FACULTAD DE INFORMÁTICA

Memoria dinámica

SUPLEMENTO

Programación orientada a objetos — Unidad 8

Autor: Luis Hernández Yáñez

FdI UCM

Contenido

FdI UCM

Declaración de punteros

El asterisco (**`*`**) que se utiliza en la declaración de una variable puntero se puede colocar pegado al tipo base, pegado al nombre del puntero o en medio:

```
int* p1;
int *p2;
int * p3;
```

Las tres formas son válidas.

La segunda forma (**`int *p2;`**) deja claro que es la variable que se declara la que es un puntero. Es preferible. En una sola línea de declaración se pueden declarar punteros (cada uno con su asterisco) y (en este caso) enteros:

```
int *p1, i, *p2, *p3, k;
```

En las listas de parámetros en las que omitimos los nombres de los parámetros (prototipos) pegaremos el asterisco al tipo, ya que con cada tipo se declara un único parámetro.

FdI UCM

Punteros y arrays

Los arrays y los punteros están muy relacionados.

El nombre de una variable array *es* la dirección base del array, la dirección de la primera celda de memoria utilizada por el array (donde comienza el primer elemento del array).

Entonces,

```
int diasmes[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
cout << *diasmes << endl;
```

muestra 31 en la pantalla (el primer elemento).

Al resto de los elementos del array, además de por índice, se puede acceder por medio de las operaciones aritméticas de punteros.

Aritmética de punteros

Para recorrer el array con aritmética de punteros podemos utilizar la propia variable array (sin indexar) o punteros auxiliares (declarados sobre el mismo tipo base que el array):

```
int diasmes[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
int *punt;
punt = diasmes;
cout << *punt << endl;
```

Cuando se aplican operaciones aritméticas a los punteros, éstas no trabajan necesariamente en incrementos de una unidad:

```
punt++;
```

El puntero no pasa a contener la siguiente dirección, sino la dirección del siguiente dato.

En la aritmética de punteros la unidad de cálculo es el número de celdas de memoria (bytes, habitualmente) que necesita el tipo base (por ejemplo, 1 para `char`, 2 para `int`, etcétera).

Aritmética de punteros

```
int diasmes[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
int *punt;
punt = diasmes;
punt++;
cout << *punt << endl;
```

Muestra **28** en la pantalla (segundo elemento del array).

```
punt = diasmes;
punt += 4;
cout << *punt << endl;
```

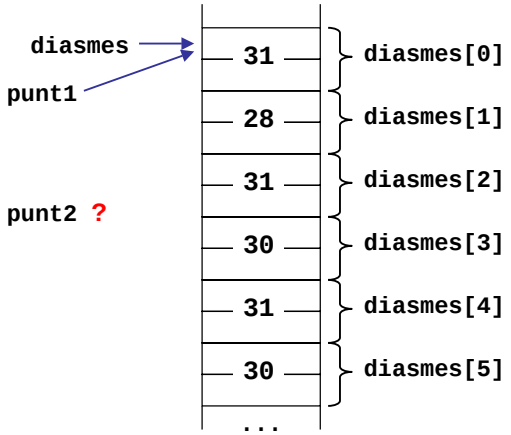
Muestra **31** en la pantalla (5º elemento: 4 más allá del primero).

Operaciones aplicables:

- ✓ Incremento (`++`) y decremento (`--`)
- ✓ Suma de puntero y entero (y la abreviatura `+=`)
- ✓ Diferencia de punteros (número de datos entre las dos direcciones)

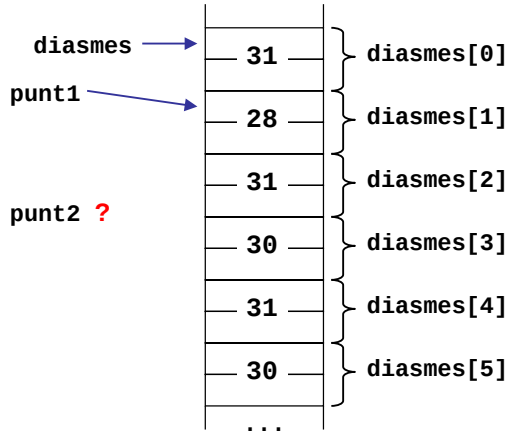
Aritmética de punteros

```
int diasmes[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
int *punt1, *punt2;
punt1 = diasmes;
```



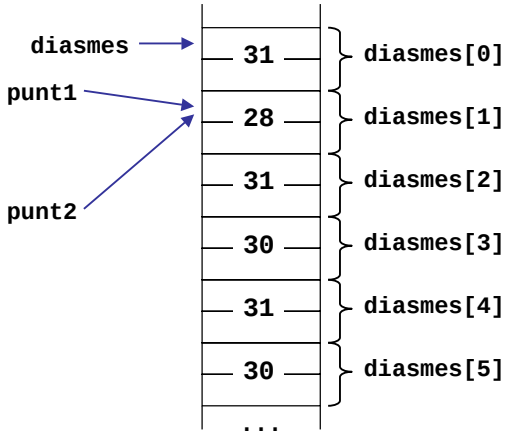
Aritmética de punteros

```
int diasmes[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
int *punt1, *punt2;
punt1 = diasmes;
punt1++;
```



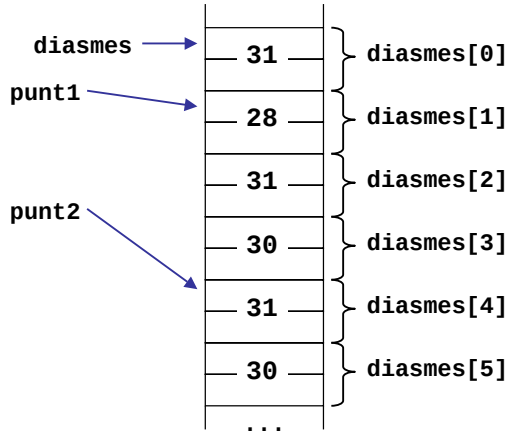
Aritmética de punteros

```
int diasmes[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
int *punt1, *punt2;
punt1 = diasmes;
punt1++;
punt2 = punt1;
```



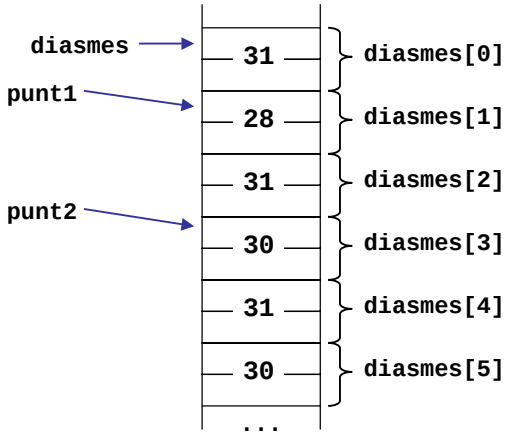
Aritmética de punteros

```
int diasmes[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
int *punt1, *punt2;
punt1 = diasmes;
punt1++;
punt2 = punt1;
punt2 += 3;
```



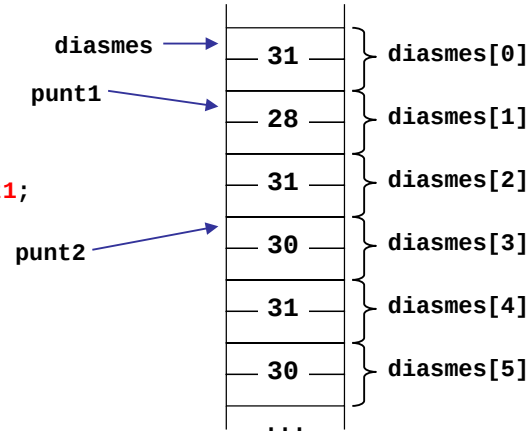
Aritmética de punteros

```
int diasmes[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
int *punt1, *punt2;
punt1 = diasmes;
punt1++;
punt2 = punt1;
punt2 += 3;
punt2--;
```



Aritmética de punteros

```
int diasmes[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
int *punt1, *punt2;
punt1 = diasmes;
punt1++;
punt2 = punt1;
punt2 += 3;
punt2--;
cout << punt2 - punt1;
// muestra 2
```



Uso del nombre de un array como puntero

El nombre de un array, como ya se ha dicho, es un puntero al primer elemento del array.

Sin embargo, es un puntero constante cuyo contenido (dirección) no se puede modificar:

```
int diasmes[12] = {31,28,31,30,31,30,31,31,30,31,30,31};

cout << *(diasmes++) << endl; // ERROR
```

Se puede utilizar en expresiones aritméticas siempre y cuando las operaciones no intenten modificar su valor (como ocurre en el caso anterior).

```
int diasmes[12] = {31,28,31,30,31,30,31,31,30,31,30,31};

cout << *(diasmes+1) << endl; // CORRECTO
```

Lo más adecuado es utilizar punteros auxiliares, como en los ejemplos de las páginas anteriores.

Paso de arrays a funciones

En la Unidad 4 vimos que las funciones sólo pueden recibir la dirección de un array, y no una copia.

Como el nombre del array es la dirección base del array (la de su primer elemento), en las llamadas basta utilizar ese nombre.

La función tiene que estar preparada para aceptar la dirección del array. El parámetro correspondiente se puede declarar como array, delimitado o no:

```
void cuadrado(int array[10]);
```

o

```
void cuadrado(int array[]);
```

Como se pasará la dirección del array como argumento, el parámetro también se puede declarar como puntero:

```
void cuadrado(int *array);
```


Paso de arrays a funciones

Independientemente de la forma en la que se haya declarado el parámetro, el código de la función lo puede manipular como un array o como un puntero:

```
int array[10]
int array[]
int *array
```

```
void cuadrado(...);
{
  for(int i = 0; i < 10; i++)
    array[i] *= array[i];
}
```

o

```
void cuadrado(...);
{
  for(int i = 0; i < 10; i++)
    *array++ *= *array;
}
```


Arrays dinámicos

Los operadores **new** y **delete** contemplan una forma alternativa de invocación que sirve para crear y destruir arrays dinámicos (arrays mantenidos en el montón de memoria dinámica y accedidos a través de un puntero que apunte al primer elemento):

```
...
Punto *punt;
punt = new Punto[10];
for(int i = 0; i < 10; i++) punt[i].x(i);
for(int i = 0; i < 10; i++) cout << punt[i].x(i) << endl;
delete [] punt;
```

Si, como en este caso, el array es un array de objetos:

- ✓ **new** ejecuta el constructor predeterminado sobre todos los objetos del array.
- ✓ **delete** ejecuta el destructor sobre todos los objetos del array.

Punteros y parámetros de funciones

Paso de parámetros por referencia: Utilizamos referencias.

```
void cuadrado(int& i)
{
  i = i*i;
}
```

Ejemplo de llamada:

```
int ent = 12;
cuadrado(ent);
```

Aunque ese es el mecanismo preferido,
los punteros proporcionan un mecanismo alternativo.

Si se utiliza un puntero como argumento de un parámetro por valor, el contenido del puntero (su dirección) no se modifica, pero nada impide que se modifique aquello a lo que apunta.

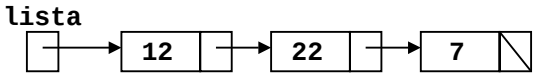
```
void cuadrado(int *p)
{
  *p = (*p)*(*p);
}
```

Ejemplo de llamada:

```
int ent = 12;
cuadrado(&ent);
```

Este es el mecanismo tradicional de C.

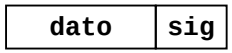
Otra implementación de listas: las listas enlazadas



En lugar de tener localizado el siguiente elemento en la siguiente posición de almacenamiento (array), los elementos ahora se guardan de forma dispersa, por lo que resulta necesario que cada uno vaya acompañado de información que permita localizar el siguiente.

Los punteros nos brindan una forma muy sencilla de localizar al siguiente elemento de la lista: apuntándolo.

Así, a cada elemento se le adjunta un enlace (puntero).
El dato junto con el enlace se conoce como **nodo** de la lista:

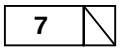


Necesitamos algún tipo de estructura que implemente los nodos.

Otra implementación de listas: las listas enlazadas

```
struct Nodo {
  int dato; // lista de enteros, por ejemplo
  Nodo *sig;
};
```

Por supuesto, si no hay elemento siguiente (caso del último), el puntero no ha de apuntar a nada: ha de valer **NULL**.



Tan sólo queda tener localizado el primer elemento (si existe).
Lo hacemos con un puntero aparte, que será el único atributo que se necesita en la clase **Lista**:

```
class Lista { ...
private:
  Nodo *_primero;
};
```

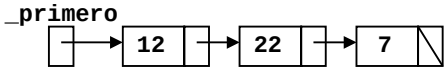

Otra implementación de listas: las listas enlazadas

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

struct Nodo {
  int dato;
  Nodo *sig;
};

class Lista {
public:
  Lista();
  ~Lista();
  void insertar(int); // Por el principio
  void mostrar();
private:
  Nodo *_primero;
};
```

Obviamos la mayoría de los métodos de la clase para simplificar el ejemplo y centrarlo en lo más significativo

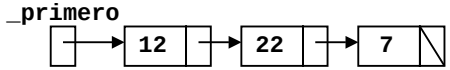


Otra implementación de listas: las listas enlazadas

```
Lista::Lista() { _primero = NULL; }

Lista::~Lista() {
  Nodo *tmp;
  while(_primero != NULL) {
    tmp = _primero;
    _primero = _primero->sig;
    delete tmp;
  }
}

void Lista::insertar(int d)
{
  Nodo *nuevo = new Nodo;
  nuevo->dato = d;
  nuevo->sig = _primero;
  _primero = nuevo;
}
```

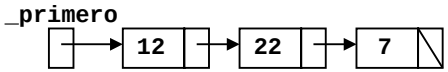


Otra implementación de listas: las listas enlazadas

```
void Lista::mostrar()
{
  Nodo *actual = _primero;
  while(actual != NULL) {
    cout << actual->dato << endl;
    actual = actual->sig;
  }
}

int main()
{
  Lista lista;
  lista.insertar(7);
  lista.insertar(22);
  lista.insertar(12);
  lista.mostrar();

  return 0;
}
```



Los nodos como objetos

En la implementación anterior de lista enlazada resulta extraño que los nodos se implementen como estructuras en lugar de definir una clase para ellos.

Podemos modificar la implementación para que los nodos sean objetos de una clase **Nodo**:

```
class Nodo {
public:
  Nodo(int, Nodo*);
  // Al crear el nodo se colocan en él tanto el dato
  // como el enlace.
  ~Nodo();
  void mostrar();
private:
  int _dato;
  Nodo *_sig;
}
```


Los nodos como objetos

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

class Nodo {
public:
  Nodo(int, Nodo*);
  // Al crear el nodo se colocan en él tanto el dato
  // como el enlace.
  ~Nodo();
  void mostrar();
private:
  int _dato;
  Nodo *_sig;
}

Nodo::Nodo(int d = 0, Nodo *s = NULL)
  : _dato(d), _sig(s) { }
```

Obviamos la mayoría de los métodos de la clase para simplificar el ejemplo y centrarlo en lo más significativo

Los nodos como objetos

```
Nodo::~Nodo() { if(_sig != NULL) delete _sig; }

void Nodo::mostrar() {
  cout << _dato << " ";
  if(_sig != NULL) _sig->mostrar(); }

class Lista {
public:
  Lista();
  ~Lista();
  void insertar(int); // Por el principio
  void mostrar();
private:
  Nodo *_primero;
};

Lista::Lista() { _primero = NULL; }

Lista::~Lista() { if(_primero != NULL) delete _primero; }
```


Los nodos como objetos

```
void Lista::insertar(int d) {
  Nodo *tmp = new Nodo(d, _primero);
  _primero = tmp; }

void Lista::mostrar() {
  if(_primero != NULL) _primero->mostrar(); }

int main()
{
  Lista lista;
  lista.insertar(7);
  lista.insertar(22);
  lista.insertar(12);
  lista.mostrar();

  return 0;
}
```


Los nodos como objetos

Como se habrá observado, algunas de las funciones miembro son recursivas. Esto es así porque las listas se definen de forma recursiva.

Una lista es:

- ✓ Una lista vacía o
- ✓ Un elemento seguido de otra lista

Cuando se procesa una lista, básicamente se procesa el elemento actual (si existe) y se propaga el proceso hacia el resto de la lista (la lista que sigue a ese elemento).

Así, por ejemplo, cuando se quiere mostrar una lista, se pasa el mensaje **`mostrar()`** al objeto **`Lista`**. Se ejecuta el método:

```
void Lista::mostrar() {
  if(_primero != NULL) _primero->mostrar(); }
```

Si no hay lista, no hay nada que hacer. Si hay primer elemento, se pasa el mensaje mostrar al mismo, un objeto de clase **`Nodo`**.

Los nodos como objetos

Se ejecuta el método **`mostrar()`** de la clase **`Nodo`**:

```
void Nodo::mostrar() {
  cout << _dato << " ";
  if(_sig != NULL) _sig->mostrar(); }
```

Después de mostrar el dato del nodo, se pasa el mensaje **`mostrar()`** al siguiente nodo (si es que existe).
Se propaga el mensaje de forma recursiva.

Se procede de la misma forma con el destructor. Cuando se destruye un nodo (con **`delete`**), se ejecuta sobre él su destructor:

```
Nodo::~Nodo() { if(_sig != NULL) delete _sig; }
```

Antes de liberar la memoria se ejecuta ese código, que se encarga de destruir el nodo siguiente (si es que existe).