FACULTAD DE INFORMÁTICA

Objetos y memoria dinámica

RESUMEN

Programación orientada a objetos — Unidad 8

Autor: Luis Hernández Yáñez

Introducción a la memoria dinámica

Mecanismo de gestión de memoria en tiempo de ejecución.

Ajuste de las necesidades de memoria a cada ejecución concreta.

Cuando se necesita memoria para una variable se solicita ésta al Sistema de gestión dinámica de memoria, quien reserva la cantidad adecuada para el tipo de variable y devuelve la dirección de la primera celda de memoria de la zona reservada.

Cuando ya no se necesita más la variable, se libera la memoria que utilizaba indicando al Sistema de gestión dinámica de memoria que puede contar de nuevo con la memoria que se había reservado anteriormente. Para "devolver" esa memoria se proporciona la dirección de la primera celda de la zona reservada.

Dos tipos de datos en los programas:

- ✓ Datos estáticos: se les asigna la memoria que necesitan antes de empezar la ejecución.
- ✓ Datos dinámicos: se les asigna memoria durante la ejecución.

Variables estáticas frente a variables dinámicas

Datos estáticos (variables estáticas) (no confundir con **static**)

- ✓ Variables declaradas como de un tipo concreto: `int i;`
- ✓ Su(s) dato(s) se accede(n) directamente con la propia variable `cout << i;`
- ✓ Existen durante todo el tiempo de ejecución de su ámbito: se les asigna una zona de la memoria principal al comenzar la ejecución y se libera esa memoria al terminar la ejecución.

Datos dinámicos (variables dinámicas)

- ✓ Variables accedidas a través de su dirección de memoria (dirección de la primera celda de memoria utilizada).
- ✓ Se necesita tener guardada esa dirección de memoria inicial en algún otro lugar: en un **puntero**.
- ✓ Los punteros guardan direcciones de memoria y sirven para acceder a las variables dinámicas.

Los datos estáticos también se pueden acceder a través de punteros.

Punteros

LOS PUNTEROS SON DIRECCIONES DE MEMORIA

La variable a la que apunta un puntero, al igual que cualquier otra variable, debe ser de un tipo concreto.

```
tipo *nombre;
```

El puntero denominado *nombre* apuntará a una variable del *tipo* indicado (el tipo base del puntero).

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

Las variables puntero no se inicializan, por lo que tras declararlas contienen direcciones que no son válidas.

Un puntero *puede* apuntar a cualquier variable de su tipo base.

Un puntero *no tiene por qué* apuntar necesariamente a una variable (puede no apuntar a nada).

Operadores para punteros

El operador & (*la dirección de*)

Devuelve la dirección de memoria de la variable a la que se aplica.

```
int i;
int *p;
```

A un puntero se le puede asignar la dirección de una variable del tipo base.

```
p = &i; // la dirección de i
```

Ahora, el puntero **p** ya contiene una dirección de memoria válida.

El operador * (*en la dirección*)

Accede a lo que hay en la dirección de memoria a la que se aplica.

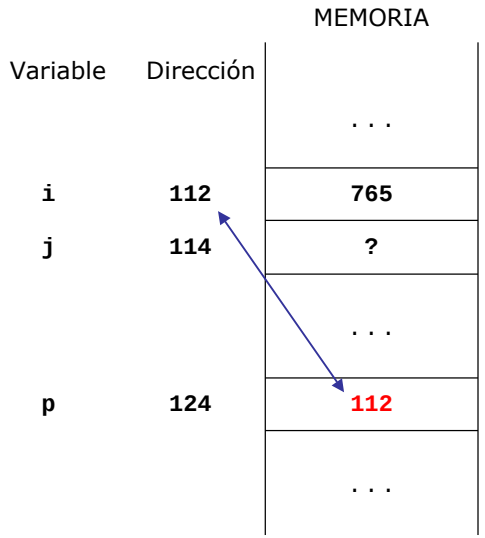
Una vez que un puntero contiene una dirección de memoria válida, se puede acceder al dato al que apunta con este operador.

```
p = &i; cout << *p;
```

Como el puntero **p** contiene la dirección de memoria de la variable **i**, ***p** accede al contenido de esa variable **i**.

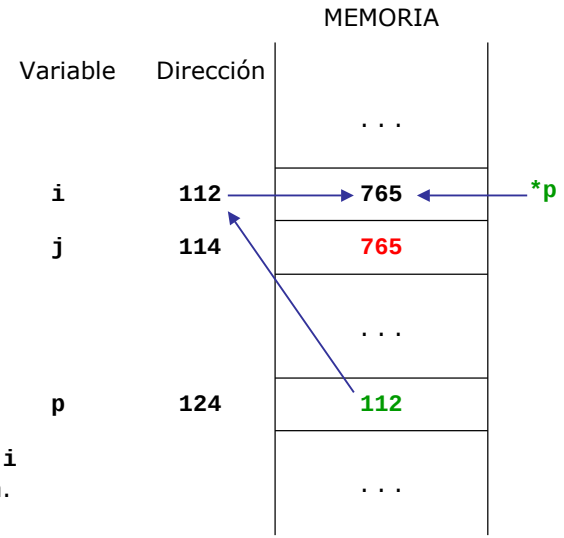
Operadores para punteros

```
int i, j;
...
int *p;
...
i = 765;
p = &i;
```



Operadores para punteros

```
int i, j;
...
int *p;
...
i = 765;
p = &i;
j = *p;
```



Esto se llama direccionamiento indirecto (*indirección*). Se accede al dato **i** de forma indirecta.

Punteros a estructuras u objetos

Cuando un puntero apunta a una estructura o a un objeto, para acceder a los miembros de la estructura u objeto se ha de utilizar el operador flecha (**->**), en lugar del operador punto:

```
struct Hora {
  int horas;
  int minutos;
  int segundos;
} unaHora;

Hora *p;

p = &unaHora;
unaHora.horas = 12;
cout << p->horas << endl;
```

Punteros y el modificador `const`

Cuando se declaran punteros con el modificador de acceso `const`, su efecto depende de dónde se coloque en la declaración:

`const tipo *puntero;` Puntero a una constante

`tipo *const puntero;` Puntero constante

```
int i[2] = { 12, 21 };
const int *p1 = i;
int *const p2 = i;

(*p1)++; // ERROR: puntero a una constante
p1++;
(*p2)++;
p2++;    // ERROR: puntero constante
```


El operador `new`

`new tipo` reserva memoria del montón para una variable de ese *tipo* y devuelve la primera dirección de memoria utilizada, que debe ser asignada a un puntero.

```
int *p; // todavía sin una dirección válida
p = new int; // ya tiene una dirección válida
*p = 12;
```

La variable dinámica se accede exclusivamente a través de punteros. No hay ningún identificador asociado con ella que permita accederla directamente.

```
int i; // i es una variable estática
int *p1, *p2;
p1 = &i; // puntero que da acceso a la variable
         // estática i (accesible con i o con *p1)
p2 = new int; // puntero que da acceso a una
              // variable dinámica
              // (accesible sólo con *p2)
```


El operador `delete`

`delete puntero` devuelve al montón la memoria utilizada por la variable dinámica apuntada por *puntero*.

```
int *p;
p = new int;
*p = 12;
...
delete p; // ya no se necesita el entero
          // apuntado por p
```

El puntero deja de contener una dirección válida y no debe ser utilizado hasta que no contenga nuevamente otra dirección válida.

Punteros y direcciones válidas - la constante `NULL`

Un puntero sólo debe ser utilizado, para acceder al dato al que apunte, si se está seguro de que contiene una dirección válida.

Un puntero NO contiene una dirección válida tras ser definido.

Un puntero obtiene una dirección válida:

- ✓ al asignarle otro puntero (con el mismo tipo base) que ya contenga una dirección válida.
- ✓ al asignarle la dirección de otra variable con el operador `&`.
- ✓ al asignarle la dirección devuelta por el operador `new`.
- ✓ al asignarle el valor `NULL` (que indica que se trata de un puntero nulo, un puntero que no apunta a nada).

```
int i;
int *q; // q no tiene aún una dirección válida
int *p = &i;
q = NULL;
q = p;
q = new int;
```

Formas en las que un puntero toma una dirección válida

Objetos dinámicos

```
#include <iostream>
#include <string>
using namespace std;

#include "persona.h"

int main()
{
  Persona *p;
  // p es un puntero a objeto Persona
1. Crear
  p = new Persona(); // Creado el objeto Persona dinámico
2. Usar
  p->leer();    // Uso del objeto pasándole mensajes
  p->mostrar(); // a través del puntero (operador flecha)
3. Destruir
  delete p; // Liberada la memoria del objeto dinámico
  return 0;
}
```

Uso de objetos dinámicos

Antes de usar el objeto dinámico, pasándole mensajes a través del puntero que lo apunta, se ha de crear el objeto con **new**.

Cuando ya no se necesita el objeto dinámico, se destruye con **delete**.

Objetos dinámicos

Creación y destrucción de objetos dinámicos

Los objetos dinámicos se crean como las variables dinámicas: con **new**. Pero a continuación de **new** colocamos la función constructora que queremos que se ejecute al crear el objeto.

Siempre que se crea un objeto dinámico se ejecuta automáticamente un constructor sobre el objeto dinámico. ¿Cuál? El que indiquemos:

```
p = new Persona();

p = new Persona("223344F", 30, "Juan", "Pérez Gómez");
```

Los objetos dinámicos se destruyen como las variables dinámicas: con **delete**.

```
delete p;
```

Siempre que se destruye un objeto dinámico se ejecuta antes el destructor sobre el objeto dinámico.

Atributos dinámicos

Simplemente otra forma de guardar en memoria los atributos.

```
class Circulo {
  ...
private:
  Punto _centro;
};
```

```
class Circulo {
  ...
private:
  Punto *_centro;
};
```

Se consideran ambos casos equivalentes, aunque con una implementación distinta.

Cada objeto de la clase **Circulo** dispone de su propio centro (objeto de la clase **Punto**), por lo que se ha de crear (**new**) el atributo dinámico al crear los objetos de esa clase **Circulo** (constructores) y se ha de liberar el atributo dinámico (**delete**) al destruir los objetos de esa clase **Circulo**.

Además, el operador de asignación ha de tener en cuenta que ya existe el atributo dinámico en el objeto de destino.

Manipulación de objetos con atributos dinámicos

El archivo **punto.h**

```
#ifndef punto_h
#define punto_h

// Todos los métodos implementados fuera
class Punto {
public:
  Punto(int = 0, int = 0); // Constructor (predeterminado)
  Punto(const Punto&); // Constructor de copia
  Punto& operator=(const Punto&) // Operador de asignación
  ~Punto(); // Destructor
  int x() const;
  int y() const;
  void x(int);
  void y(int);
private:
  int *_x, *_y;
};

#endif
```

Manipulación de objetos con atributos dinámicos

El archivo **punto.cpp**

```
#include "punto.h"

Punto::Punto(int x, int y) {
  _x = new int; *_x = x;
  _y = new int; *_y = y;
}

Punto::Punto(const Punto& p) {
  _x = new int; *_x = *p._x;
  _y = new int; *_y = *p._y;
}

Punto& Punto::operator=(const Punto& p) {
  *_x = *p._x; *_y = *p._y; return *this;
}

Punto::~Punto() { delete _x; delete _y; }
```

El constructor crea los atributos dinámicos

El constructor de copia crea los atributos dinámicos

El operador de asignación tiene en cuenta que ya existen los atributos

El destructor destruye los atributos dinámicos

(*continúa*)

Manipulación de objetos con atributos dinámicos

```
int Punto::x() const { return *_x; }

int Punto::y() const { return *_y; }

void Punto::x(int i) { *_x = i; }

void Punto::y(int i) { *_y = i; }
```

Los accedentes y mutadores trabajan con lo apuntado por el atributo puntero (con el atributo dinámico)

Manipulación de objetos con atributos dinámicos

El archivo **circulo.h**

```
#ifndef circulo_h
#define circulo_h
#include "punto.h" // Inclusión de la clase Punto

class Circulo {
public:
  Circulo(int radio = 0); // Constructor (predeterminado)
  Circulo(const Circulo&); // Constructor de copia
  Circulo& operator=(const Circulo&); // Asignación
  ~Circulo(); // Destructor
  int radio() const;
  Punto centro() const;
  void radio(int);
  void centro(Punto);
private:
  int *_radio;
  Punto *_centro;
};

#endif
```


Manipulación de objetos con atributos dinámicos

El archivo **circulo.cpp**

```
#include "circulo.h"

Circulo::Circulo(int radio) {
  _radio = new int; *_radio = radio;
  _centro = new Punto();
}

Circulo::Circulo(const Circulo& c) {
  _radio = new int; *_radio = *c._radio;
  _centro = new Punto(); *_centro = *c._centro;
}
```

Constructor de la clase **Punto**

Operador de asignación de la clase **Punto**

Constructor de la clase **Punto**

(*continúa*)

Manipulación de objetos con atributos dinámicos

```
Circulo& Circulo::operator=(const Circulo& c) {
  *_radio = *c._radio; *_centro = *c._centro;
  return *this;
}
```

Operador de asignación de la clase **Punto**

```
Circulo::~Circulo() { delete _radio; delete _centro; }
```

Destructor de la clase **Punto**

```
int Circulo::radio() const { return *_radio; }

Punto Circulo::centro() const { return *_centro; }
```

Constructor de copia de la clase **Punto**

```
void Circulo::radio(int r) { *_radio = r; }

void Circulo::centro(Punto p) { *_centro = p; }
```

Constructor de copia de la clase **Punto**

Operador de asignación de la clase **Punto**

Atributos relaciones

El archivo **alumno.h**

```
#ifndef alumno_h
#define alumno_h

#include "persona.h" // Inclusión de la clase Persona

class Alumno : public Persona {
public:
  // Constructor (predeterminado):
  Alumno(Persona* = NULL, string = "", int = 0,
         string = "", string = "", int = 1);
  Alumno(const Alumno&); // Constructor de copia
  Alumno& operator=(const Alumno&); // Asignación
  ~Alumno(); // Destructor
  void curso(int);
  void profesor(Persona*);
  int curso() const;
  Persona* profesor() const;
  void leer();
  void mostrar() const;
```

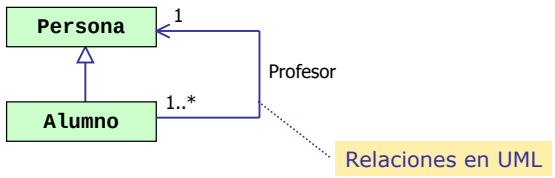
Los constructores no deben crear el objeto con el que están relacionados, como tampoco debe ser liberado en el destructor. Y el operador de asignación copiará la relación (punteros).

(*continúa*)

Atributos relaciones

```
private:
  int _curso;
  Persona *_profesor; // RELACIÓN con una Persona
};                    // La que es profesor de ésta

#endif
```



Relaciones en UML

Atributos relaciones

El archivo **alumno.cpp**

```
#include "alumno.h"
#include <iostream>
#include <string>
using namespace std;

Alumno::Alumno(Persona* profe, string nif, int edad,
   string nombre, string apellidos, int curso) :
   _curso(curso), _profesor(profe),
  Persona(nif, edad, nombre, apellidos) { }

Alumno::Alumno(const Alumno& otro) : Persona(otro)
{
  _curso = otro._curso;
  _profesor = otro._profesor;
}
```

Los atributos relaciones se manejan igual que los atributos que no son punteros. No requieren operaciones de reserva (**new**) o liberación (**delete**) de memoria.

(*continúa*)

Atributos relaciones

```
Alumno& Alumno::operator=(const Alumno& otro)
{
  Persona::operator=(otro);
  _curso = otro._curso;
  _profesor = otro._profesor;
  return *this;
}

Alumno::~Alumno() {}

void Alumno::curso(int num) { _curso = num; }

void Alumno::profesor(Persona* profe)
{ _profesor = profe; }

int Alumno::curso() const { return _curso; }

Persona* Alumno::profesor() const { return _profesor; }
```

No hay **delete**

(*continúa*)

Atributos relaciones

```
void Alumno::mostrar() const
{
  Persona::mostrar();
  cout << "Curso: " << _curso << endl;
  cout << "Profesor:" << endl;
  if(_profesor == NULL) cout << "No asignado" << endl;
  else _profesor->mostrar();
}

void Alumno::leer()
{
  Persona::leer();
  cout << "Curso: ";
  cin >> _curso;
}
```

Los datos del **_profesor** ya se habrán obtenido en alguna otra parte