variable de tal manera que se comporte como un “puntero” en algunos aspectos y como una “referencia” en otros.
11. Las acciones para reservar y liberar memoria en FORTRAN 90 son `allocate` y `deallocate`, respectivamente.
12. El operador para cambiar la referencia a la que apunta un `pointer` es `=>` ya que el operador `=` sirve para asignar un valor a la variable referenciada.
13. En FORTRAN 90 no existe una constante `NIL` pero sí una función `nullify` que consigue un efecto análogo. Para saber si un `pointer` es nulo o apunta a alguna referencia se utiliza la función booleana `associated`.