

## Capítulo 12

# Memoria Dinámica. Punteros

Hasta ahora, todos los programas que se han visto en capítulos anteriores almacenan su estado interno por medio de variables que son automáticamente gestionadas por el compilador. Las variables son *creadas* cuando el flujo de ejecución entra en el ámbito de su definición (se reserva espacio en memoria y se crea el valor de su estado inicial), posteriormente se *manipula* el estado de la variable (accediendo o modificando su valor almacenado), y finalmente se *destruye* la variable cuando el flujo de ejecución sale del ámbito donde fue declarada la variable (liberando los recursos asociados a ella y la zona de memoria utilizada). A este tipo de variables gestionadas automáticamente por el compilador se las suele denominar *variables automáticas* (también variables locales), y residen en una zona de memoria gestionada automáticamente por el compilador, la *pila* de ejecución, donde se alojan y desalojan las variables locales (automáticas) pertenecientes al ámbito de ejecución de cada subprograma.

Así, el tiempo de vida de una determinada variable está condicionado por el ámbito de su declaración. Además, el número de variables automáticas utilizadas en un determinado programa está especificado explícitamente en el propio programa, y por lo tanto su capacidad de almacenamiento está también especificada y predeterminada por lo especificado explícitamente en el programa. Es decir, con la utilización única de variables automáticas, la capacidad de almacenamiento de un determinado programa está predeterminada desde el momento de su programación (tiempo de compilación), y no puede adaptarse a las necesidades reales de almacenamiento surgidas durante la ejecución del programa (tiempo de ejecución).<sup>1</sup>

La gestión de *memoria dinámica* surge como un mecanismo para que el propio programa, durante su ejecución (tiempo de ejecución), pueda solicitar (alojar) y liberar (desalojar) memoria según las necesidades surgidas durante una determinada ejecución, dependiendo de las circunstancias reales de cada momento de la ejecución del programa en un determinado entorno. Esta ventaja adicional viene acompañada por un determinado coste asociado a la mayor complejidad que requiere su gestión, ya que en el caso de las variables automáticas, es el propio compilador el encargado de su gestión, sin embargo en el caso de las *variables dinámicas* es el propio programador el que debe, mediante código software, gestionar el tiempo de vida de cada variable dinámica, cuando debe ser alojada y creada, como será utilizada, y finalmente cuando debe ser destruida y desalojada. Adicionalmente, como parte de esta gestión de la memoria dinámica por el propio programador, la memoria dinámica pasa a ser un *recurso* que debe gestionar el programador, y se debe preocupar de su alojamiento y de su liberación, poniendo especial cuidado y énfasis en no perder recursos (perder zonas de memoria sin liberar y sin capacidad de acceso).

---

<sup>1</sup>En realidad esto no es completamente cierto, ya que en el caso de subprogramas recursivos, cada invocación recursiva en tiempo de ejecución tiene la capacidad de alojar nuevas variables que serán posteriormente desalojadas automáticamente cuando la llamada recursiva finaliza.

## 12.1. Punteros

El *tipo puntero* es un tipo *simple* que permite a un determinado programa acceder a posiciones concretas de memoria, y más específicamente a determinadas zonas de la memoria dinámica. Aunque el lenguaje de programación C++ permite otras utilizaciones más diversas del tipo puntero, en este capítulo sólo se utilizará el tipo puntero para acceder a zonas de memoria dinámica.

Así, una determinada variable de tipo puntero apunta (o referencia) a una determinada entidad (variable) de un determinado tipo alojada en la zona de memoria dinámica. Por lo tanto, para un determinado tipo puntero, se debe especificar también el tipo de la variable (en memoria dinámica) a la que apunta, el cual define el espacio que ocupa en memoria y las operaciones (y métodos) que se le pueden aplicar, entre otras cosas.

De este modo, cuando un programa gestiona la memoria dinámica a través de punteros, debe manejar y gestionar por una parte la propia variable de tipo puntero, y por otra parte la variable dinámica apuntada por éste.

Un tipo puntero se define utilizando la palabra reservada **typedef** seguida del tipo de la variable dinámica apuntada, un asterisco para indicar que es un **puntero** a una variable de dicho tipo, y el identificador que denomina al tipo. Por ejemplo:



```
typedef int* PInt ;           // Tipo Puntero a Entero

struct Persona {             // Tipo Persona
    string nombre ;
    string telefono ;
    int edad ;
} ;
typedef Persona* PPersona ; // Tipo Puntero a Persona
```



Así, el tipo PInt es el tipo de una variable que apunta a una variable dinámica de tipo `int`. Del mismo modo, el tipo PPersona es el tipo de una variable que apunta a una variable dinámica de tipo `Persona`.

Es importante remarcar que el *tipo puntero*, en sí mismo, es un **tipo simple**, aunque el tipo apuntado puede ser tanto un tipo simple, como un tipo compuesto.

Es posible definir variables de los tipos especificados anteriormente. Nótese que estas variables (`p1` y `p2` en el siguiente ejemplo) son variables automáticas (gestionadas automáticamente por el compilador), es decir, se crean automáticamente (con un valor indeterminado) al entrar el flujo de ejecución en el ámbito de visibilidad de la variable, y posteriormente se destruyen automáticamente cuando el flujo de ejecución sale del ámbito de visibilidad de la variable. Por otra parte, las variables apuntadas por ellos son variables dinámicas (gestionadas por el programador), es decir el programador se encargará de solicitar la memoria dinámica cuando sea necesaria y de liberarla cuando ya no sea necesaria, durante la ejecución del programa. En el siguiente ejemplo, si las variables se definen sin inicializar, entonces tendrán un valor inicial inespecificado:

```
int main()
{
    PInt p1 ;           p1: 
    PPersona p2 ;       p2: 
}
```

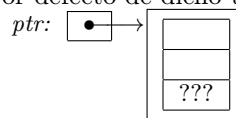
La constante `NULL` es una constante especial de tipo puntero que indica que una determinada variable de tipo puntero no apunta a nada, es decir, especifica que la variable de tipo puntero que contenga el valor `NULL` no apunta a ninguna zona de la memoria dinámica. Para utilizar la constante `NULL` se debe incluir la biblioteca estándar `<cstdlib>`. Así, se pueden definir las variables `p1` y `p2` e inicializarlas a un valor indicando que no apuntan a nada.

```
#include <cstdlib>
int main()
{
    PInt p1 = NULL ;     p1: 
    PPersona p2 = NULL ; p2: 
}
```

## 12.2. Gestión de Memoria Dinámica

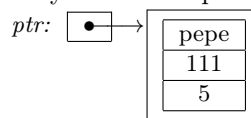
La memoria dinámica la debe gestionar el propio programador, por lo que cuando necesite crear una determinada variable dinámica, debe *solicitar memoria dinámica* con el operador **new** seguido por el tipo de la variable dinámica a crear. Este operador (**new**) realiza dos acciones principales, primero **aloja** (reserva) espacio en memoria dinámica para albergar a la variable, y después **crea** (invocando al constructor especificado) el contenido de la variable dinámica. Finalmente, a la variable **ptr** se le asigna el valor del puntero (una dirección de memoria) que apunta a la variable dinámica creada por el operador **new**. Por ejemplo, para crear una variable dinámica del tipo **Persona** definido anteriormente utilizando el constructor por defecto de dicho tipo.

```
int main()
{
    PPersona ptr ;
    ptr = new Persona ;
}
```



En caso de que el tipo de la variable dinámica tenga otros constructores definidos, es posible utilizarlos en la construcción del objeto en memoria dinámica. Por ejemplo, suponiendo que el tipo **Persona** tuviese un constructor que reciba el nombre, teléfono y edad de la persona:

```
int main()
{
    PPersona ptr ;
    ptr = new Persona("pepe", "111", 5) ;
}
```

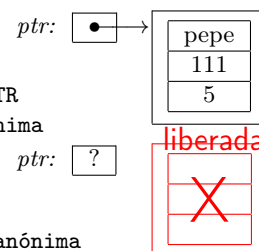


Posteriormente, tras manipular adecuadamente, según las características del programa, la memoria dinámica alojada, llegará un momento en que dicha variable dinámica ya no sea necesaria, y su tiempo de vida llegue a su fin. En este caso, el programador debe *liberar* explícitamente dicha variable dinámica mediante el operador **delete** de la siguiente forma:

```
int main()
{
    PPersona ptr ;          // Creación automática de la variable PTR
    ptr = new Persona ;     // Creación de la variable dinámica anónima

    // manipulación ...

    delete ptr ;           // Destrucción de la variable dinámica anónima
    // Destrucción automática de la variable PTR
}
```



La sentencia **delete ptr** realiza dos acciones principales, primero **destruye** la variable dinámica (invocando a su destructor), y después **desaloja** (libera) la memoria dinámica reservada para dicha variable. Finalmente la variable local **ptr** queda con un valor inespecificado, y será destruida automáticamente por el compilador cuando el flujo de ejecución salga de su ámbito de declaración.

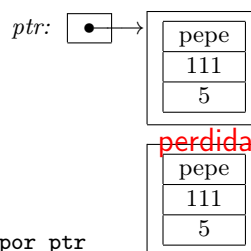
Si se ejecuta la operación **delete** sobre una variable de tipo puntero que tiene el valor **NULL**, entonces esta operación no hace nada.

En caso de que no se libere (mediante el operador **delete**) la memoria dinámica apuntada por la variable **ptr**, y esta variable sea destruida al terminar su tiempo de vida (su ámbito de visibilidad), entonces se *perderá* la memoria dinámica a la que apunta, con la consiguiente **pérdida de recursos** que ello conlleva.

```
int main()
{
    PPersona ptr ;
    ptr = new Persona("pepe", "111", 5) ;

    // manipulación ...

    // no se libera la memoria dinámica apuntada por ptr
    // se destruye la variable local ptr
}
```



## 12.3. Operaciones con Variables de Tipo Puntero

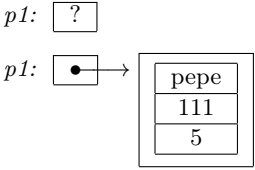
### Asignación de Variables de Tipo Puntero

El puntero nulo (NULL) se puede asignar a cualquier variable de tipo puntero. Por ejemplo:

```
int main()
{
    PPersona p1 ;           p1: [ ? ]
    // ...
    p1 = NULL ;             p1: [ / ]
    // ...
}
```

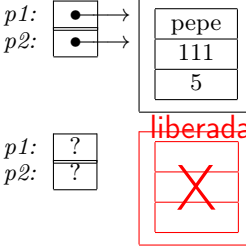
El resultado de crear una variable dinámica con el operador **new** se puede asignar a una variable de tipo puntero al tipo de la variable dinámica creada. Por ejemplo:

```
int main()
{
    PPersona p1 ;           p1: [ ? ]
    // ...
    p1 = new Persona("pepe", "111", 5) ;
    // ...
}
```



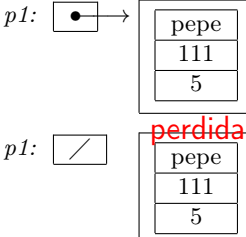
Así mismo, a una variable de tipo puntero se le puede asignar el valor de otra variable puntero. En este caso, ambas variables de tipo puntero apuntarán a la misma variable dinámica, que será compartida por ambas. Si se libera la variable dinámica apuntada por una de ellas, la variable dinámica compartida se destruye, su memoria se desaloja y ambas variables locales de tipo puntero quedan con un valor inespecificado.

```
int main()
{
    PPersona p1 = new Persona("pepe", "111", 5) ;
    PPersona p2 ;
    // ...
    p2 = p1 ;
    // ...
    delete p1 ;
}
```



En la operación de asignación, el valor anterior que tuviese la variable de tipo puntero se pierde, por lo que habrá que tener especial cuidado de que no se pierda la variable dinámica que tuviese asignada, si tuviese alguna.

```
int main()
{
    PPersona p1 = new Persona("pepe", "111", 5) ;
    // ...
    p1 = NULL ; // se pierde el valor anterior
    // ...
    delete p1 ;
}
```



### Desreferenciación de una Variable de Tipo Puntero

Para acceder a una variable dinámica apuntada por una variable de tipo puntero, se utiliza el operador unario *asterisco* (\*) precediendo al nombre de la variable de tipo puntero a través de la cual es apuntada. Por ejemplo, si **ptr** es una variable local de tipo puntero que apunta a una variable dinámica de tipo **Persona**, entonces **\*ptr** es la variable dinámica apuntada, y se trata de igual forma que cualquier otra variable de tipo **Persona**.

```

int main()
{
    PPersona ptr = new Persona("pepe", "111", 5) ;

    Persona p = *ptr ; // Asigna el contenido de la variable dinámica a la variable p

    *ptr = p ;          // Asigna el contenido de la variable p a la variable dinámica

    delete ptr ;        // destruye la variable dinámica y libera su espacio de memoria
}

```

Sin embargo, si una variable de tipo puntero tiene el valor `NULL`, entonces *desreferenciar* la variable produce un **error** en tiempo de ejecución que aborta la ejecución del programa. Así mismo, *desreferenciar* un puntero con valor inespecificado produce un comportamiento **anómalo** en tiempo de ejecución.

Es posible, así mismo, acceder a los elementos de la variable apuntada mediante el operador de desreferenciación. Por ejemplo:

```

int main()
{
    PPersona ptr = new Persona ;

    (*ptr).nombre = "pepe" ;
    (*ptr).telefono = "111" ;
    (*ptr).edad = 5 ;

    delete ptr ;
}

```

Nótese que el uso de los paréntesis es obligatorio debido a que el operador punto (.) tiene mayor precedencia que el operador de desreferenciación (\*). Por ello, en el caso de acceder a los campos de un registro en memoria dinámica a través de una variable de tipo puntero, es más adecuado utilizar el operador de desreferenciación (->). Por ejemplo:

```

int main()
{
    PPersona ptr = new Persona ;

    ptr->nombre = "pepe" ;
    ptr->telefono = "111" ;
    ptr->edad = 5 ;

    delete ptr ;
}

```

Este operador también se utiliza para invocar a métodos de un objeto si éste se encuentra alojado en memoria dinámica. Por ejemplo:

```

#include <iostream>
using namespace std ;
class Numero {
public:
    Numero(int v) : val(v) {}
    int valor() const { return val ; }
private:
    int val ;
} ;
typedef Numero* PNumero ;
int main()
{

```

```

PNumero ptr = new Numero(5) ;

cout << ptr->valor() << endl ;

delete ptr ;
}

```

## Comparación de Variables de Tipo Puntero

Las variables del mismo tipo puntero se pueden comparar entre ellas por igualdad (==) o desigualdad (!=), para comprobar si apuntan a la misma variable dinámica. Así mismo, también se pueden comparar por igualdad o desigualdad con el puntero nulo (NULL) para saber si apunta a alguna variable dinámica, o por el contrario no apunta a nada. Por ejemplo:

```

int main()
{
    PPersona p1, p2 ;
    // ...
    if (p1 == p2) {
        // ...
    }
    if (p1 != NULL) {
        // ...
    }
}

```

## 12.4. Paso de Parámetros de Variables de Tipo Puntero

El tipo puntero es un *tipo simple*, y por lo tanto se tratará como tal. En caso de paso de parámetros de tipo puntero, si es un parámetro de entrada, entonces se utilizará el *paso por valor*, y si es un parámetro de salida o de entrada/salida, entonces se utilizará el *paso por referencia*.

Hay que ser consciente de que un parámetro de tipo puntero puede apuntar a una variable dinámica, y en este caso, a partir del parámetro se puede acceder a la variable apuntada.

Así, si el parámetro se pasa *por valor*, entonces se copia el valor del puntero del parámetro actual (en la invocación) al parámetro formal (en el subprograma), por lo que ambos apuntarán a la misma variable dinámica compartida, y en este caso, si se modifica el valor almacenado en la variable dinámica, este valor se verá afectado, así mismo, en el exterior del subprograma, aunque el parámetro haya sido pasado por valor.

Por otra parte, las funciones también pueden devolver valores de tipo puntero.

```

void modificar(PPersona& p) ;

PPersona buscar(PPersona l, const string& nombre) ;

```

## 12.5. Listas Enlazadas Lineales

Una de las principales aplicaciones de la Memoria Dinámica es el uso de estructuras enlazadas, de tal forma que un campo o atributo de la variable dinámica es a su vez también de tipo puntero, por lo que puede apuntar a otra variable dinámica que también tenga un campo o atributo de tipo puntero, el cual puede volver a apuntar a otra variable dinámica, y así sucesivamente, tantas veces como sea necesario, hasta que un puntero con el valor NULL indique el final de la estructura enlazada (lista enlazada).

Así, en este caso, vemos que un campo de la estructura es de tipo puntero a la propia estructura, por lo que es necesario definir el tipo puntero antes de definir la estructura. Sin embargo, la estructura todavía no ha sido definida, por lo que no se puede definir un puntero a ella. Por ello es necesario realizar una *declaración adelantada* de un *tipo incompleto* del tipo de la variable