FACULTAD DE INFORMÁTICA

# Arrays y cadenas de caracteres

RESUMEN

Programación orientada a objetos — Unidad 4

Autor: Luis Hernández Yáñez

## FdI UCM Arrays unidimensionales en C++

Declaración de un array unidimensional:

```
tipo nombre[tamaño];
```

```
char p[10];
```

Todos los arrays tienen el 0 como primer índice:

```
p[0]  p[1]  p[2]  p[3]  ...  p[9]
```

> **!** No se comprueban los intentos de acceso a posiciones inexistentes de los arrays

```
#include <iostream>
using namespace std;
int main()
{
  int x[10]; /* 10 datos enteros */
  int t;
  for(t = 0; t < 10; t++) x[t] = t;
  for(t = 0; t < 10; t++) cout << x[t] << endl;
  return 0;
}
```



## FdI UCM Arrays de más de una dimensión

### Arrays bidimensionales

*tipo nombre*[*tamaño2*][*tamaño1*];

```
int d[10][20]
```

- ✓ No se utilizan comas para separar las dimensiones.
- ✓ Cada dimensión va en su propio par de corchetes.

```
d[3][12] = 132;
```

### Arrays multidimensionales

*tipo nombre*[*tamañoN*]...[*tamaño2*][*tamaño1*];

```
int m[4][3][6][5];
m[2][1][5][3] = 132;
```



## FdI UCM Inicialización de arrays

*tipo nombre*[*tamañoN*]..[*tamaño1*]={*lista_de_valores*};

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

```
int cuads[5][2] = {1,1, 2,4, 3,9, 4,16, 5,25};
```

| | 0 | 1 |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 2 | 4 |
| 2 | 3 | 9 |
| 3 | 4 | 16 |
| 4 | 5 | 25 |

## Paso de arrays a funciones

Por defecto, las funciones reciben los arrays mediante una simulación de paso de parámetro **por referencia**, sin que haya que poner & en la declaración del parámetro.

Es decir: las funciones sólo pueden recibir la dirección de un array, no una copia de un array.

Si se quiere pasar un array evitando que se modifique (equivalente a un paso de parámetro por valor), se debe utilizar `const`.

El nombre de una variable array se interpreta como la dirección del primer elemento del array (un puntero a su primer elemento).

Por tanto, cuando se pasa un array a una función se usa solamente su nombre, lo que constituye la dirección (puntero) que se espera para el parámetro de la función, al estar éste declarado por defecto como una referencia.



## Parámetros para aceptar argumentos arrays

Tres formas para declarar un parámetro que va a recibir un array (dirección del primer elemento):

- ✓ Declararlo como array con todas sus dimensiones:

```
void intro(double unidades[100]);
void intro(double ventas[12][31]);
void mostrar(const int notas[50]);
```

- ✓ Declararlo como array no delimitado (no se indica la primera dimensión):

```
void intro(double unidades[]);
void intro(double ventas[][31]);
void mostrar(const int notas[]);
```

- ✓ Declararlo como puntero:

```
void intro(double* unidades);
void intro(double* ventas);
void mostrar(const int* notas);
```



## Un ejemplo de arrays de estructuras

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

const int N = 10;
struct venta {
  int articulo;
  int unidades;
  double precio;
};

struct vendido {
  venta items[N];
  int cont;
};

void leer(venta&);
void mostrar(vendido);
double total(vendido);
```

(*continúa*)



## Un ejemplo de arrays de estructuras

```
void leer(venta& unaVenta)
{
  cout << "Código del artículo: ";
  cin >> unaVenta.articulo;
  cout << "Número de unidades: ";
  cin >> unaVenta.unidades;
  cout << "Precio del artículo: ";
  cin >> unaVenta.precio;
}

void mostrar(vendido lista)
{
  for(int i = 0; i < lista.cont; i++)
  {
    cout << "Artículo " << i + 1 << ":" << endl;
    cout << "  Código del artículo: "
         << lista.items[i].articulo << endl;
    cout << "  Unidades: "
         << lista.items[i].unidades << endl;
```

(*continúa*)

## Un ejemplo de arrays de estructuras

```
      cout << "  Precio del artículo: "
           << lista.items[i].precio << endl;
    }
  }

  double total(vendido lista)
  {
    double tot = 0;

    for(int i = 0; i < lista.cont; i++)
      tot += lista.items[i].unidades *
             lista.items[i].precio;
    return tot;
  }
```

(*continúa*)



## Un ejemplo de arrays de estructuras

```
int main()
{
  vendido lista;
  venta unaVenta;
  lista.cont = 0;
  for(int i = 0; i < 4; i++)
  {
    leer(unaVenta);
    lista.items[lista.cont] = unaVenta;
    lista.cont++;
  }
  mostrar(lista);
  cout << endl << "Total de las ventas: "
       << total(lista) << endl;
  return 0;
}
```



## Arrays como atributos de objetos: una clase `PilaInt`

```
// Clase PilaInt - Archivo de cabecera "pilaint.h"
#ifndef pilaint_h // Evitar inclusiones múltiples
#define pilaint_h

class PilaInt { // Pila de enteros
public:
  PilaInt();
  PilaInt(const PilaInt&);
  PilaInt& operator=(const PilaInt&);
  ~PilaInt();
  bool pilaVacia() const;
  bool pilaLlena() const;
  bool push(int); // false si la pila está llena
  bool pop(int&); // false si la pila está vacía
private:
  enum { MAX = 10 };
  int _array[MAX];
  int _tope;
};

#endif
```

Una forma de definir en la clase una constante para el tamaño del array



## Arrays como atributos de objetos: una clase `PilaInt`

```
// Clase PilaInt - Implementación "pilaint.cpp"
#include <iostream>
using namespace std;
#include "pilaint.h"

PilaInt::PilaInt() : _tope(0) { }

PilaInt::PilaInt(const PilaInt& otra)
{
  _tope = otra._tope;
  for(int i = 0; i < MAX; i++)
    _array[i] = otra._array[i];
}

PilaInt& PilaInt::operator=(const PilaInt& otra)
{
  _tope = otra._tope;
  for(int i = 0; i < MAX; i++)
    _array[i] = otra._array[i];
  return *this;
}
```

(*continúa*)

## Arrays como atributos de objetos: una clase **PilaInt**

```
PilaInt::~PilaInt() { }

bool PilaInt::pilaVacia() const { return _tope == 0; }

bool PilaInt::pilaLlena() const { return _tope == MAX; }

bool PilaInt::push(int dato)
{
  if(_tope == MAX) return false;
  _array[_tope] = dato;
  _tope++;
  return true;
}

bool PilaInt::pop(int& dato)
{
  if(_tope == 0) return false;
  _tope--;
  dato = _array[_tope];
  return true;
}
```



## Arrays como atributos de objetos: una clase **PilaInt**

```
// Un programa de prueba de la clase PilaInt - "pruebapila.cpp"

#include <iostream>
using namespace std;
#include "pilaint.h"

int main()
{
  PilaInt unaPila;
  int op, valor;

  do {
    cout << "1. Apilar ..." << endl;
    cout << "2. Desapilar ..." << endl;
    cout << "0. Salir" << endl;
    cout << "     Opción: ";
    cin >> op;
    if(op == 1) {
      cout << "Valor a apilar: ";
      cin >> valor;
```



## Arrays como atributos de objetos: una clase **PilaInt**

```
      if(unaPila.push(valor))
        cout << "Apilado correctamente." << endl;
      else
        cout << "Lo siento. La pila esta llena." << endl;
    }
    if(op == 2)
      if(unaPila.pop(valor))
        cout << "Valor desapilado: " << valor << endl;
      else
        cout << "Lo siento. La pila esta vacía." << endl;
  } while(op != 0);

  while(!unaPila.pilaVacia()) {
    unaPila.pop(valor);
    cout << valor << " ";
  }
  cout << endl;

  return 0;
}
```



## Cadenas de caracteres: la clase **string**

Mejor alternativa que las cadenas de caracteres al estilo de C (se tratan en el suplemento de esta unidad).

- ✓ La clase asume la responsabilidad de la gestión de memoria.
- ✓ Tiene definidos operadores sobrecargados (por ejemplo, el + para concatenar).
- ✓ Cadenas más eficientes y seguras de usar.

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

int main()
{
  string cad1("Hola");   // inicialización
  string cad2 = "amigo"; // inicialización
  string cad3;
```

(*continúa*)

## Cadenas de caracteres: la clase `string`

```
    cad3 = cad1; // copia
    cout << "cad3 = " << cad3 << endl;

    cad3 = cad1 + " "; // concatenación
    cad3 += cad2;      // concatenación
    cout << "cad3 = " << cad3 << endl;

    cad1.swap(cad2);  // intercambio
    cout << "cad1 = " << cad1 << endl;
    cout << "cad2 = " << cad2 << endl;

    return 0;
}
```



## Entrada/salida con objetos de la clase `string`

La función **`getline()`** permite leer cadenas con espacios en blanco.

**`cin.ignore()`** sirve para que se ignore cualquier entrada pendiente (puede hacer falta entre dos lecturas de cadenas).

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

int main()
{
  string nombre, apellidos;
  cout << "Introduzca un nombre: ";
  cin >> nombre; // la entrada termina al encontrar el
           // primer espacio en blanco o pulsar Intro
  cout << "Introduzca los apellidos: ";
  getline(cin, apellidos); // la entrada termina al
      // pulsar Intro (se leen los espacios en blanco)
  cout << "Nombre completo: " << nombre << " "
       << apellidos << endl;

  return 0;
}
```

>>> Prog0404



## Otras operaciones sobre objetos de la clase `string`

### Comparación (alfanumérica) de cadenas:

Los operadores relacionales están definidos como métodos operadores en la clase **`string`**.

### Acceso a los caracteres de una cadena:

- ✓ **`[]`**: sin control de intentos de acceso a posiciones inexistentes.
- ✓ **`at(`***posición***`)`**: provoca una excepción si se intenta acceder a una *posición* inexistente (más adelante hablaremos de excepciones).

### Longitud de una cadena:

Están definidos los métodos **`size()`** y **`length()`**.
Cualquiera de los dos devuelve la longitud de la cadena.



## Una clase con atributos de clase `string`

```
// Clase Persona - Archivo de cabecera "persona.h"

#ifndef persona_h // Evitar inclusiones múltiples
#define persona_h

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

class Persona {
public:
  Persona(string = "", int = 0, string = "", string = "");
  // Constructor. Datos: NIF, edad, nombre y apellidos
  Persona(const Persona&); // Constructor de copia
  Persona& operator=(const Persona&); // Copia (asignación)
  ~Persona(); // Destructor
  // Mutadores:
  void nif(string);
  void edad(int);
  void nombre(string);
  void apellidos(string);
```

(*continúa*)

## Una clase con atributos de clase `string`

```
  // Accedentes:
  string nif() const;
  int edad() const;
  string nombre() const;
  string apellidos() const;
  // Visualizador:
  void mostrar() const;
  // Resto de métodos:
  string nombreCompleto() const; // Nombre y apellidos
  void felizCumple(); // El día del cumpleaños
  void leer(); // Lectura de los datos de la persona
private:
  // Atributos:
  string _nif;
  int _edad;
  string _nombre, _apellidos;
};

#endif
```





## Una clase con atributos de clase `string`

```
// Clase Persona - Implementación "persona.cpp"
#include "persona.h"

Persona::Persona(string nif, int edad, string nombre,
                 string apellidos)
  : _nif(nif), _edad(edad), _nombre(nombre),
   _apellidos(apellidos) { }

Persona::Persona(const Persona& otra) {
  _nif = otra._nif;
  _edad = otra._edad;
  _nombre = otra._nombre;
  _apellidos = otra._apellidos;
}

Persona& Persona::operator=(const Persona& otra) {
  _nif = otra._nif;
  _edad = otra._edad;
  _nombre = otra._nombre;
  _apellidos = otra._apellidos;
  return *this;
}
```

(*continúa*)





## Una clase con atributos de clase `string`

```
Persona::~Persona() { }

void Persona::nif(string cad) { _nif = cad; }

void Persona::edad(int num) { _edad = num; }

void Persona::nombre(string cad) { _nombre = cad; }

void Persona::apellidos(string cad) { _apellidos = cad; }

string Persona::nif() const { return _nif; }

int Persona::edad() const { return _edad; }

string Persona::nombre() const { return _nombre; }

string Persona::apellidos() const { return _apellidos; }

void Persona::mostrar() const {
  cout << _nif << endl;
  cout << nombreCompleto() << endl;
  cout << "Edad: " << _edad << endl;
}
```

(*continúa*)





## Una clase con atributos de clase `string`

```
string Persona::nombreCompleto() const
{ return _nombre + " " + _apellidos; }

void Persona::felizCumple() {
  _edad++;
  cout << "Registro actualizado: " << endl;
  mostrar();
}

void Persona::leer()
{
  cout << "NIF: ";
  cin >> _nif;
  cout << "Nombre: ";
  getline(cin, _nombre);
  cout << "Apellidos: ";
  getline(cin, _apellidos);
  cout << "Edad: ";
  cin >> _edad;
}
```