

## Paso de argumentos: array

Las funciones son esenciales para descomponer un problema en subtareas, haciendo que cada función sea responsable de cierta parte del trabajo. Es fundamental conocer la forma en que se pasan arrays a funciones, tanto para entrada como para salida de datos.

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

## Paso de argumentos: array

Función cuya responsabilidad será la de imprimir el contenido de un array de caracteres. El array de caracteres se pasa a la función como argumento.

```
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 }
```



## Paso de argumentos: array

### Consideraciones:

- la función asume que el tamaño del array es 5. ¿Es general esto?
- ¿qué ocurre si deseamos imprimir un array de enteros? ¿sirve esta función? ¿se genera error de compilación?

**Nota:** si necesitamos usar el mismo método para diferentes tipos de datos, habrá que implementar una función para cada tipo.

## Paso de argumentos: array

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




## Paso de argumentos: array

El error de ejecución no se traduce siempre en un core..... Puede que se muestren caracteres raros en pantalla (la conversión de las posiciones de memoria fuera del array a caracteres). Siempre hay que evitar esto, ya que el comportamiento del programa es impredecible.

```
a e i o u
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4
a e i o u   0 1 2 3 4 5 6 7 8 9 @   P %   8   C   m   (   X %   s @
```

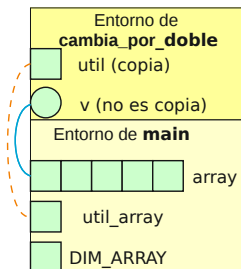


Los arrays es como si se pasasen por referencia, en el sentido de que podemos modificar las componentes pero **no hay que poner &**.



```

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 cambia_por_doble(int v[], int util){
9     for (int i=0; i<util; i++)
10         v[i] *= 2;
11 }
12 int main(){
13     const int DIM_ARRAY = 5;
14     int array[DIM_ARRAY]={4,2,7};
15     int util_array=3;
16     cout << "Original: ";
17     imprime_array(array, util_array);
18     cout << endl << "Modificado: ";
19     cambia_por_doble(array, util_array);
20     imprime_array(array, util_array);
21 }
  
```



## Paso de argumentos: array

La salida del programa anterior es la siguiente (observad que el array ha quedado modificado)

```
Original: 4 2 7  
Modificado: 8 4 14
```



A considerar:

- ¿habría algún problema en que el nombre del array, en el **main**, sea **array**, y en **cambia\_por\_doble** sea **v**?
- ¿habría algún problema si el parámetro de las funciones llamado **util** se llamase también **util\_array**?

Debe quedar clara la relación entre parámetros actuales y formales.

## Paso de argumentos: array

### Problema

¿Cómo evitamos que se puedan modificar los elementos contenidos en el array? ¿Interesa que el método que imprime el contenido del array pueda cambiar su contenido?



## Paso de argumentos: array

### Problema

¿Cómo evitamos que se puedan modificar los elementos contenidos en el array? ¿Interesa que el método que imprime el contenido del array pueda cambiar su contenido?

### 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 cambia_por_doble(const int v[], int util){
6     for (int i=0; i<util; i++)
7         v[i] *= 2; // ERROR de compilación
8 }
```

# Paso de argumentos: array

## Atención al error de compilación:

```
imprimedoble2.cpp: En la función 'void cambia_por_doble(const int*, int)':  
imprimedoble2.cpp:10:15: error: asignación de la ubicación de sólo lectura  
    *(v + ((sizetype)(((long unsigned int)i) * 4ul)))'
```



## Paso de argumentos: array

- 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:

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



# Devolución de arrays por 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 la función **llamante** y pasarlo como parámetro.

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



## Devolución de arrays por funciones II

```
1 void solo_pares(const int v[], int util_v,  
2                 int pares[], int &util_pares){  
3     util_pares=0;  
4     for (int i=0; i<util_v; i++)  
5         if (v[i]%2 == 0){  
6             pares[util_pares]=v[i];  
7             util_pares++;  
8         }  
9 }  
10 void imprime_array(const int v[], int util){  
11     for (int i=0; i<util; i++)  
12         cout << v[i] << " ";  
13 }
```

# Ejemplo de devolución de un array por una función

## Ejemplo

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

# Ejemplo de devolución de un array por una función

```
1 #include <iostream>
2 using namespace std;
3 /**
4  * Metodo para imprimir vector: nos aseguramos que el vector no se modificara
5  * @param vector a imprimir
6  * @param numero de elementos en el vector
7  */
8 void imprime_array(const char v[], int util);
9 /**
10  * Metodo para quitar repetidos: solo si son valores consecutivos
11  * @param vector original
12  * @param contador de elementos en el vector original
13  * @param vector de destino
14  * @return contador de elementos en el vector resultado
15  */
16 int quita_repes(const char original[], int util_original, char destino[]);
```



## Ejemplo de devolución de un array por una función

```
1 // Quitar repetidos consecutivos
2 int quita_repes(const char original[], int util_original, char destino[]){
3     int util_destino=1;
4     // Se copia el primero tal cual
5     destino[0] = original[0];
6     // Bucle de recorrido del vector: desde la primera posicion
7     // en adelante. Se copia el valor si no es igual al previo
8     for (int i=1; i<util_original;i++){
9         if (original[i] != original[i-1]){
10             destino[util_destino] = original[i];
11             util_destino++;
12         }
13     }
14     // Se devuelve el contador de elementos
15     return util_destino;
16 }
```

# Ejemplo de devolución de un array por una función



```
1 // Metodo de impresion
2 void imprime_array(const char v[], int util){
3     for (int i=0; i<util; i++){
4         cout << v[i] << " ";
5     }
6 }
7 int main(){
8     const int DIM =100;
9     char entrada[DIM]={ 'b', 'b', 'i', 'e', 'n', 'n', 'n' }, salida[DIM];
10    int util_entrada = 7, util_salida;
11
12    // Se quitan los repetidos
13    util_salida=quita_repes(entrada, util_entrada, salida);
14    // Se muestra el vector
15    imprime_array(salida, util_salida);
16 }
```

# Ejemplo de devolución de un array por una función

Entrada: b b i e n n n

Salida: b i e n



# Trabajando con arrays locales a funciones

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

Algoritmo:

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

# Trabajando con arrays locales a funciones

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

Algoritmo:

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

## Problema

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

# Trabajando con arrays locales a funciones

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

Algoritmo:

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

## Problema

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 a funciones

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];
```

# Trabajando con arrays locales a funciones

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).



# Trabajando con arrays locales a funciones

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 la implementación de **capicua** de la definición de la constante.

# Trabajando con arrays locales a funciones

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 la 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
10 void imprimevector(const int v[], int util){
11     for (int i=0; i<util; i++)
12         cout << v[i] << " ";
13 }
```

```
1 void quita_nodigitos(const int original[],
2     int util_original, int destino[],
3     int &util_destino){
4     util_destino=0;
5     for (int i=0; i<util_original; i++)
6         if (original[i] > -1 && original[i] < 10){
7             destino[util_destino]=original[i];
8             util_destino++;
9         }
10 }
```

```
1 bool capicua(const int v[], int longitud){
2     bool escapicua = true;
3     int solodigitos[DIM];
4     int long_real;
5
6     quita_nodigitos(v, longitud, solodigitos, long_real);
7     for (int i=0; i< long_real/2 && escapicua; i++)
8         if(solodigitos[i] != solodigitos[long_real-1-i])
9             escapicua = false;
10    return escapicua;
11 }
```



```
1 int main(){
2     int entrada1[DIM]={1,2,3,4,3,2,1};
3     int util_entrada1=7;
4     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};
5     int util_entrada2=24;
6
7     imprimevector(entrada1, util_entrada1);
8     if (capicua(entrada1, util_entrada1))
9         cout << " es capicua\n";
10    else
11        cout << " no es capicua\n";
12    imprimevector(entrada2, util_entrada2);
13    if (capicua(entrada2, util_entrada2))
14        cout << " es capicua\n";
15    else
16        cout << " no es capicua\n";
17 }
```

## 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 }
```



# 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 }
```





# Funciones y matrices

## Paso de matrices como parámetro de funciones y métodos

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

- Ejemplo:

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

# Funciones y matrices

## Paso de matrices como parámetro de funciones y métodos

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

- Ejemplo:

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

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

```
const int FIL=20, COL=30;
void lee_matriz(double m[][COL], int util_fil, int util_col);

int main(){
    double m[FIL][COL];
    int util_fil=7, util_col=12;
    lee_matriz(m, util_fil, util_col);
}
```

## Problema

Hacer un programa para buscar un elemento en una matriz 2D de doubles.

```
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 }
```

```
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     bool encontrado=false;
5     fil_encontrado = -1; col_encontrado = -1;
6     for (int f=0; !encontrado && (f<util_fil) ; f++)
7         for (int c=0; !encontrado && (c<util_col) ; c++)
8             if (m[f][c] == elemento){
9                 encontrado = true;
10                fil_encontrado = f;
11                col_encontrado = c;
12            }
13 }
```

```
1 int lee_int(const char mensaje[], int min, int max){
2     int aux;
3     do{
4         cout << mensaje;
5         cin >> aux;
6     }while ((aux<min) || (aux>max));
7     return aux;
8 }
9 int main(){
10     double m[FIL][COL];
11     int fil_enc, col_enc, util_fil, util_col;
12     double buscado;
13
14     util_fil = lee_int("Introducir el número de filas: ",
15                        1, FIL);
16     util_col = lee_int("Introducir el número de columnas: ",
17                        1, COL);
18     lee_matriz(m, util_fil, util_col);
```



```
1  cout << "\nIntroduzca elemento a buscar: ";
2  cin >> buscado;
3
4  busca_matriz(m, util_fil, util_col, buscado,
5              fil_enc, col_enc);
6  if (fil_enc != -1)
7      cout << "Encontrado en la posición "
8          << fil_enc << "," << col_enc << endl;
9  else
10     cout << "Elemento no encontrado\n";
11
12  return 0;
13 }
```

# Gestión de filas de una matriz como arrays I

## Problema

Hacer una función que encuentre un elemento en una matriz 2D de doubles.

- Supongamos que disponemos de una función que permite buscar (búsqueda secuencial) un elemento en un array:

```
int busca_sec(double array[], int utilArray,  
              double elemento);
```

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

## Gestión de filas de una matriz como arrays II

```
1 void busca_matriz(const double m[][COL], int util_fil,
2     int util_col, double elemento,
3     int &fil_enc, int &col_enc){
4     int f;
5     fil_enc = -1;
6     col_enc = -1;
7     for (f=0; col_enc == -1 && (f<util_fil); f++)
8         col_enc = busca_sec(m[f], util_col, elemento);
9     if (col_enc != -1)
10         fil_enc = f-1;
11 }
```



# Gestión de filas de una matriz como arrays III

## Otra solución

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, elto);
4     if (encontrado != -1){
5         fil_encontrado = encontrado / COL;
6         col_encontrado = encontrado % COL;
7     } else{
8         fil_encontrado = -1;
9         col_encontrado = -1;
10    }
11 }
```

# 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 }

```

