

Metodología de la Programación

Tema 1. Arrays, cadenas estilo C y matrices

Departamento de Ciencias de la Computación e I.A.



DECSAI
UNIVERSIDAD DE GRANADA



ETSIIT Universidad de Granada

Curso 2012-13

Parte I: Arrays

- 1 Definición
- 2 Declaración y representación en memoria
- 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
- 4 Sobre el tamaño de los arrays
- 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
- 6 Diferencias entre arrays y clase vector
- 7 Cadenas de caracteres estilo C

Parte II: Matrices

- 8 Declaración e inicialización de matrices de 2 dimensiones
- 9 Operaciones con matrices
 - Acceso, asignación, lectura y escritura
- 10 Sobre el tamaño de las matrices
- 11 Matrices de más de 2 dimensiones
- 12 Funciones y matrices
- 13 Gestión de filas de una matriz como arrays

Motivación

Pensar en un programa para calcular la varianza de 500 datos introducidos por el usuario.

Problema

Número de variables imposible de sostener y recordar (nota1, nota2, ..., nota500)

Solución

Introducir un tipo de dato nuevo que permita representar todos esos valores en una única **estructura de datos**, reconocible bajo un nombre único.

notas[0]	notas[1]	...	notas[499]
2.4	4.9	...	6.7

Objetivos

Objetivos

- Conocer los mecanismos disponibles en C++ para almacenar colecciones de valores de forma secuencial.
- Ser capaces de valorar el mecanismo más adecuado para un determinado problema.
- Conocer los conceptos básicos del uso de cadenas de caracteres estilo C (por compatibilidad entre C y C++) .
- Comprender las matrices y saber realizar operaciones sobre ellas.

Importante: comprensión de los ejemplos de código y realización de los ejercicios que se vayan indicando.

Parte I

Arrays

Contenido del tema

1 Definición

- 2 Declaración y representación en memoria
- 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
- 4 Sobre el tamaño de los arrays
- 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
- 6 Diferencias entre arrays y clase vector
- 7 Cadenas de caracteres estilo C

- 8 Declaración e inicialización de matrices de 2 dimensiones
- 9 Operaciones con matrices
 - Acceso, asignación, lectura y escritura
- 10 Sobre el tamaño de las matrices
- 11 Matrices de más de 2 dimensiones
- 12 Funciones y matrices
- 13 Gestión de filas de una matriz como arrays

Definición

Tipo de dato compuesto

Una composición de tipos de datos simples (o incluso compuestos) caracterizado por la **organización** de sus datos y por las **operaciones** que se definen sobre él.

Array

Un **tipo de dato compuesto** de un número fijo de componentes **del mismo tipo** y donde cada una de ellos es **directamente accesible** mediante un **índice**.

Contenido del tema

- 1 Definición
- 2 Declaración y representación en memoria
- 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
- 4 Sobre el tamaño de los arrays
- 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
- 6 Diferencias entre arrays y clase vector
- 7 Cadenas de caracteres estilo C

Declaración y representación en memoria

Consejo

Usar constantes para especificar el tamaño de los arrays.

Ventaja: es más fácil adaptar el código ante cambios de tamaño.

```

1 int main(){
2     const int NUM_REACTORES=20;
3     const int TOTAL_ALUMNOS = 100;
4     int longitud=50;
5
6     double notas[TOTAL_ALUMNOS];
7     int TemperaturasReactores[NUM_REACTORES];
8     // Otros ejemplos
9     bool casados[40];
10    char NIF[9];
11    bool Vector[longitud]; // Error en compilación.
12    .....

```

Declaración y representación en memoria

`<tipo> <identificador> [<N.Componentes>];
double notas[3];`

- `<tipo>` indica el tipo de dato común a todas las componentes del array (`double`).
 - `<N.Componentes>` determina el número de componentes del array (`3`).
 - El número de componentes debe conocerse a priori y no es posible alterarlo durante la ejecución del programa.
 - Pueden usarse literales o constantes enteras pero **nunca una variable**¹.
 - Las posiciones ocupadas por el array están contiguas en memoria.
- `double notas[3];`

`notas = [? | ? | ?]`

¹El estándar C99 permite usar una variable pero **C++ estándar no lo admite**. g++ lo admite como extensión propia.

Contenido del tema

- 1 Definición
- 2 Declaración y representación en memoria
- 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
- 4 Sobre el tamaño de los arrays
- 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
- 6 Diferencias entre arrays y clase vector
- 7 Cadenas de caracteres estilo C
- 8 Declaración e inicialización de matrices de 2 dimensiones
- 9 Operaciones con matrices
 - Acceso, asignación, lectura y escritura
- 10 Sobre el tamaño de las matrices
- 11 Matrices de más de 2 dimensiones
- 12 Funciones y matrices
- 13 Gestión de filas de una matriz como arrays

Acceso a las componentes

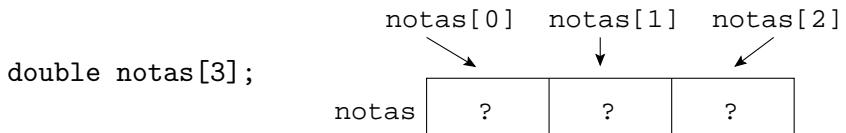
Dada la declaración

`<tipo> <identificador> [<N.Componentes>];`

cada componente se accede de la forma:

`<identificador> [<índice>]
notas[2]`

- El índice de la primera componente del array es 0.
- El índice de la última componente es `<N.Componentes>-1`.



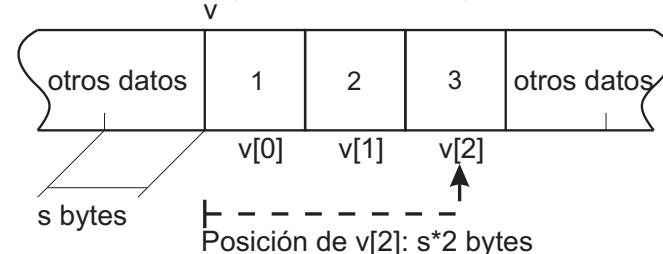
`notas[9]`, `notas['1']` o `notas[1.5]` no son correctas.

- Cada componente es una variable más del programa, del tipo indicado en la declaración del array.

Acceso a las componentes

- Para acceder a la componente i , el compilador se debe desplazar i posiciones desde el comienzo del array.
- Cada posición tiene un número de bytes determinado por el tipo de dato base.

Para acceder a `v[2]` el compilador saltará 2 posiciones desde el comienzo de `v`. Si cada posición tiene s bytes, saltará $s*2$ bytes.



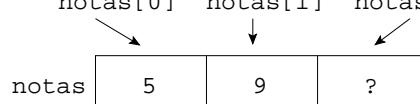
Inicialización

`<tipo> <identificador> [<N.Componentes>] = {<valores separados por coma>};`

```
int arrayInt[3]={4,5,6};
initializa arrayInt[0]=4, arrayInt[1]=5, arrayInt[2]= 6
int arrayInt[7]={3,5};
initializa arrayInt[0]=3, arrayInt[1]= 5 y el resto se inicializan a cero.
int arrayInt[7]={0};
initializa todas las componentes a cero.
int arrayInt []={1,3,9};
automáticamente el compilador asume int arrayInt[3]
```

Asignación

```
<identificador> [<índice>] = <expresión>;
1 int main(){
2     double notas[3];
3
4     notas[0] = 5;
5     notas[1] = 9;
6 }
```



¡Cuidado!

No se permite la asignación global a todos los elementos del array.
Las asignaciones se deben realizar componente a componente *salvo en el momento de la definición de la variable, la inicialización*.

```
1 int main(){
2     double notas[3]={5,6,8};
3     double notas2[3];
4
5     notas2=notas; // Error de compilación
6 }
```

Asignación

¡Cuidado!

El compilador no comprueba el acceso a las componentes: modificar una componente inexistente tiene consecuencias imprevisibles.

```

1 int main(){
2     double notas[3];
3
4     notas[0] = 5;
5     notas[1] = 9;
6     notas[3] = 7; // Probable Error ejecución
7     notas[-2] = 6; // Probable Error ejecución
8     notas[15] = 5; // Probable Error ejecución
9 }
```

Lectura y escritura

La lectura y escritura se realiza componente a componente.

```

1 int main(){
2     const int NUM_NOTAS = 5;
3     double notas[NUM_NOTAS], media;
4
5     for (int i=0; i<NUM_NOTAS; i++){
6         cout << "Nota del alumno " << i << ": ";
7         cin >> notas[i];
8     }
9
10    media = 0;
11    for (int i=0; i<NUM_NOTAS; i++)
12        media += notas[i];
13    media /= NUM_NOTAS;
14
15    cout << "\nMedia = " << media << endl;
16 }
```



Sobre el tamaño de los arrays

Contenido del tema

- 1 Definición
- 2 Declaración y representación en memoria
- 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
- 4 Sobre el tamaño de los arrays
- 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
- 6 Diferencias entre arrays y clase vector
- 7 Cadenas de caracteres estilo C
- 8 Declaración e inicialización de matrices de 2 dimensiones
- 9 Operaciones con matrices
 - Acceso, asignación, lectura y escritura
- 10 Sobre el tamaño de las matrices
- 11 Matrices de más de 2 dimensiones
- 12 Funciones y matrices
- 13 Gestión de filas de una matriz como arrays

Sobre el tamaño de los arrays

En C++ no es posible declarar un array de tamaño variable. → Debemos dar un tamaño *suficientemente grande* a los arrays.

Pero, ¿cómo sabemos el número de elementos que realmente estamos usando en un momento dado?

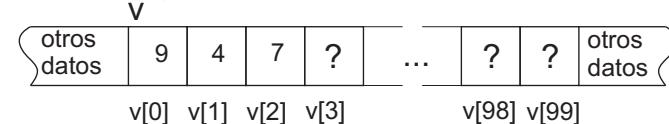
Normalmente se dejan libres los elementos con índice más alto.

Habitualmente se usa una variable entera que indique el número de componentes usadas.

Como convención, usaremos identificadores del tipo `util`, `utilNombreDelArray` o `util_nombre_del_array`.

```
int v[100] = {9, 4, 7};
```

```
int util_v = 3;
```



Sobre el tamaño de los arrays

Ejemplo de código en que se pide al usuario que indique el número de alumnos cuyas notas se van a procesar. Este valor se guarda en `util_notas`

```

1 int main(){
2     const int DIM_NOTAS = 100;
3     double notas[DIM_NOTAS];
4     int util_notas;
5     double media;
6     do{
7         cout << "Introduzca el número de alumnos (entre 1 y "
8             << DIM_NOTAS << "): ";
9         cin >> util_notas;
10    }while (util_notas<1 || util_notas>DIM_NOTAS);
11    for (int i=0; i<util_notas; i++){
12        cout << "nota[" << i << "]: ";
13        cin >> notas[i];
14    }
15    media=0.0;
16    for (int i=0; i<util_notas; i++)
17        media += notas[i];
18    media /= util_notas;
19    cout << "\nMedia: " << media << endl;
20 }
```



Sobre el tamaño de los arrays

Un método alternativo es usar un elemento *especial* que indique el final del array (`elemento centinela`).

```

1 int main(){
2     const int DIM_NOTAS = 100;
3     double notas[DIM_NOTAS];
4     double media;
5     int i;
6
7     cout << "nota[0]: (-1 para terminar): ";
8     cin >> notas[0];
9     for(i=1; notas[i-1] != -1 && i < DIM_NOTAS-1; i++){
10        cout << "nota[" << i << "]: (-1 para terminar): ";
11        cin >> notas[i];
12    }
13    if (i==DIM_NOTAS-1)
14        notas[i] = -1;
15
16    media=0;
17    for (i=0; notas[i] != -1; i++)
18        media += notas[i];
19
20    if (i == 0)
21        cout << "No se introdujo ninguna nota\n";
22    else{
23        media /= i;
24        cout << "\nMedia: " << media << endl;
25    }
26 }
```



Contenido del tema

- 1 Definición
- 2 Declaración y representación en memoria
- 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
- 4 Sobre el tamaño de los arrays
- 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
- 6 Diferencias entre arrays y clase vector
- 7 Cadenas de caracteres estilo C

- 8 Declaración e inicialización de matrices de 2 dimensiones
- 9 Operaciones con matrices
 - Acceso, asignación, lectura y escritura
- 10 Sobre el tamaño de las matrices
- 11 Matrices de más de 2 dimensiones
- 12 Funciones y matrices
- 13 Gestión de filas de una matriz como arrays



Funciones y arrays

El paso de arrays a funciones se hace mediante un parámetro formal del mismo tipo **exactamente** (no basta con que sea compatible).

```

1 #include <iostream>
2 using namespace std;
3
4 void imprime_array (char v[5]){
5     for (int i=0; i<5; i++)
6         cout << v[i] << " ";
7 }
8 int main(){
9     char vocales[5]={'a','e','i','o','u'};
10    imprime_array(vocales);
11 }
```

Si necesitamos usar el mismo algoritmo para diferentes tipos de dato tendremos que implementar una función para cada tipo.

Funciones y arrays

C++ permite usar un array sin dimensiones como parámetro formal.

Necesitamos saber el número de componentes usadas.

```

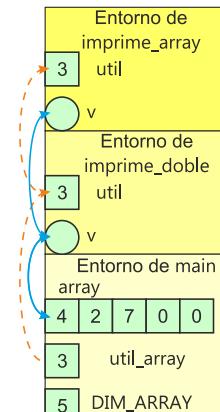
1 #include <iostream>
2 using namespace std;
3
4 void imprime_array(char v[], int util){
5     for (int i=0; i<util; i++)
6         cout << v[i] << " ";
7 }
8 int main(){
9     char vocales[5]={'a','e','i','o','u'};
10    char digitos[10]={'0','1','2','3','4',
11                      '5','6','7','8','9'};
12    imprime_array(vocales, 5); cout<<endl;
13    imprime_array(digitos, 10); cout<<endl;
14    imprime_array(digitos, 5); cout<<endl; // del '0' al '4'
15    imprime_array(vocales, 100); cout<<endl; // ERROR al ejecutar, no al
16                                // compilar
17 }
```



Los arrays es como si se pasasen por referencia, en el sentido de que podemos modificar las componentes pero **no se pone &**.

```

1 #include <iostream>
2 using namespace std;
3
4 void imprime_array(int v[], int util){
5     for (int i=0; i<util; i++)
6         cout << v[i] << " ";
7 }
8 void imprime_doble(int v[], int util){
9     for (int i=0; i<util; i++)
10        v[i] *= 2;
11    imprime_array(v, util);
12 }
13 int main(){
14     const int DIM_ARRAY = 5;
15     int array[DIM_ARRAY]={4,2,7};
16     int util_array=3;
17     imprime_doble(array, util_array);
18     imprime_array(array, util_array);
19 }
```



Funciones y arrays

Problema

¿Cómo evitamos que se puedan modificar los elementos contenidos en el array?

Solución: arrays de constantes

Utilizando el calificador `const`.

```

1 void imprime_array(const int v[], int util){
2     for (int i=0; i<util; i++)
3         cout << v[i] << " ";
4 }
5 void imprimedoble(const int v[], int util){
6     for (int i=0; i<util; i++)
7         v[i] *= 2; // ERROR de compilación
8     imprime_array(v, util);
9 }
```

Paso de argumentos: array

Atención al error de compilación:

```
imprimedoble.cpp: En la función 'void imprime_array(const int*, int)':
imprimedoble.cpp:3:7: error: 'cout' no se declaró en este ámbito
imprimedoble.cpp: En la función 'void imprimedoble(const int*, int)':
imprimedoble.cpp:7:15: error: asignación de la ubicación de sólo lectura
      `*(v + ((size_type)((long unsigned int)i * 4ul)))'
```



Funciones y arrays

- Si no se utiliza el calificador `const`, el compilador asume que el array se va a modificar (aunque no se haga).
- No es posible pasar un array de constantes a una función cuya cabecera indica que el array se modifica (aunque la función realmente no modifique el array)

```

1 #include<iostream>
2 using namespace std;
3
4 void imprime_array (char v[]){
5     for (int i=0; i<5; i++)
6         cout << v[i] << " ";
7 }
8 int main(){
9     const char vocales[5]={‘a’, ‘e’, ‘i’, ‘o’, ‘u’};
10    imprime_array(vocales); //ERROR de compilación
11 }
```

Paso de argumentos: array

Atención al error de compilación:

```
imprimevocalesconst.cpp: En la función 'int main()':
imprimevocalesconst.cpp:10:25: error: conversión inválida de
      'const char*' a 'char*' [-fpermissive]
imprimevocalesconst.cpp:4:6: error:   argumento de inicialización 1 de
      'void imprime_array(char*)' [-fpermissive]
```



Devolución de arrays en funciones I

Si queremos que una función **devuelva** un array, éste no puede ser local ya que al terminar la función, su zona de memoria *desaparecería*.

Debemos declarar dicho array en el `main` (en general, en la función llamante) y pasarlo como parámetro.

```

1 #include <iostream>
2 using namespace std;
3
4 void imprime_array(const int v[], int util);
5 void solo_pares(const int v[], int util_v,
6                  int pares[], int &util_pares);
7 int main(){
8     const int DIM=100;
9     int entrada[DIM] = {8,1,3,2,4,3,8},
10        salida[DIM];
11    int util_entrada = 7, util_salida;
12    solo_pares(entrada, util_entrada, salida, util_salida);
13    imprime_array(salida, util_salida);
14 }
```



Devolución de arrays en funciones II

```

16 void solo_pares(const int v[], int util_v,
17                   int pares[], int &util_pares){
18     util_pares=0;
19     for (int i=0; i<util_v; i++)
20         if (v[i]%2 == 0){
21             pares[util_pares]=v[i];
22             util_pares++;
23         }
24 }
25 void imprime_array(const int v[], int util){
26     for (int i=0; i<util; i++)
27         cout << v[i] << " ";
28 }
```

Ejemplo

Quitar los elementos consecutivos repetidos de un array, guardando el resultado en otro array.

```

1 #include <iostream>
2 using namespace std;
3
4 void imprime_array(const char v[], int util);
5 void quita_repes(const char original[], int util_original,
6                  char destino[], int &util_destino);
7
8 int main(){
9     const int DIM =100;
10    char entrada[DIM]={'b','b','i','e','n','n','n'},
11        salida[DIM];
12    int util_entrada = 7, util_salida;
13    quita_repes(entrada, util_entrada, salida, util_salida);
14    imprime_array(salida, util_salida);
15 }
16
17

```



```

18 void quita_repes(const char original[], int util_original,
19                     char destino[], int &util_destino){
20     destino[0] = original[0];
21     util_destino = 1;
22     for (int i=1; i<util_original; i++){
23         if (original[i] != original[i-1]){
24             destino[util_destino] = original[i];
25             util_destino++;
26         }
27     }
28
29 void imprime_array(const char v[], int util){
30     for (int i=0; i<util; i++)
31         cout << v[i] << " ";
32 }

```

Trabajando con arrays locales

Comprobar si un array de dígitos (0 a 9) de int es capicua

Algoritmo:

- ① Eliminar elementos que no estén entre 0 y 9.
- ② Recorrer el array desde el principio hasta la mitad
 - ① Comprobar que el elemento en la posición actual desde el inicio, es igual al elemento en la posición actual desde el final.

Necesitamos un array local donde guardar el resultado del paso 1. ¿Cómo lo declaramos?

Lo ideal sería poder crear un array con el tamaño justo: el número de dígitos. Pero no sabemos cuántos habrá....

Trabajando con arrays locales

Así que habrá que usar una constante global

```
const int DIM = 100;
```

```

bool capicua(const int v[], int longitud){
    int solodigitos[DIM];
    .....
}
int main(){
    int entrada[DIM];

```

Es la única solución (de momento).

Inconveniente: no podemos separar las implementación de `capicua` de la definición de la constante.

Solución: Memoria dinámica o clase vector.

```

1 #include <iostream>
2 using namespace std;
3 const int DIM = 100;
4
5 void quita_nodigitos(const int original[],
6     int util_original,int destino[], int &util_destino);
7 void imprimevector(const int v[], int util);
8 bool capicua(const int v[], int longitud);
9 int main(){
10     int entrada1[DIM]={1,2,3,4,3,2,1};
11     int util_entrada1=7;
12     int entrada2[DIM]={1,2,3,4,5,6,10, 7,8,9,10, 11, 9,12,
8, 13, 7, 6, -1, 5, 4, 3, 2, 1};
13     int util_entrada2=24;
14
15     imprimevector(entrada1, util_entrada1);
16     if (capicua(entrada1, util_entrada1))
17         cout << " es capicua\n";
18     else
19         cout << " no es capicua\n";
20
21     imprimevector(entrada2, util_entrada2);
22

```



```

23     if (capicua(entrada2, util_entrada2))
24         cout << " es capicua\n";
25     else
26         cout << " no es capicua\n";
27 }
28
29 void quita_nodigitos(const int original[],
30                         int util_original,
31                         int destino[], int &util_destino){
32     util_destino=0;
33     for (int i=0; i<util_original; i++)
34         if (original[i] > -1 && original[i] < 10){
35             destino[util_destino]=original[i];
36             util_destino++;
37 }
38
39
40
41
42
43
44
45

```

```

46 bool capicua(const int v[], int longitud){
47     bool escapicua = true;
48     int solodigitos[DIM];
49     int long_real;
50
51     quita_nodigitos(v, longitud, solodigitos, long_real);
52
53     for (int i=0; i< long_real/2 && escapicua; i++)
54         if(solodigitos[i] != solodigitos[long_real-1-i])
55             escapicua = false;
56
57     return escapicua;
58 }
59
60 void imprimevector(const int v[], int util){
61     for (int i=0; i<util; i++)
62         cout << v[i] << " ";
63 }

```

Diferencias entre arrays y clase vector

- ### Contenido del tema
- 1 Definición
 - 2 Declaración y representación en memoria
 - 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
 - 4 Sobre el tamaño de los arrays
 - 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
 - 6 Diferencias entre arrays y clase vector
 - 7 Cadenas de caracteres estilo C
 - 8 Declaración e inicialización de matrices de 2 dimensiones
 - 9 Operaciones con matrices
 - Acceso, asignación, lectura y escritura
 - 10 Sobre el tamaño de las matrices
 - 11 Matrices de más de 2 dimensiones
 - 12 Funciones y matrices
 - 13 Gestión de filas de una matriz como arrays

Diferencias entre array y vector

Características vector

- ① Los objetos de la clase **vector** se dimensionan de forma automática, según se necesite.
- ② Recorrer un array es más rápido que recorrer un vector: el acceso a los elementos de un vector involucra comprobación de límites. Aunque también pueden recorrerse con `[]` usando índices (así no hay comprobación de límites, lo que acelara la velocidad).

Características array

- ① Los índices del array pueden salirse del rango permitido (C++ no lo comprueba en tiempo de ejecución). Esto suele conllevar a errores de ejecución.
- ② Cuando pasamos un array a una función, necesitamos pasar también el tamaño del array.
- ③ Un array no puede asignarse a otro con el operador de asignación.

Contenido del tema

- 1 Definición
- 2 Declaración y representación en memoria
- 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
- 4 Sobre el tamaño de los arrays
- 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
- 6 Diferencias entre arrays y clase vector
- 7 Cadenas de caracteres estilo C
- 8 Declaración e inicialización de matrices de 2 dimensiones
- 9 Operaciones con matrices
 - Acceso, asignación, lectura y escritura
- 10 Sobre el tamaño de las matrices
- 11 Matrices de más de 2 dimensiones
- 12 Funciones y matrices
- 13 Gestión de filas de una matriz como arrays

Literales de cadena de caracteres

- Un literal de cadena de caracteres es una secuencia de cero o más caracteres encerrados entre comillas dobles.
- Su longitud es el número de caracteres que tiene.
- Su tipo es un array de **char** con un tamaño igual a su longitud más uno (para el carácter nulo).

"Hola" de tipo **const char[5]**

"Hola mundo" de tipo **const char[11]**

"" de tipo **const char[1]**

Cadenas de caracteres estilo C

Cadena de caracteres

Secuencia ordenada de caracteres de longitud variable.

Permiten trabajar con datos como apellidos, direcciones, etc...

Tipos de cadenas de caracteres en C++

- ① **cstring**: cadena de caracteres heredado de C.
- ② **string**: cadena de caracteres propia de C++ (estudiada en FP).

Cadenas de caracteres de C

Un array de tipo **char** de un tamaño determinado acabado en un carácter especial, el carácter '**\0**' (carácter nulo), que marca el fin de la cadena (véase [uso del elemento centinela](#)).

Cadenas de caracteres: declaración e inicialización

```
char nombre[10] ={'J', 'a', 'v', 'i', 'e', 'r', '\0'};
```

'J'	'a'	'v'	'i'	'e'	'r'	'\0'	?	?	?
-----	-----	-----	-----	-----	-----	------	---	---	---

```
char nombre[] ={'J', 'a', 'v', 'i', 'e', 'r', '\0'}; // Asume char[7]
```

Equivalentes a las anteriores son:

```
char nombre[10] = "Javier";
char nombre[] = "Javier";
```

¡Cuidado!

```
char cadena[] = "Hola";           //char[5]
char cadena[] = {'H', 'o', 'l', 'a'}; // char[4]
```

Paso de cadenas a funciones II

Ejemplo

Función que concatena dos cadenas

```
1 void concatena(const char cad1[], const char cad2[],
2                 char res[]){
3     int pos=0;
4     for (int i=0;cad1[i]!='\0';i++){
5         res[pos]=cad1[i];
6         pos++;
7     }
8     for (int i=0;cad2[i]!='\0';i++){
9         res[pos]=cad2[i];
10        pos++;
11    }
12    res[pos]='\0';
13 }
```



Paso de cadenas a funciones I

El paso de cadenas corresponde al paso de un array a una función. Como la cadena termina con el carácter nulo, no es necesario especificar su tamaño.

Ejemplo

Función que nos diga la longitud de una cadena

```
1 int longitud(const char cadena[]){
2     int i=0;
3     while (cadena[i]!='\0')
4         i++;
5     return i;
6 }
```



Entrada/salida de cadenas

Para leer y escribir cadenas se pueden usar las operaciones de lectura y escritura ya conocidas.

```
1 #include<iostream>
2 using namespace std;
3
4 int main(){
5     char nombre[80];
6     cout << "Introduce tu nombre: ";
7     cin >> nombre;
8     cout << "El nombre introducido es: " << nombre;
9 }
```



Problema

`cin` salta separadores antes del dato y se detiene cuando encuentra un separador (saltos de línea, espacios en blanco y tabuladores)

Entrada/salida de cadenas

Solución: (si deseamos leer algún espacio en blanco)

```
cin.getline(<cadena>, <tamaño>);
```

Lee hasta que se encuentra un salto de línea o se alcanzó el límite de lectura.

Cuidado: al combinar el uso de `cin` y `cin.getline` hay que ser consciente dónde se dejará la lectura en cada momento.

```
1 char nombre[80],direccion[120];
2 int edad;
3 cout << "Introduce tu nombre: ";
4 cin.getline(nombre,80);
5 cout << "El nombre introducido es: " << nombre;
6 cout << "\nIntroduce tu edad: ";
7 cin >> edad;
8 cout << "La edad introducida es: " << edad;
9 cout << "\nIntroduce tu direccion: ";
10 cin.getline(direccion,120);
11 cout << "La direccion introducida es: " << direccion;
```



Conversión entre cadenas `cstring` y `string`

Podemos hacer fácilmente la conversión entre cadenas `cstring` y `string`



```
1 #include <iostream>
2 #include <string>
3 #include <cstring>
4 using namespace std;
5
6 int main(){
7     char cadena1[]="Hola";
8     string cadena2;
9     char cadena3[10];
10
11     cadena2=cadena1; // cstring-->string
12     strcpy (cadena3, cadena2.c_str()); // string-->cstring
13     cout<<"cadena2="<<cadena2<<endl;
14     cout<<"cadena3="<<cadena3<<endl;
15 }
```

Entrada/salida de cadenas

```
Introduce tu nombre: Andrés Cano Utrera
El nombre introducido es: Andrés Cano Utrera
Introduce tu edad: 20
La edad introducida es: 20
Introduce tu direccion: La direccion introducida es:
```



Problema

`cin` se detiene cuando encuentra un separador, ¡y no lee el separador! (no lo consume y hace que `getline` dé por finalizada su operación al encontrarlo)

Solución: Crear una función `lee_linea` que evite las líneas vacías

```
1 void lee_linea(char c[], int tamano){
2     do{
3         cin.getline(c, tamano);
4     } while (c[0] == '\0'); // equivale a } while(longitud(c)==0);
5 }
```

```
1 cout << "Introduce tu nombre: ";
2 lee_linea(nombre,80);
3 cout << "Introduce tu edad: ";
4 cin >> edad;
5 cout << "Introduce tu direccion: ";
6 lee_linea(direccion,120);
```



La biblioteca `cstring` I

La biblioteca `cstring` proporciona funciones de manejo de cadenas de caracteres de C.

Entre otras:

- `char * strcpy(char cadena1[], const char cadena2[])`
Copia `cadena2` en `cadena1`. Es el operador de asignación de cadenas.
- `int strlen(const char s[])`
Devuelve la longitud de la cadena `s`.
- `char * strcat(char s1[], const char s2[])`
Concatena la cadena `s2` al final de `s1` y el resultado se almacena en `s1`.

La biblioteca `cstring` II

- `int strcmp(const char s1[], const char s2[])`
Compara las cadenas s1 y s2. Si la cadena s1 es menor (lexicográficamente) que s2 devuelve un valor menor que cero, si son iguales devuelve 0 y en otro caso devuelve un valor mayor que cero.
- `consts char * strstr(const char s1[], const char s2[])`
`char * strstr(char s1[], const char s2[])`
Devuelve un puntero a la primera ocurrencia de s2 en s1, o un puntero nulo si s2 no es parte de s1.

Ejercicio: Problema 5.3 pág. 161 de A. Garrido

Implemente una función que reciba una cadena de caracteres, y la modifique para que contenga únicamente la primera palabra (considere que si tiene más de una palabra, están separadas por espacios o tabuladores).

```

1 #include <iostream>
2 #include <cstring>
3 using namespace std;
4
5 const int DIM=100;
6 void lee_linea(char c[], int tamano);
7 void deja_solo_primera_palabra(char c[]);
8 int main() {
9     char cadena[DIM];
10    cout << "Introduce una cadena: ";
11    lee_linea(cadena, DIM-1);
12    deja_solo_primera_palabra(cadena);
13    cout << "Resultado = " << cadena << endl;
14 }
15 void deja_solo_primera_palabra(char c[]) {
16     int i=0;
17     // No hay espacios en blanco al inicio
18     while (c[i] != ' ' && c[i] != '\t' && i < strlen(c))
19         i++;
20     if (i < strlen(c))
21         c[i] = '\0';
22 }
```



La biblioteca `cstring` III

```

1 #include<iostream>
2 #include<cstring>
3 using namespace std;
4 int main(){
5     const int DIM=100;
6     char c1[DIM]="Hola";
7     char c2[DIM];
8
9     strcpy(c2, "mundo");
10    strcat(c1, " ");
11    strcat(c1, c2);
12    cout << "Longitudes:" << strlen(c1) << " " << strlen(c2);
13    cout << "\nc1: " << c1 << " c2: " << c2;
14    if (strcmp(c1,"adiós mundo cruel") < 0)
15        cout << "\nCuidado con las mayúsculas\n";
16    if (strcmp(c2, "mucho") > 0)
17        cout << "\n\"mundo\" es mayor que \"mucho\"\n";
18 }
```



Ejercicio: Problema 5.6 pág. 161 de A. Garrido

Escriba una función que reciba una cadena de caracteres, una posición de inicio l y una longitud L, y que nos devuelva la subcadena que comienza en l y tiene tamaño L. Nota: Si la longitud es demasiado grande (se sale de la cadena original), se devolverá una cadena de menor tamaño.

```

1 #include <iostream>
2 #include <cstring>
3 using namespace std;
4
5 const int DIM=100;
6 void lee_linea(char c[], int tamano);
7 void recorta(const char c1[], int ini, int lon, char c2[]);
8 int main() {
9     char cadena1[DIM], cadena2[DIM];
10    int i, l;
11    cout << "Introduce una cadena: ";
12    lee_linea(cadena1, DIM-1);
13    cout << "Introduce el inicio y la longitud (enteros): ";
14    cin >> i >> l;
15    recorta(cadena1,i,l,cadena2);
16    cout << "Resultado = " << cadena2 << endl;
17 }
18 void recorta(const char c1[], int ini, int lon, char c2[]) {
19     int i=0;
20     while (i+ini < strlen(c1)//para que ini o lon no sean muy grandes
21             && i<lon) { // para contar hasta lon
22         c2[i] = c1[i+ini];
23         i++;
24     }
25     c2[i] = '\0';
26 }
```



Parte II

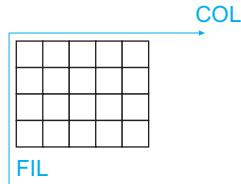
Matrices

Declaración de matrices de 2 dimensiones

```
<tipo> <identificador> [DIM_FIL][DIM_COL];
```

- El tipo base de la matriz es el mismo para todas las componentes.
- Ambas dimensiones han de ser de tipo entero

```
1 int main(){  
2     const int DIM_FIL = 2;  
3     const int DIM_COL = 3;  
4  
5     double parcela[DIM_FIL] [DIM_COL];  
6 }
```



Contenido del tema

- 1 Definición
- 2 Declaración y representación en memoria
- 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
- 4 Sobre el tamaño de los arrays
- 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
- 6 Diferencias entre arrays y clase vector
- 7 Cadenas de caracteres estilo C
- 8 Declaración e inicialización de matrices de 2 dimensiones
- 9 Operaciones con matrices
 - Acceso, asignación, lectura y escritura
- 10 Sobre el tamaño de las matrices
- 11 Matrices de más de 2 dimensiones
- 12 Funciones y matrices
- 13 Gestión de filas de una matriz como arrays

Inicialización

- “Forma segura”: Poner entre llaves los valores de cada fila.

```
int m[2] [3]={ {1,2,3}, {4,5,6} }; // m tendrá: 1 2 3  
// 4 5 6
```

- Si no hay suficientes valores para una fila determinada, los elementos restantes se inicializan a 0.

```
int mat[2] [2]={ {1}, {3,4} }; // mat tendrá: 1 0  
// 3 4
```

- Si se eliminan los corchetes que encierran cada fila, se inicializan los elementos de la primera fila y después los de la segunda, y así sucesivamente.

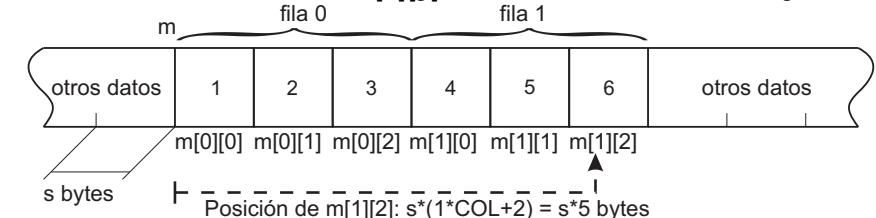
```
int A[2] [3]={1, 2, 3, 4, 5} // A tendrá: 1 2 3  
// 4 5 0
```

La declaración en detalle

- El compilador procesa las matrices como array de arrays.
- Es decir, es un array con un tipo base también array (cada fila).
- En la declaración
`int m[2][3]`
 m es un array de 2 elementos (`m[2]`) y cada elemento es un array de 3
`int (int xxxx[3])`.
- Observad que la sintaxis de la inicialización es la de un array de arrays
`int m[2][3]={{1,2,3},{4,5,6}}`;

Almacenamiento y límites de matrices

- Todos los elementos de las matrices se almacenan en un bloque contiguo de memoria.
- La organización depende del lenguaje: en C++ se almacenan por filas.
- Para acceder al elemento `m[i][j]` en una matriz `FIL × COL` el compilador debe *pasar a la fila i* y desde ahí moverse `j` elementos
- La posición del elemento `m[i][j]` se calcula como $i * \text{COL} + j$



Contenido del tema

- 1 Definición
 - 2 Declaración y representación en memoria
 - 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
 - 4 Sobre el tamaño de los arrays
 - 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
 - 6 Diferencias entre arrays y clase vector
 - 7 Cadenas de caracteres estilo C
- 8 Declaración e inicialización de matrices de 2 dimensiones
 - 9 Operaciones con matrices
 - Acceso, asignación, lectura y escritura
 - 10 Sobre el tamaño de las matrices
 - 11 Matrices de más de 2 dimensiones
 - 12 Funciones y matrices
 - 13 Gestión de filas de una matriz como arrays

Acceso, asignación, lectura y escritura

Acceso

`<identificador> [<ind1>][<ind2>]` (los índices comienzan en cero).
`<identificador> [<ind1>][<ind2>]` es una variable más del programa y se comporta como cualquier variable del tipo de dato base de la matriz.
¡El compilador no comprueba que los accesos sean correctos!

Asignación

`<identificador> [<ind1>][<ind2>] = <expresión>;`
`<expresión>` ha de ser compatible con el tipo base de la matriz.

Lectura y escritura

`cin >> <identificador> [<ind1>][<ind2>];`
`cout << <identificador> [<ind1>][<ind2>];`

Contenido del tema

- 1 Definición
- 2 Declaración y representación en memoria
- 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
- 4 Sobre el tamaño de los arrays
- 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
- 6 Diferencias entre arrays y clase vector
- 7 Cadenas de caracteres estilo C

Sobre el tamaño de las matrices II

```

18 do{
19     cout << "Introducir el número de columnas: ";
20     cin >> util_col;
21 }while ((util_col<1) || (util_col>COL));
22
23 for (f=0 ; f<util_fil; f++)
24     for (c=0 ; c<util_col ; c++){
25         cout << "Introducir el elemento (" 
26             << f << "," << c << "): ";
27         cin >> m[f][c];
28     }
29 cout << "\nIntroduzca elemento a buscar: ";
30 cin >> buscado;
31 encontrado=false;
32 for (f=0; !encontrado && (f<util_fil) ; f++)
33     for (c=0; !encontrado && (c<util_col) ; c++)
34         if (m[f][c] == buscado){
35             encontrado = true;
36             fil_enc = f; col_enc = c;
37         }

```

Sobre el tamaño de las matrices I

Para cada dimensión usaremos una variable que indique el número de componentes usadas.



```

1 #include <iostream>
2 using namespace std;
3
4 int main(){
5     const int FIL=20, COL=30;
6     double m[FIL][COL];
7     int fil_enc, col_enc, util_fil,
8         util_col, f, c;
9     double buscado;
10    bool encontrado;
11
12    do{
13        cout << "Introducir el número de filas: ";
14        cin >> util_fil;
15    }while ((util_fil<1) || (util_fil>FIL));
16
17

```

Sobre el tamaño de las matrices III

```

38 if (encontrado)
39     cout << "Encontrado en la posición "
40         << fil_enc << "," << col_enc << endl;
41 else
42     cout << "Elemento no encontrado\n";
43
44 return 0;
45 }

```

Contenido del tema

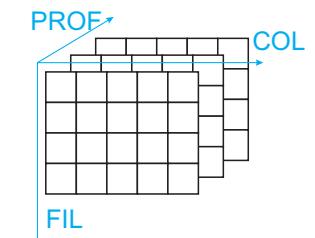
- 1 Definición
- 2 Declaración y representación en memoria
- 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
- 4 Sobre el tamaño de los arrays
- 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
- 6 Diferencias entre arrays y clase vector
- 7 Cadenas de caracteres estilo C

Contenido del tema

- 1 Definición
- 2 Declaración y representación en memoria
- 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
- 4 Sobre el tamaño de los arrays
- 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
- 6 Diferencias entre arrays y clase vector
- 7 Cadenas de caracteres estilo C

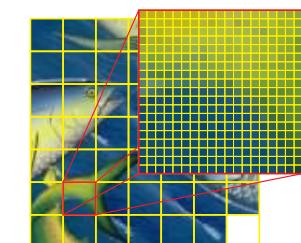
Matrices de más de 2 dimensiones

Podemos declarar tantas dimensiones como queramos añadiendo más corchetes.



```

1 int main(){
2     const int FIL = 4;
3     const int COL = 5;
4     const int PROF =3;
5     double mat[PROF][FIL][COL];
6
7     double puzzle[7][7][19][19];
8 }
```



Funciones y matrices

- Para pasar una matriz hay que especificar todas las dimensiones menos la primera

```
void lee_matriz(double m[][COL], int util_fil, int util_col);
```

- COL no puede ser local a main. Debe ser global

```

1 const int FIL=20, COL=30;
2
3 void lee_matriz(double m[] [COL], int util_fil,
4                 int util_col);
5
6 int main(){
7     double m[FIL] [COL];
8     int util_fil=7, util_col=12;
9
10    lee_matriz(m, util_fil, util_col);
```

```

1 #include <iostream>
2 using namespace std;
3 const int FIL=20, COL=30;
4 void lee_matriz(double m[][] [COL],
5                 int util_fil, int util_col){
6     for (int f=0 ; f<util_fil; f++)
7         for (int c=0 ; c<util_col ; c++){
8             cout << "Introducir el elemento (" 
9                 << f << "," << c << "): ";
10            cin >> m[f] [c];
11        }
12    }
13 int lee_int(const char mensaje[], int min, int max){
14     int aux;
15     do{
16         cout << mensaje;
17         cin >> aux;
18     }while ((aux<min) || (aux>max));
19     return aux;
20 }
21 void busca_matriz(const double m[][] [COL], int util_fil,
22                     int util_col, double elemento,
23                     int &fil_encontrado, int &col_encontrado){

```



```

24     bool encontrado=false;
25     fil_encontrado = -1; col_encontrado = -1;
26     for (int f=0; !encontrado && (f<util_fil) ; f++)
27         for (int c=0; !encontrado && (c<util_col) ; c++)
28             if (m[f][c] == elemento){
29                 encontrado = true;
30                 fil_encontrado = f;
31                 col_encontrado = c;
32             }
33     }
34
35 int main(){
36     double m[FIL] [COL];
37     int fil_enc, col_enc, util_fil,
38         util_col;
39     double buscado;
40
41     util_fil = lee_int("Introducir el número de filas: ",
42                         1, FIL);
43     util_col = lee_int("Introducir el número de columnas: ",
44                         1, COL);
45     lee_matriz(m, util_fil, util_col);
46     cout << "\nIntroduzca elemento a buscar: ";

```

```

47     cin >> buscado;
48
49     busca_matriz(m, util_fil, util_col, buscado,
50                   fil_enc, col_enc);
51     if (fil_enc != -1)
52         cout << "Encontrado en la posición "
53             << fil_enc << "," << col_enc << endl;
54     else
55         cout << "Elemento no encontrado\n";
56
57     return 0;
58 }

```

Contenido del tema

- 1 Definición
- 2 Declaración y representación en memoria
- 3 Operaciones con arrays
 - Acceso a las componentes
 - Inicialización
 - Asignación
 - Lectura y escritura
- 4 Sobre el tamaño de los arrays
- 5 Funciones y arrays
 - Construcción de arrays en funciones
 - Trabajando con arrays locales
- 6 Diferencias entre arrays y clase vector
- 7 Cadenas de caracteres estilo C
- 8 Declaración e inicialización de matrices de 2 dimensiones
- 9 Operaciones con matrices
 - Acceso, asignación, lectura y escritura
- 10 Sobre el tamaño de las matrices
- 11 Matrices de más de 2 dimensiones
- 12 Funciones y matrices
- 13 Gestión de filas de una matriz como arrays

Gestión de filas de una matriz como arrays I

- Dado que los elementos de cada fila están contiguos en memoria, podemos gestionar cada fila como si fuese un array.
- La fila i -ésima de una matriz m es $m[i]$.
- Cada fila $m[i]$ tiene `util_col` componentes usadas

```

1 void busca_matriz(const double m[][COL], int util_fil,
2                     int util_col, double elemento,
3                     int &fil_encontrado, int &col_encontrado){
4     fil_enc = -1;
5     col_enc = -1;
6     for (int f=0; col_enc == -1 && (f<util_fil); f++)
7         col_enc = busca_sec(m[f], util_col, elemento);
8     if (col_enc != -1)
9         fil_enc = f-1;
10 }
```

Gestión de filas de una matriz como arrays II

- Como toda la matriz está contigua en memoria, si la matriz está completamente llena, podemos hacer

```

1 void busca_matriz(const double m[][COL], double elto,
2                     int &fil_encontrado, int &col_encontrado){
3     int encontrado = busca_sec(m[0], COL*FIL,
4                                 elto);
5     if (encontrado != -1){
6         fil_encontrado = encontrado / COL;
7         col_encontrado = encontrado % COL;
8     } else{
9         fil_encontrado = -1;
10        col_encontrado = -1;
11    }
12 }
```