FACULTAD DE INFORMÁTICA

Arrays y cadenas de caracteres

TEMA

Programación orientada a objetos — Unidad 4

Autor: Luis Hernández Yáñez

FdI UCM

Contenido

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]
```

```
#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;
}
```

!

No se comprueban los intentos de acceso a posiciones inexistentes de los arrays

FdI UCM

Un ejemplo de uso de arrays unidimensionales

Programa que calcula el producto escalar de dos vectores

```
#include <iostream>
using namespace std;
int main()
{
  declaraciones

  entrada de datos

  cálculos

  salida de datos

  return 0;
}
```

```
int v1[3], v2[3], i;
int prod;
```

Un ejemplo de uso de arrays unidimensionales

Programa que calcula el producto escalar de dos vectores

```
#include <iostream>
using namespace std;
int main()
{
  declaraciones

  entrada de datos

  cálculos

  salida de datos

  return 0;
}
```

```
cout << "Vector 1:" << endl;
for(i = 0; i < 3; i++) {
  cout << "Componente " << i+1
       << ": ";
  cin >> v1[i];
}
cout << "Vector 2:" << endl;
for(i = 0; i < 3; i++) {
  cout << "Componente " << i+1
       << ": ";
  cin >> v2[i];
}
```


Un ejemplo de uso de arrays unidimensionales

Programa que calcula el producto escalar de dos vectores

```
#include <iostream>
using namespace std;
int main()
{
  declaraciones

  entrada de datos

  cálculos

  salida de datos

  return 0;
}
```

```
prod = 0;
for(i = 0; i < 3; i++)
  prod += v1[i] * v2[i];
```


Un ejemplo de uso de arrays unidimensionales

Programa que calcula el producto escalar de dos vectores

```
#include <iostream>
using namespace std;
int main()
{
  declaraciones

  entrada de datos

  cálculos

  salida de datos

  return 0;
}
```

```
cout << "Producto: " << prod
     << endl;
```


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;
```

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

Arrays no delimitados

Un array que se inicializa se puede declarar como array no delimitado (en tamaño):

- ✓ Se deja sin especificar la dimensión de más a la izquierda.
- ✓ El compilador establecerá automáticamente el tamaño de esa dimensión de acuerdo con el número de inicializadores.

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

Como hay diez inicializadores, el compilador establece la dimensión del array en 10.

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

Como hay diez inicializadores y la segunda dimensión es 2, el compilador establece la primera dimensión del array en 5.

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);
```

Determinación del tamaño del array en la función

El tamaño de un array se suele definir por medio de una constante:

```
const int longitud = 10;

double miArray[longitud];
```

Cuando se pasa el array a una función es importante que ésta conozca el tamaño del array, con el fin de no sobrepasar su límite.

Si la constante es un dato global, aunque acceder al dato global dentro de la función no conlleva grandes riesgos por tratarse de un dato constante, lo mejor es proporcionar la dimensión del array como un argumento más de la función:

```
void f(double [], const int); // prototipo

...

void f(double miArray[], const int longitud)

{ ... }
```

Este método servirá además para aquellos casos en los que no se tenga acceso directo a la constante que define el tamaño del array.

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;
}
```

>>> Prog0402

Arrays como atributos de objetos: una clase **Pila**

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

class Pila { // Pila de enteros
public:
  Pila() : _tope(0) {}
  Pila(const Pila&);
  Pila& operator=(const Pila&);
  ~Pila() {}
  bool pilaVacia() const;
  bool pilaLlena() const;
  void push(int);
  int pop();
private:
  enum { MAX = 10 };
  int _array[MAX];
  int _tope;
};
```

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

(*continúa*)

Arrays como atributos de objetos: una clase **Pila**

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

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

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

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

(*continúa*)

Arrays como atributos de objetos: una clase **Pila**

```
void Pila::push(int dato)
{
  if(_tope < MAX) {
    _array[_tope] = dato;
    _tope++;
  }
}

int Pila::pop()
{
  if(_tope == 0) {
    cout << "No hay elementos en la pila" << endl;
    return 0;
  }
  else {
    _tope--;
    return _array[_tope];
  }
}
```

MAL DISEÑO: En lugar de mostrar un mensaje en la pantalla cuando falla, es mejor que la función indique su éxito/fallo con un resultado de tipo **bool**. [Método mejor implementado en el resumen.]

(*continúa*)

Arrays como atributos de objetos: una clase **Pila**

```
int main()
{
  Pila unaPila;
  int i;

  do {
    cout << "Un entero (0 para terminar): ";
    cin >> i;
    if(i != 0) unaPila.push(i);
   } while(i != 0);

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

  return 0;
}
```

>>> Prog0403

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.

Cadenas de caracteres: la clase **string**

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

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

Una clase con atributos de clase `string`

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

class Persona {
public:
```

Forma canónica ortodoxa

```
 // Constructor (predeterminado):
 Persona(string nif = "", int edad = 0,
         string nombre = "", string apellidos = "") :
         _nif(nif), _edad(edad), _nombre(nombre),
         _apellidos(apellidos) { }
 // Constructor de copia:
 Persona(const Persona&);
 // Copia (operador de asignación):
 Persona& operator=(const Persona&);
 // Destructor:
 ~Persona() {}
 // Mutadores:
 void nif(string cad) { _nif = cad; }
 void edad(int num) { _edad = num; }
```

(*continúa*)

Una clase con atributos de clase `string`

```
  void nombre(string cad) { _nombre = cad; }
  void apellidos(string cad) { _apellidos = cad; }
  // Accedentes:
  string nif() const { return _nif; }
  int edad() const { return _edad; }
  string nombre() const { return _nombre; }
  string apellidos() const { return _apellidos; }
  // 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;
}; // Fin de la definición de la clase
```

(*continúa*)

Una clase con atributos de clase `string`

```
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;
}

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;
}
```

Lectura de cadenas (`string`)

Recuérdese que para leer cadenas con espacios en blanco se debe usar la función **`getline()`**.

Y si experimentamos problemas con la lectura (**`cin >> ...`**) consecutiva de varias cadenas, a menudo éstos se solucionan pasando el mensaje **`ignore()`** a **`cin`** entre medias de algunas de las operaciones de lectura.

Téngase en cuenta que con algunos compiladores hay que pulsar Intro dos veces en ocasiones para que se proceda con la siguiente operación de lectura.

Métodos de búsqueda de la clase `string`

La clase **`string`** tiene definidos algunos métodos que permiten buscar caracteres y subcadenas en los objetos de la clase.

`find(`*subcadena***`)`**: devuelve la posición donde empieza la primera ocurrencia de la *subcadena* en la cadena receptora (el primer carácter está en la posición 0).

```
string cad; int n;
...
n = cad.find("abc");
```

`find_first_of(`*cadena***`)`**: devuelve la posición donde se encuentra, en la cadena receptora, la primera ocurrencia de cualquier carácter que contenga la *cadena* proporcionada.

```
string cad; int n;
...
n = cad.find_first_of("aeiou");
```


Métodos de búsqueda de la clase `string`

`find_first_not_of(`*cadena***`)`**: devuelve la posición donde se encuentra, en la cadena receptora, la primera ocurrencia de cualquier carácter que no contenga la *cadena* proporcionada.

`rfind(`*subcadena***`)`**: devuelve la posición donde empieza la última ocurrencia de la *subcadena* en la cadena receptora (busca desde el final hacia el principio).

`find_last_of(`*cadena***`)`**: devuelve la posición donde se encuentra, en la cadena receptora, la última ocurrencia de cualquier carácter que contenga la *cadena* proporcionada.

`find_last_not_of(`*cadena***`)`**: devuelve la posición donde se encuentra, en la cadena receptora, la última ocurrencia de cualquier carácter que no contenga la *cadena* proporcionada.

Modificación de objetos de la clase `string`

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

int main()
{
  string cad1, cad2, cad3;
  ...
  cad1.erase(0,7); // elimina 7 caracteres desde el 1º
  cad1.replace(9,5,cad2); // reemplaza 5 caracteres
              // a partir de la posición 9 por cad2
  cad1.insert(0,cad3); // inserta en la posición 0 cad3
  cad1.append(3,'!'); // añade por el final 3 caracteres
                      // de signo de exclamación
  ...
```


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.