Metodología de la Programación II

Introducción

0.1 Tipos de Datos elementales

- El tamaño, rango y precisión, depende del compilador y el microprocesador.
- Consumo de memoria mínimo estándar:

  booleano ($bool$ 1 byte)

  carácter ($char$ 1 byte)

  entero (int 2 bytes)

  coma flotante o reales ($float$ 4 bytes)

  coma flotante y doble precisión ($double$ 8 bytes).

- Rangos de acuerdo con el almacenamiento estándar

  $char$(de -127 a 128)

  int(de -32768 a 32767)

  $float$(de 3.4E-38 a 3.4E+38)

  $double$(1.7E-308 a 1.7E+308)

Modificadores de tipos

Palabras reservadas que se anteponen a los nombres de los tipos básicos, sirven para ajustar los tipos elementales a necesidades concretas.

short, *long*, *signed* y *unsigned*.

Significado:

long y *short* se refieren al tamaño.

signed y *unsigned*: utilización o no del bit más significativo para el signo.

Aplicación.

char: *signed* y *unsigned*.

int: los 4 modificadores independientes y algunas combinaciones.

double: *long*.

Ejemplos.

unsigned char (de 0 a 255) *short* ó *short int* ó *int*(de -32768 a 32767) *unsigned int*(de 0 a 65535) *long* ó *long int* (de (-)2E31 a 2E31 - 1)

Literales

Secuencia de dígitos que pueden estar precedidos por:

- 0x ó 0X en el caso de representación hexadecimal (p.e. 0x345 ó 0X3A4)
- 0 en el caso de representación octal (p.e. 034)
- No estar precedidos por ninguno de los caracteres anteriores en el caso de la representación decimal. p.e 435

Pueden contener al final la posfijos (l ó L) y/o (u ó U) que significan que son *long* y *unsigned* respectivamente, p.e.j. 758463L

Los literales de coma flotante o constantes de coma flotante, pueden contener al final (f ó F ó l ó L) para indicar que son $float$ o $long\ double$.

Por omisión son de tipo $double$.

Los rangos de los tipos de C y otras propiedades se encuentran en los ficheros de cabecera `climits` y `cfloats`.

En C++ se proporcionan estas características en la plantilla $numeric\_limits$

Conversión automática de tipos

La conversión de tipos, se produce de forma automática en determinadas condiciones.

No se pierde información cuando se realiza una promoción (conversión de tipos menores a mayores, entre tipos de enteros o entre tipos de reales, p.e. de $short$ a int, ó de $float$ a $double$)

Se puede perder información en la conversión de rangos mayores a menores: **No debe usarse**.

En una asignación de variables de distintos tipos, el valor del lado derecho de la asignación se convierte al tipo del lado izquierdo.

Ejemplo int x; char ch; ch $=$ x; pudiéndose perder información al tomar ch los 8 bits más significativos de x.

En el paso de argumentos a funciones, la conversión se realiza sin problemas de tipos menores a mayores, como si existiese un $cast$ (operación que fuerza a convertir un objeto a un determnado tipo).

Tipos enumerados

Tipo definido por el usuario, conjunto de constantes enteras con nombre cuya asignación puede ser automática.

Formato de definición

$enum$ nombre_tipo {Lista de etiquetas} variables;

ó $enum$ {Lista de etiquetas};

Ejemplo

```
enum dias {lunes, martes, miercoles, jueves, viernes, sabado, domingo};
```

Cada etiqueta corresponde a un valor entero, que puede estar repetido.

Si no se indican valores de las etiquetas como en este caso, se corresponderán sucesivamente con los enteros 0, 1, 2 ...

Se puede indicar el valor de las etiquetas

```
enum dinero { euro = 1, euro_dos = 2, billete_diez = 10, billete_cien = 100};
```

Podemos realizar asignaciones y comparaciones. Si declaramos: dinero pasta;

```
dinero pasta;
pasta = billete_diez;
.....
if (euro == billete_diez){...}
```

Si no se indican valores de algunas etiquetas, se corresponderán con el siguiente de la etiqueta anterior.

```
enum tipo_enum {etq1 = 2, etq2, etq3 = 20, etq4 = 22};
```

En esta enumeración, `etq2` valdrá 3.

Algunas consideraciones finales:

1. Las etiquetas se pueden usar en cualquier lugar donde se espere una expresión entera.

2. Una variable de tipo enumerada, puede asignársele a cualquier variable de tipo entero.

3. Los nombres de las etiquetas tienen que ser distintos, incluso en tipo enumerados distintos.

4. Las etiquetas no pueden ser leidas, ni escritas directamente.

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

enum gradoAltura {MUYALTO=10, ALTO=7, MEDIO=5, BAJO=3, MUYBAJO=1};

gradoAltura lee_altura (void);
void        muestra_altura (gradoAltura);
void        escribe_resultado (gradoAltura);


int main()
{

   int altura;
   gradoAltura altura_subjetiva;
```

```
    altura_subjetiva = lee_altura();
    cout << "Una persona de esa altura se considera: " << endl;
    muestra_altura(altura_subjetiva);
    escribe_resultado (altura_subjetiva);
    return 0;
}

gradoAltura lee_altura()
{
    gradoAltura tmp;
    int altura;

    cout << "Introducir la altura en cm. " << endl;
    cin >> altura;
```

```
    if (altura < 150)
       tmp = MUYBAJO;
    else
       if (altura < 160)
          tmp = BAJO;
       else
          if (altura < 170)
             tmp = MEDIO;
          else
             if (altura < 180)
                tmp = ALTO;
             else
                tmp = MUYALTO;
     return tmp;
}
```

```
void muestra_altura (gradoAltura a)
{
  switch (a) {
   case (MUYBAJO) :
      cout << "\t Muy bajo" << endl;
      break;
   case (BAJO) :
      cout << "\t Bajo" << endl;
      break;
   case (MEDIO) :
      cout << "\t Medio" << endl;
      break;
   case (ALTO) :
      cout << "\t Alto" << endl;
      break;
```

```
    case (MUYALTO) :
       cout << "\t Muy alto" << endl;
       break;
    default:
       break;
  }
}

void escribe_resultado (gradoAltura altura_subjetiva)
{
  if (altura_subjetiva < 7)
   cout << "\t Lo siento, no se admite "<< endl;
  else
   cout << "\t Felicidades, tiene opciones"<<endl;
}
```

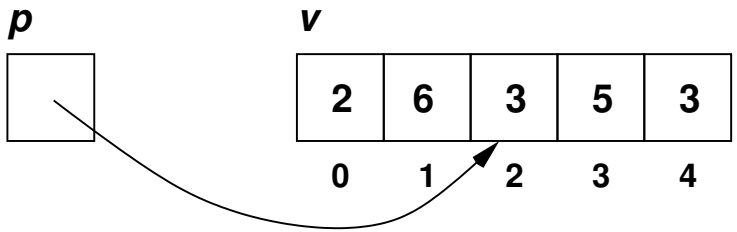
0.2 Punteros y vectores

Íntima relación entre vectores y punteros.

```
int v[5] = {2, 6, 3, 5, 3};
int *p;
```

Los componentes del vector `v` son de tipo entero, por lo que `p` *puede referenciar a cada uno de éstos*, asignando su dirección a `p`.

```
p = &v[2]; // *p devolverá 3 (valor almacenado en v[2]).
```

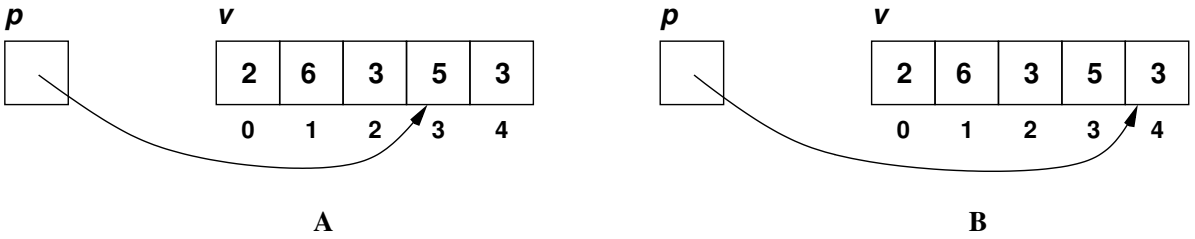


Un puntero que referencia a una casilla de un vector

Aritmética de direcciones

1. **Operaciones de incremento y decremento**.

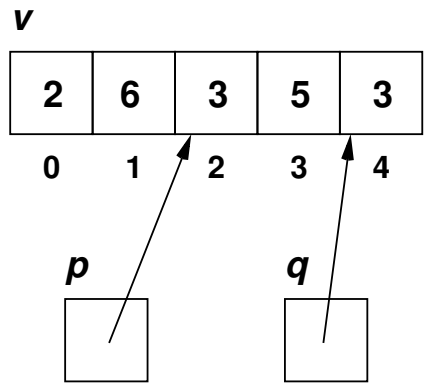
```
p++      p = p+i    p += i
p--      p = p-i    p -= i
```



A) Efecto de ejecutar p++. B) Efecto de ejecutar p+=2

2. **Operaciones con punteros que referencian a objetos del mismo vector**.

```
p = &v[2];
q = &v[4];
```



a) Operadores relacionales: ==, !=, <, >, <= y >=.
- p==q es *falsa* (p!=q es *verdadera*). Sin embargo, *p==*q es *verdadera*
- p<q es *verdadera* (p>q es *falsa*).
- p<=q es *verdadera* y p>=q es *falsa*.

b) Si q>=p, La expresión q-p es válida, y devuelve el número de elementos entre p y q (en realidad, es q-p+1).

Traducción de la notación de índices a direcciones de memoria

Equivalencia clave: `p = &v[0];` `p = v`

ya que, escribir `&v[0]` equivale a escribir `v`.

- `*p == *(&v[0]) == v[0]`
- `p+1 == &v[1]`, entonces,

  `*(p+1) == *(&v[1]) == v[1]`
- En general, `p+i == &v[i]`, por lo que `*(p+i) == *(&v[i]) == v[i]`

0.3 Tipo referencia.

Una variable de tipo referencia es un *alias* para una variable o constante,

```
int a=5;
int &r = a;
```

Las variables de referencia deben ser iniciadas cuando se crean, antes de usarse.

En el caso de ser iniciadas a constantes, las referencias deben ser constantes.

El uso normal de las referencias es el paso de argumentos a las funciones, la devolución de refencias en las funciones y en los operadores sobrecargados.

0.4 Constantes

Objetivo: Evitar poder modificar el valor de una variable.

En C no es totalmente efectivo:

```
const int a =3;
int *p = &a;
(*p)++;
```

En C++ sí es efectivo, con el modificador `const` no es posible modificar el valor de la variable a.

Si se dispone de una variable de tipo puntero, podemos contemplar tres casos:

- *const* int *p;

  No se permite modificar el entero al que apunta p, pero sí la dirección almacenada en p

- int * *const* p;

  No se permite modificar la dirección almacenada en p, pero sí el valor entero al que apunta p.

- *const* int * *const* p;

  No se permite modificar, ni el valor de p, ni el valor al que apunta p.

En el caso de disponer de un vector de un determinado tipo, se puede utilizar al modificador de acceso *const*, evitando la modificación de sus elementos.

```
const int v[]=  {3, 5, 7, 4 };
```

Si se intenta modificar algún elemento del vector v, se produciría un error.

0.5 Modificador de almacenamiento *static*

El modificador *static* permite que los objetos se creen antes de la ejecución del módulo en el que están declarados, y no en tiempo de ejecución, como ocurre con los datos locales.

Como consecuencia:

- El almacenamiento de un objeto local *static* **no se realiza en la pila**.
- Su valor se mantendrá aunque se salga del ámbito donde estan definidos.

Por omisión, las variables locales numéricas *static*, igual que las globales, se inician a cero, a diferencia de las locales que contienen basura.

```
#include <iostream>
using namespace std;
void f();

int main()
{
   f();
   f();
   return 0;
}
```

```
void f()
{
   int a=0;
   static int b;
   a++;
   b++;
   cout << a << "  " << b << endl;
   return;
 }
```

El resultado es:

```
 1 1
 1 2
```

Este ejemplo muestra que la variable *static* `b` se inicia solo una vez a cero, y **no** se vuelve a iniciar.

Puede aplicarse el modificador *static* a variables globales, que solo serán reconocidas y manipuladas en el archivo donde han sido declaradas.

Puede aplicarse el modificador *static* a funciones, pudiendo utilizar dichas funciones solo en el archivo donde están definidas.

0.6 Características de algunos operadores

Los operadores **incremento** ++ **y decremento** -- preceder o seguir al operando.

Si el operador precede al operando, se incrementa o decrementa antes de utilizar el valor del operando, y si no, utiliza su valor antes de incrementarlo o decrementarlo.

Ejemplo

```
x=10;        x=10;
y=++x;       y=x++;
```

El valor de `y` en los dos casos es distinto, en el primero es 11, y en el segundo es 10. En ambos casos el valor final de `x` es 11.

El **operador** `? :` es una forma simplificada de expresar las sentencias if-else

Formato: *Exp1* `?` *Exp2* `:` *Exp3*

Se evalúa *Exp1*, si es verdadera, se evalúa *Exp2*, y la expresión final toma ése valor; si *Exp1* es falso, se evalúa *Exp3*, y ése es el valor que toma la expresión.

```
x = 10;
if (x>9) y = 100;
else     y = 200;
```

```
x = 10;
y = (x >9) ? 100 : 200;
```

```
if (x %2 == 0)
   cout << "PAR";
else
   cout << "IMPAR";
```

```
cout << (x %2 == 0) ? "PAR" : "IMPAR";
```

Operador ,

Permite situar múltiples expresiones dentro de un paréntesis.

Las expresiones se evalúan de izquierda a derecha, y la expresión resultante asume el valor y el tipo de la última expresión evaluada.

```
x = (y=3, y + 1);
```

Asigna 3 a `y`, después evalúa `y+1`, obteniendo 4, y a continuación se asigna a `x` el resultado de la evaluación de la expresión, 4.

Se usa fundamentalmente en las sentencias de control `for`

```
for(x=10, y =30; y<100; x++,y++){...}
```

Las comas en los argumentos de una función o en las declaraciones de variables, no son operadores coma: **no** se garantiza la evaluación de izquierda a derecha.

Operador de *casting* o *moldeado*

Se utiliza para forzar a que una expresión sea de un determinado tipo, siendo responsabilidad del programador, si en dicha conversión, se puede o no perder información. El formato es: (*tipo*) *exp*

```
int x;
float y = (float) x / 2;
```

El tipo de la expresión `x / 2` sería `int`, sin embargo con el *molde* expresado, `x` se trata como `float`, por lo que el resultado es `float`.

Aunque en C++, se admite este tipo de *moldeado* se aconseja el uso de *moldes* más específicos:

`static_cast<`*tipo*`>(`*exp*`)` : tipos simples (enteros, reales, ...)

`reinterpret_cast<`*tipo*`>(`*exp*`)` : *tipo* y *exp* son del tipo *puntero a un tipo*.

0.7 typedef

Permite crear un alias para un tipo de dato ya definido. **No crea un nuevo tipo**.

Ventaja: Programas más fáciles de leer, y si se utilizan tipos que dependan de la máquina, solo los `typedef` requerirían cambios.

Formato: `typedef` tipo *nombre_de_alias*;

```
typedef unsigned char byte;
byte edad;  // edad (unsigned char)

typedef float errores;
errores err_medida;  // err_medida (float)
```

```
typedef int  *Ptr;       // puntero (puntero a entero)
Ptr puntero, *Ptr_Ptr    // Ptr_Ptr (puntero a puntero a un entero)

typedef double Dblfun(), Dblfun2Ent(int, int);

Dblfun fun_array[10];    // (ERROR: no existen arrays de funciones)

Dblfun f1(int x){...}    // (ERROR: una funcion no puede devolver
                         //         una funcion)

Dblfun2Ent fun2;         // fun2 (funcion que recibe dos enteros y
                         // devuelve un double)
```

0.8 Parámetros formales de las funciones

En **C**, el paso de argumentos a una función **siempre es por valor**:

1. El parámetro formal es un tipo no puntero.
2. El parámetro formal es un puntero a un tipo.

```
int a=1; float b= 3.0;
int f( int pa, float * pb){...}
int c = f(a, &b);
```

Los argumentos pasados a las funciones **son variables locales**.

- Se crea una copia del argumento en la pila, en el primer caso de un tipo no puntero, y en el segundo de tipo puntero.

- Los cambios realizados en el cuerpo de definición de la función no afectan a los valores de los argumentos, es decir, **los argumentos permanecen inalterables al finalizar la ejecución de la función**.

- Si el argumento es un puntero, éste valor no puede ser modificado, pero sí los valores que existen en la dirección a la que apuntan. *En este sentido se dice que es un paso por referencia en C*.

Por lo tanto, en la llamada a una función en C, los argumentos son valores fijos, ya que aunque se pase una variable, su valor no puede ser alterado.

Caso especial: El parámetro formal es un vector, ya que el nombre de un vector contiene la dirección del primer elemento.

```
int v[]={3,5,7};
int f(int a[])
{...}
int c = f(v);
```

En **C++** es posible el paso por referencia real, mediante **referencias**.

```
float a=5.0;
int f(int pa, float &pr)
{...}
int c = f(3,a);
```

Las modificaciones que se realicen dentro del cuerpo de la función sobre `pr`, afectan permanentemente al valor de la variable `a`.

Un parámetro por referencia funciona como un puntero constante, que es desreferenciado cuando se utiliza.

En el paso por referencia el argumento siempre debe ser una variable, no una constante, ni una expresión.

Si no deseamos modificar el argumento pasado por referencia, debemos declarar el parámetro formal, constante.

```
int f(const float &par) {...}
```

El modificador $const$ se puede aplicar a cualquier tipo de parámetro formal.

Si en una función `f` se hace una llamada a otra función `f1`, donde aparecen parámetros formales con el modificador $const$, y en la función `f` se hace referencia a los parámetros $const$ de `f1`, éste aspecto deberá reflejarse con el mismo tipo de modificador $const$ en la declaración de los parámetros formales de `f`.

0.9 Algunos aspectos sobre las funciones.

Escribir código que dependa del orden de evaluación, no es buena práctica porque la evaluación de expresiones de la forma:

```
x = f(x) + g(x)
```

dependen del compilador, ya que no se conoce *a priori* s se evaluará primero `f(x)` ó `g(x)`.

Lo mismo ocurre con los argumentos de una función que tienen que ser previamente evaluados.

0.10 Prototipo de una función.

Puede invocarse una función definida por el usuario antes de haberla definido, aunque es necesario, al menos, que esté previamente declarada.

Es conveniente realizar la declaración al principio del programa: **prototipo**.

> Un **prototipo** es una declaración de una función, que **informa del número, orden y tipo de parámetros formales**, así como del **tipo que devuelve**.

Debe aparecer antes de ser llamada la función, usualmente al principio del programa.

Ejemplo

```
int fun( int a, float *b, double &c);
```

que puede ser expresado como

```
int fun( int, float *, double &);
```

- Debe coincidir con la cabecera de la definición de la función.
- Si la función no tiene parámetros, debemos utilizar la palabra `void` como lista de parámetros.
- Las funciones prototipo permiten al compilador detectar errores en el número y/ ó tipo de argumentos.

```
int main()
{
   int b=10;
   b = incrementa(&b);
   return 0;
}

int incrementa(int *p)
{
   (*p)++;
   return (*p);
}
```

Este programa daría error al compilarlo, ya que en `main()`, no se tiene conocimiento de la existencia de la función incrementa.

```
int incrementa(int *);  // Prototipo de la funcion

int main()
{
   int b=10;
   b = incrementa(&b);
   return 0;
}

int incrementa(int *p)
{ ... }
```

Ahora sí se compilaría, ya que se ha incluido el prototipo de la función `incrementa()`

```
int incrementa(int *p)
{
   (*p)++;
   return (*p);
}
int main()
{
   int b=10;
   b = incrementa(&b);
   return 0;
}
```

Este programa funcionaría pero este estilo no es aconsejable.

Es una buena costumbre de programación escribir los prototipos de las funciones que se definen y dejar que la función `main()` sea la primera.

0.11 Ficheros de cabecera

Una costumbre usual es incluir todos las funciones prototipo *similares* en ficheros llamados de cabecera (extensión `.h`), e incluirlos mediante la directiva:

`#include <`*nombre*`>` ó `#include "`*nombre*`"`

En C++ los ficheros de cabecera estándar de C, aparecen con una la letra *c* delante: $cmath$, $cstring$, $cstdlib$, ...

Los ficheros de cabecera contienen los prototipos de las funciones definidas en los módulos asociados, (ficheros creados por el usuario, módulos de bibliotecas propias del compilador o creadas por el usuario) que son utilizados por el programa.

Además, pueden contener definiciones de tipos, de constantes, etc.

Cuando se llama a una función en un fichero, y no está definida en el propio fichero, deberá estar definida en otro módulo (otro fichero objeto, biblioteca etc.), y debe existir un fichero de cabecera paralelo a dicho módulo que haya sido incluido.

Ejemplo:

Cuando usamos `cin` en cualquier módulo, como su declaración (prototipo) está en el fichero de cabecera $iostream$, debe incluirse mediante

```
#include <iostream>
```

0.12 Paso de vectores a funciones

Cuando se pasa un vector a una función se pasa el *nombre* del vector, y como ya sabemos, equivale a pasar la dirección del primer elemento del vector.

```
int z[10];
......
f (z);
......
```

La función que recibe el vector puede hacerlo de tres maneras:

1. Especificando el número de casillas del vector.

```
void f (int v[10])
{
......
}
```

El compilador descarta el número de casillas que escribimos.

2. Sin especificar el número de casillas del vector.

```
void f (int v[])
   {
   ......
   }
```

Equivale a la anterior.

3. Mediante un puntero al primer elemento (primera casilla) del vector.

```
void f (int *v)
{
.......
}
```

`v` *es un puntero a (un vector de) enteros*.

En este caso, `v` apuntará al primero de ellos.

Todos los pasos de vectores se hacen por referencia

Paso de “partes” de vectores a funciones

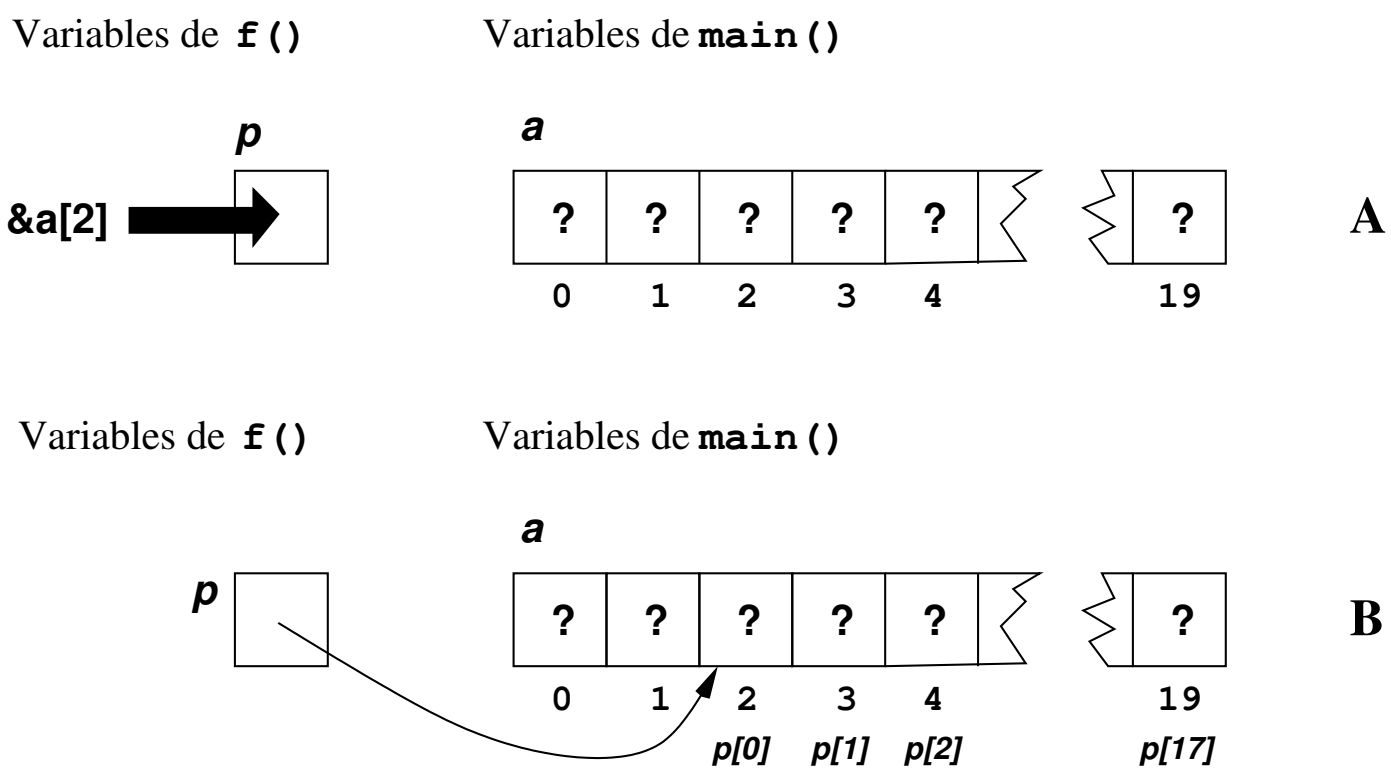
Haremos que la función que recibe el vector “crea” que éste comienza en una posición diferente.

```
int a[20];
   ......
int f (int *p);
```

Una llamada de este tipo:

```
f (&a[2]); // equivalentemente, f (a+2);
```

hará que la función `f()` *crea* que el vector empieza en la posición 2 de `a`, por lo que para la función, `p[0]` será en realidad `a[2]`, `p[1]` será en realidad `a[3]`, y así sucesivamente.



A) Correspondencia entre parámetros actuales y formales el la llamada a `f()`. B) Estado de la memoria tras establecer la correspondencia entre parámetros actuales y formales.

La función podrá acceder a las casillas `p[-1]` y `p[-2]` para leer y escribir en éstas sin ningún problema: *la memoria asignada a estas casillas está reservada*.

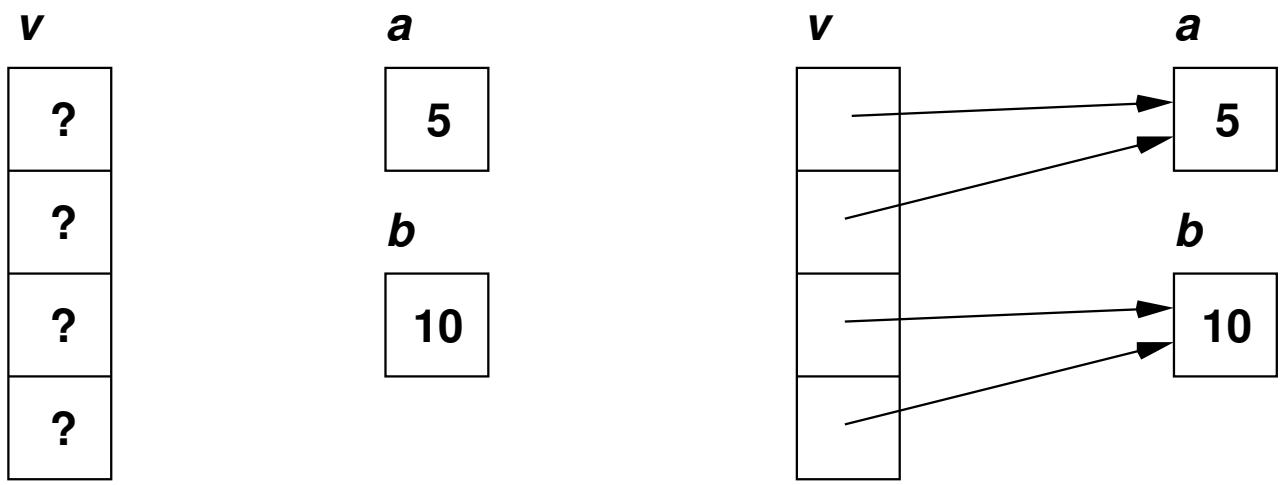
¿Qué ocurre con la llamada `f (&a[-1])`?

0.13 Vectores de punteros (1)

Diferenciar: `int v1[4];` `int *v2[4];`

```
/*********************************************************/
// Programa: vec_pun.cpp
// Declaracion y uso de un vector de punteros
/*********************************************************/

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

```
int main (void)
{
   const int TOPE = 4;

   int *v[TOPE]; // Vector de TOPE punteros a enteros
   int  i, a = 5, b = 10;

   v[0] = v[1] = &a;  // v[0] y v[1] apuntan a "a"
   v[2] = v[3] = &b;  // v[2] y v[3] apuntan a "b"

   for (i = 0; i < TOPE; i++)   // Mostrar el contenido
      cout << *(v[i]) << "  ";  // de los objetos
   return (0);                 // referenciados por "v[i]"
}
```



A **B**

Estado de la memoria: A) tras la declaración, B) tras la iniciación de los `v[i]`.

Resultado de la ejecución:

```
5   5 10 10
```

0.14 Matrices y punteros

```
/*************************************************************/
// Fichero: representacion.cpp
// Las matrices se representan en memoria por filas (secuencialmente)
/*************************************************************/
#include <iostream>
#include <iomanip>

using namespace std;

#define FILS 2
#define COLS 3
#define ALTS 2
```

```
int main (void)
{
   int  m2D[FILS][COLS]     = {{1,2,3}, {4,5,6}};
   int  m3D[FILS][COLS][ALTS]={{{1,2},{3,4},{5,6}}, {{7,8},{9,10},{11,12}}};
   int f, c, a, i; // "f", "c", y "a" son indices
   int *pm;        // puntero de acceso a cada casilla

   // Recorrido "convencional" (dos indices)

   cout << "\nRecorr. convencional 2D:\n";
   for (f=0; f<FILS; f++) {
      cout << "   Fila " << f << ":";
      for (c=0; c<COLS; c++)
         cout << setw(3) << m2D[f][c];
      cout << endl;
   }
```

```
 // Recorrido secuencial de matriz 2D (puntero)

 cout << "\nRecorr. secuencial 2D:\n";
 pm = &m2D[0][0]; // primera casilla de "m2D"
 for (i=0; i<FILS*COLS; i++, pm++) cout << setw(3) << *pm;

// Recorrido "convencional" (tres indices)

 cout << "\nRecorr. convencional 3D:\n";
 for (f=0; f<FILS; f++) {
    cout << "   Fila " << f << ":" << endl;
    for (c=0; c<COLS; c++) {
       cout << "      Columna " << c << ":";
       for (a=0; a<ALTS; a++) cout << setw(3) << m3D[f][c][a];
       cout << endl;
    }
 }
```

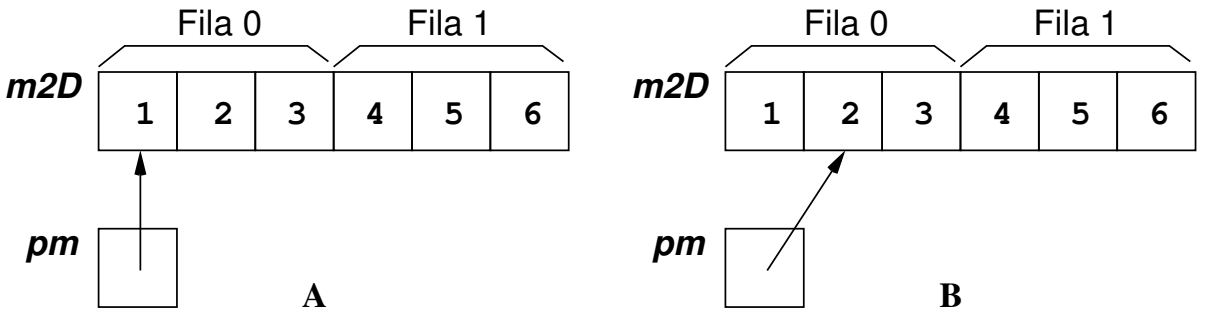
```
   // Recorrido secuencial de matriz 3D (puntero)

   cout << endl;
   cout << "Recorr. secuencial 3D:" << endl;

   pm = &m3D[0][0][0]; // primera casilla de "m3D"

   for (i=0; i<FILS*COLS*ALTS; i++, pm++)
      cout << setw(3) << *pm;

   cout << endl << endl;

   return (0);
}
```

Resultado de la ejecución:

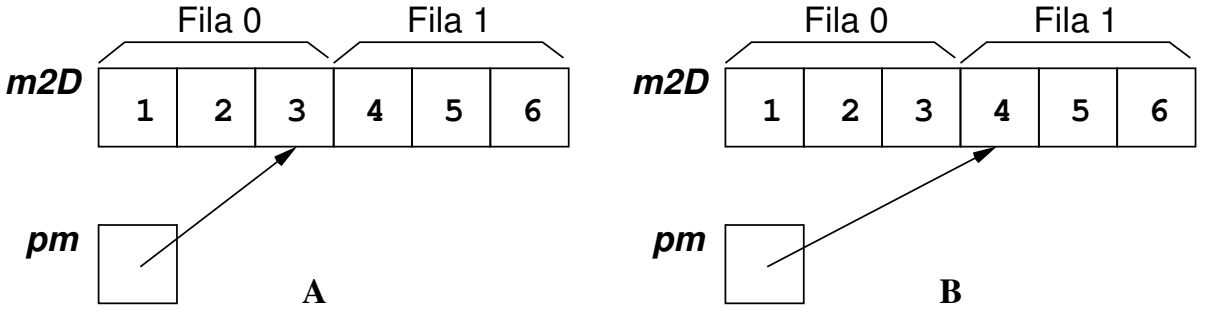
```
Recorr. convencional 2D:
   Fila 0:  1  2  3
   Fila 1:  4  5  6

Recorr. secuencial 2D:
  1  2  3  4  5  6

Recorr. convencional 3D:
   Fila 0:
      Columna 0:  1  2
      Columna 1:  3  4
      Columna 2:  5  6
   Fila 1:
      Columna 0:  7  8
      Columna 1:  9 10
      Columna 2: 11 12
```

```
Recorr. secuencial 3D:
  1  2  3  4  5  6  7  8  9 10 11 12
```

Detalle: recorrido secuencial de `m2D`:



A) `pm` referencia a `m2D[0][0]`. B) Después de imprimir `m2D[0][0]`, se incrementa `pm`



A) `pm` referencia a `m2D[0][2]`. B) Después de imprimir `m2D[0][2]`, se incrementa `pm`

0.15 Paso de matrices a funciones

Cuando se pasa una matriz a una función se pasa el *nombre* de la matriz:

```
int m [10][20];
......

f (m);
......
```

La función que recibe la matriz puede hacerlo de tres maneras:

1. Especificando *totalmente* la matriz.

```
void f (int a [10][20])
{
......
}
```

El compilador descarta el número de filas que escribimos.

2. Sin especificar la primera dimensión (es irrelevante para el compilador),

```
void f (int a [][20])
{
......
}
```

Equivale a la anterior.

3. Con un puntero al primer subvector (primera fila),

```
void f (int (*a) [20])
{
        ......
}
```

Importante: a *es un puntero a un vector de 20 enteros.*

0.16 Cadenas -clásicas- de caracteres

- Una cadena de caracteres no es más que una clase especial de los vectores de caracteres que contiene un carácter especial, `'\0'` (carácter nulo) que indica el fin de la cadena: **delimitador de fin de cadena**.

- Literales de cadena de caracteres:

```
cout <<  "Hola";
```

- Un vector de caracteres no tiene porqué ser una cadena, a no ser que:

1. Se inicie de forma adecuada.

2. Las funciones que manejan cadenas añadan el carácter `'\0'`:

   a) De forma automática, utilizando funciones de lectura o de gestión de cadenas (`cstring`)

   b) De forma "manual" con funciones construidas por el programador.

0.16.1 Iniciación de cadenas de caracteres

1. Especificando el tamaño.

   ```
   char saludo1[5] = {'H','o','l','a','\0'};
   ```

   Precaución: Hay que reservar para el '\0'.
   Puede reservarse más de lo necesario.

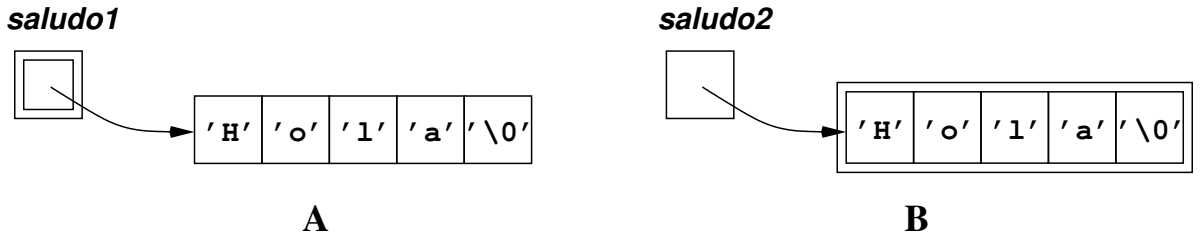
2. Sin especificar el tamaño.

   ```
   char saludo1[] = {'H','o','l','a','\0'};
   ```

   Se reserva lo estrictamente necesario.

3. Cadenas constantes.

```
char *saludo2 = "Hola";  // Las dos son
char saludo2[]="Hola";   // equivalentes
```



A) Representación de la cadena *saludo1*. B) Representación de la cadena constante *saludo2*

0.16.2 Funciones de lectura/escritura de cadenas

- El operador << está sobrecargado para las cadenas de caracteres clásicas.

```
// Fichero: EscribeCadenas.cpp
#include <iostream>
using namespace std;

int main (void)
{
   char  saludo0[5] = {'H','o','l','a','\0'};
   char  saludo1[] = {'H','o','l','a','\0'};
   char *saludo2 = "Hola";
```

```
    cout << saludo0 << " " << saludo1 << " "
         << saludo2 << endl<< endl;

    return (0);
}

% EscribeCadenas
Hola Hola Hola
```

- Pare lectura de cadenas proponemos:

  `istream & getline (char *p, int n);`

  Lee, como mucho $n-1$ caracteres de la entrada estándar.
  Termina antes si se introduce `'\n'` (ENTER). Los caracteres leidos los copia (por orden) en `p`, salvo el `'\n'`, que lo sustituye por `'\0'`.
  La memoria referenciada por `p` debe estar reservada y ser suficiente.

```
// Fichero: LeeCadena.cpp

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

```
int main (void)
{
   const int TAM = 80;
   char  cad[TAM];
   cout << "\nIntroducir cadena: ";
   cin.getline (cad, TAM);
   cout << "Cadena: "<< cad<<endl;
   return (0);
}
```

```
% LeeCadena
Introducir cadena: Una cadena multipalabra
Cadena: Una cadena multipalabra
```

0.16.3 Funciones de gestión de cadenas

Declaradas en `cstring`

1. *Funciones más relevantes*.

   ```
   char* strcpy (char* cs, const char* ct);
   ```

   Copia de cadenas. Copia la cadena `ct` en `cs`, incluyendo el carácter terminador `'\0'`.
   Devuelve `cs` (en la práctica, `void`).
   Nota: `cs` debe tener suficiente espacio.

```
char* strcat (char* cs, const char* ct);
```

Concatenación de cadenas. Concatena la cadena `ct` a la cadena `cs`.
Devuelve `cs` (en la práctica, `void`).
Nota: `s` debe tener suficiente espacio.

```
size_t strlen (const char* cs);
```

Longitud de una cadena.
Devuelve la longitud de `cs`.

```
// Fichero: cadenas1.cpp

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

int main (void)
{
   char  saludo1[20] = {'H','o','l','a','\0'};
       // Hay 15 casillas no usadas en "saludo1"

   char *nombre1 = "Pepe";
   char *espacio = " ";
   char  cad[100]; // No inicializada (ni '\0')
```

```
   strcpy (cad, saludo1);
   strcat (cad, espacio);
   strcat (cad, nombre1);
   cout  << "Cadena compuesta: " << cad << endl;

   cout << "Saludo: " << '|' << saludo1 << '|'
        <<"   Long = "<< strlen(saludo1)<<endl;

   return (0);
}

% cadenas1
Cadena compuesta: Hola Pepe
Saludo: |Hola|   Long = 4
```

```
int strcmp (const char* cs, const char* ct);
```

Comparación de cadenas. Compara `cs` y `ct`. Devuelve un valor negativo si `cs` $<$ `ct`, cero si `cs` $=$ `ct`, o positivo si `cs` $>$ `ct`.

```
char* strstr (const char* cs, const char* ct);
```

Búsqueda de subcadenas. Devuelve un puntero a la primera ocurrencia de `ct` en `cs`, o 0 si no se encuentra.

```
// Fichero: cadenas2.cpp

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

```
int main (void)
{
   const int TOPE=80;
   char  c1[TOPE];
   char  c2[TOPE];

   cout << "\nIntroducir cadena 1: ";
   cin.getline (c1, TOPE);
   cout << "\nIntroducir cadena 2: ";
   cin.getline (c2, TOPE);
   bool iguales = !strcmp(c1,c2);

   cout << endl;
```

```
if (iguales)
   cout <<'|'<< c1 <<'|'<< " y "
        <<'|'<< c2 <<'|'<< " son iguales";

else {
   cout <<'|'<< c1 <<'|'<< " y "
        <<'|'<< c2 <<'|'<< " NO son iguales ==> ";

   if (strcmp(c1,c2) > 0)
      cout <<'|'<< c1 <<'|'<<" > "<<'|'<< c2 <<'|';
   else
      cout <<'|'<< c1 <<'|'<<" < "<<'|'<< c2 <<'|';
}
cout << endl << endl;
```

```
    if (strstr(c1, c2))
       cout <<'|'<< c2 <<'|'<< " es subcadena de "
            <<'|'<< c1 <<'|';
    else
      cout <<'|'<< c2 <<'|'<< " NO es subcadena de "
           <<'|'<< c1<<'|';

    cout << endl << endl;

    return (0);
}
```

2. *Funciones sobre partes de cadenas*.

```
char* strncpy (char* cs, const char* ct, size_t n);
```

Copia como mucho `n` caracteres de `ct` en `cs`, incluyendo el carácter terminador `'\0'`. Rellena con `'\0'` si la longitud de `ct` es menor que `n`. Devuelve `cs` (ver `strcpy()`).

```
char* strncat (char* cs, const char* ct, size_t n);
```

Concatena como mucho `n` caracteres de `ct` a la cadena `cs` y añade `'\0'`. Devuelve `cs` (ver `strcat()`).

```
int strncmp (const char* cs,const char* ct,size_t n);
```

Compara como mucho `n` caracteres de `cs` y `ct` (los primeros). Devuelve un valor negativo si `cs` $<$ `ct`, cero si `cs` $=$ `ct`, o positivo si `cs` $>$ `ct` (igual que `strcmp()`).

3. *Funciones de gestión de bloques de memoria* (sucesiones de bytes).

```
void* memcpy (void* cs, const void* ct, size_t n);
```

Copia `n` bytes de `ct` en `cs`. Devuelve `cs`. *No funciona correctamente si* `ct` *y* `cs` *se solapan*.

```
void* memmove (void* cs, const void* ct, size_t n);
```

Copia `n` bytes de `ct` a `cs`. Devuelve `cs`. *Funciona correctamente aunque* `ct` *y* `cs` *se solapen*.

```
void* memset (char* cs, int v, size_t n);
```

Fija los primeros `n` bytes de `cs` al valor `v`, sustituyendo lo que hubiera. Devuelve `cs`.

```
int memcmp (const void* cs, const void* ct, size_t n);
```

Compara los primeros `n` bytes de `cs` con `ct`. Devuelve un valor negativo si `cs` $<$ `ct`, cero si `cs` $=$ `ct`, o positivo si `cs` $>$ `ct`.

```
void* memchr (const char* cs, int c, size_t n);
```

Devuelve un puntero a la primera ocurrencia de `c` en los primeros `n` caracteres de `cs`, o 0 si no se encuentra.

```
// Fichero: cadenas3.cpp

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

int main ()
{
   // memset() y memcpy()

   char barra[6], superbarra[26];
```

```
memset (barra, '*', 5);
memset (&(barra[6]), '\0', 1); // barra[6]='\0';
memcpy (superbarra, barra, 4);
superbarra[4] = '+';
memcpy (&(superbarra[5]), barra, 4);
superbarra[9] = '+';
superbarra[10] = '\0';

cout << "     Barra es: |" << barra << "|\n";
cout << "SuperBarra es: |" << superbarra << "|\n\n";

// memcpy()

char orig[]="Ejemplo";
char copia[40];
```

```
    memcpy (copia,orig,strlen(orig)+1);
    cout << " Orig es: |" << orig << "|\n";
    cout << "Copia es: |" << copia << "|\n\n";

    // memmove()

    char cad[] = "memmove puede ser muy util.....";
    cout << "  Antes: |" << cad << "|\n";
    memmove (cad+22, cad+18, 8);
    cout << "Despues: |" << cad << "|\n\n";

    return (0);
}
```

```
% cadenas3

     Barra es: |*****|
SuperBarra es: |**********+|

 Orig es: |Ejemplo|
Copia es: |Ejemplo|

  Antes: |memmove puede ser muy util.....|
Despues: |memmove puede ser muy muy util.|
```

4. *Otras*.

```
char* strerror (int n);
```

Mensajes de error predefinidos. Devuelve un puntero a la cadena definida en la implementación que corresponde al error `n`.

```
char* strchr (const char* cs, int c);
```

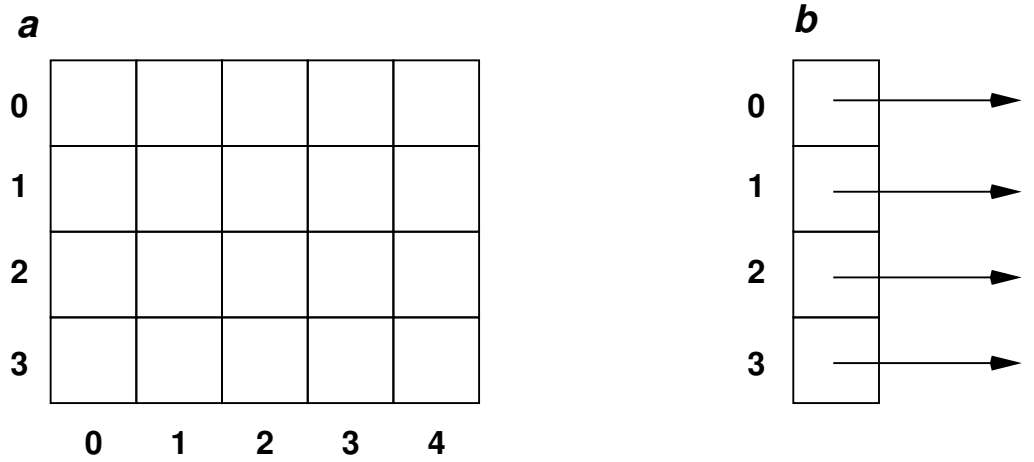
Búsqueda de caracteres (1). Devuelve un puntero a la primera ocurrencia de `c` en `cs`, o `0` si no se encuentra.

```
char* strrchr (const char* cs, int c);
```

Búsqueda de caracteres (2). Devuelve un puntero a la última ocurrencia de `c` en `cs`, o `0` si no se encuentra.

0.17 Matrices bidimensionales y vectores de punteros

```
int   a[4][5];
int  *b[4];
```



Representación en memoria de la matriz `int a[4][5]` y del vector `int *b[4]`

Con estas declaraciones, las expresiones:

`a[3][4]` `b[3][4]`

son referencias *sintácticamente válidas* a un entero.

¡OJO! Aunque *es posible* acceder al elemento `b[3][4]`, esta zona de memoria **no está reservada** (no se ha inicializado de ninguna forma).

Escribir y/o leer sobre esta estructura de datos (en la situación actual) presenta **serios problemas**:

- Escribir sobre `b[`i`][`j`]` producirá resultados inesperados, posiblemente haga que termine la ejecución del programa por un *error en tiempo de ejecución*.
- Leer los valores de estas casillas hace que obtengamos valores "basura" ya que ni `b[`i`]` ni `b[`i`][`j`]` han sido inicializados.

La iniciación de las zonas de memoria referenciadas por `b`$[i]$ debe hacerse de forma **explícita**:

1. De forma estática
   1.1. Usando un vector de punteros.
   1.2. Usando una matriz bidimensional.

2. Con funciones de asignación de memoria dinámica.

   (Tema 2)

```
// Fichero: vecpun.cpp

#include <iostream>
#include <iomanip>

using namespace std;

int main ()
{
   const int FILS = 3;
   const int COLS = 5;

   int   a[FILS][COLS];
   int  *b[FILS];
```

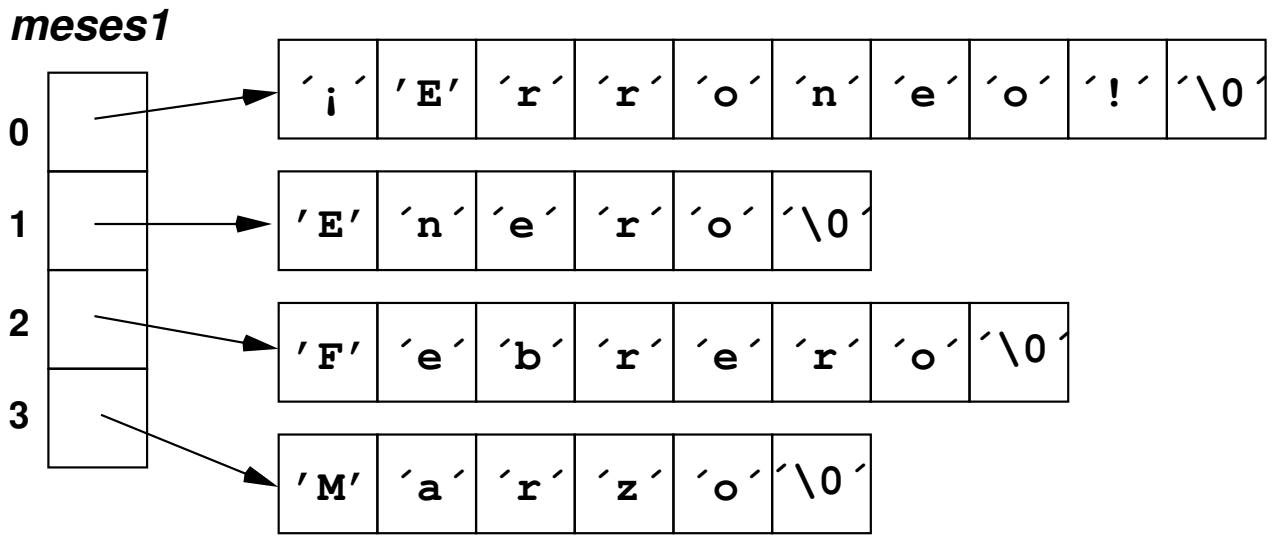
```
// Inicializacion de "a"

for (int cont=1, f=0; f<FILS; f++)
   for (int c=0; c<COLS; c++)
      a[f][c] = cont++;

cout << "Recorr. convencional de \"a\":\n";

for (int f=0; f<FILS; f++) {
   cout << "   Fila " << f << ":";
   for (int c=0; c<COLS; c++)
      cout << setw(3) << a[f][c];
   cout << endl;
}
cout << endl;
```

```
   // Asignacion a los punteros de "b"

   for (int fb=0, fa=FILS-1; fa >=0 ; fa--, fb++)
      b[fb] = &(a[fa][0]);

   cout << "Recorr. convencional de \"b\": \n";
   for (int f=0; f<FILS; f++) {
      cout << "   Fila " << f << ":";
      for (int c=0; c<COLS; c++)
         cout << setw(3) << b[f][c];
      cout << endl;
   }
   cout << endl;
   return (0);
}
```

```
%vecpun

Recorr. convencional de "a":
   Fila 0:  1  2  3  4  5
   Fila 1:  6  7  8  9 10
   Fila 2: 11 12 13 14 15

Recorr. convencional de "b":
   Fila 0: 11 12 13 14 15
   Fila 1:  6  7  8  9 10
   Fila 2:  1  2  3  4  5
```

1. **Vector de punteros a carácter**.

```
char *meses1[] = {"!Erroneo!", "Enero", "Febrero", "Marzo"};
```



Ragged arrays, o vectores "a jirones".

2. **Matriz bidimensional de caracteres**.

```
char meses2[][10] = {"!Erroneo!", "Enero", "Febrero", "Marzo"};
```

meses2

	0	1	2	3	4	5	6	7	8	9
0	´¡´	'E'	´r´	´r´	´o´	´n´	´e´	´o´	´!´	´\0´
1	'E'	´n´	´e´	´r´	´o´	´\0´				
2	'F'	´e´	´b´	´r´	´e´	´r´	´o´	´\0´		
3	'M'	´a´	´r´	´z´	´o´	´\0´				

1. La ventaja de utilizar vectores de punteros frente a matrices dimensionadas es que **cada línea puede tener un número arbitrario de columnas, optimizándose el espacio en memoria**.

2. Un vector de punteros (convenientemente inicializado) se *comporta* como una matriz bidimensional, ya que se puede acceder a los elementos de esta estructura de datos mediante índices, como si de una matriz se tratara.

Diferencia clave:

En un vector de punteros las "filas" no tienen porqué estar almacenadas de forma secuencial en memoria.

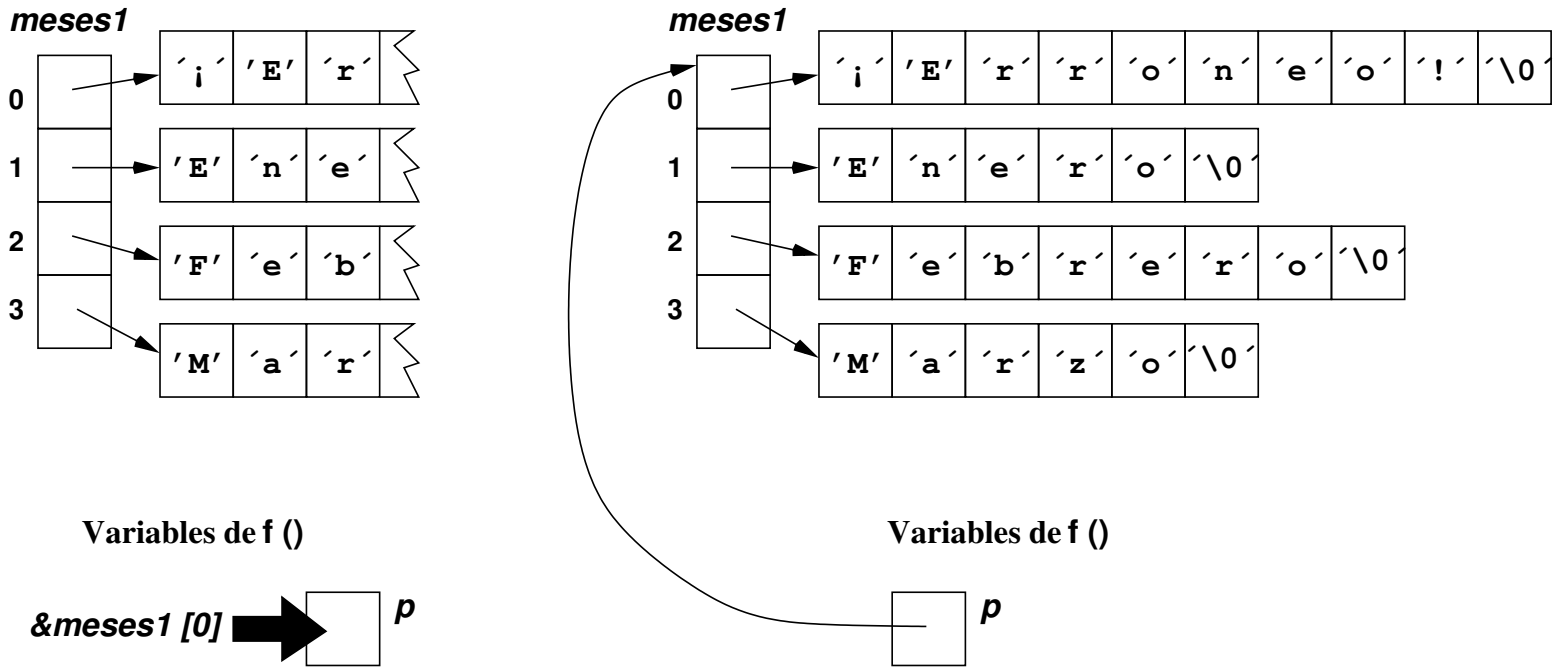
Paso de un vector de punteros a una función:

- Se pasa el nombre del vector.
- Se recibe con una de estas dos alternativas:

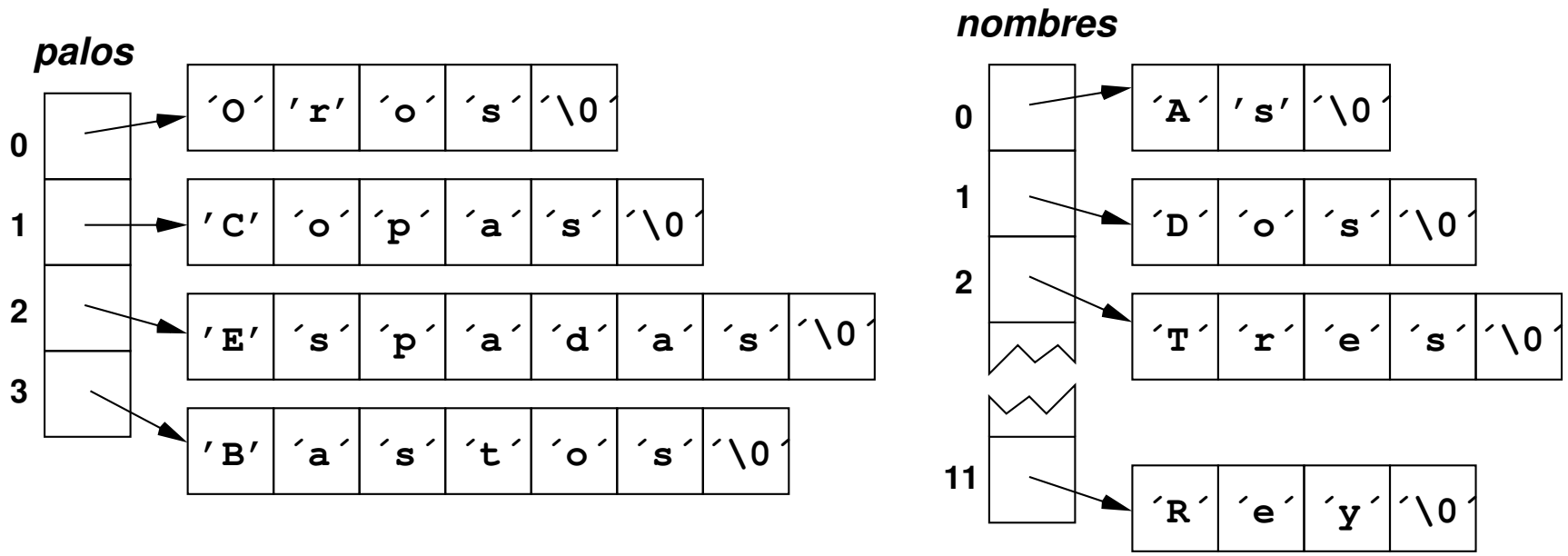
1. En un puntero a datos del tipo base del vector.
2. Un vector con un número indeterminado de casillas.

```
f1 (meses1);
......

void f1 (char **p)
{
   ......
}
```

```
f2 (meses1);
......

void f2 (char *p[])
{
   ......
}
```



Paso de un vector de punteros a una funcion



Vectores "a jirones" usados en el programa de la baraja

Estructura de datos básica:

```
int baraja [PALOS][CARTAS_PALO] = {0};
```

baraja

	0	1	2	3	4	5	6	7	8	9	10	11
0	29	25	2	30	17	32	6	26	7	48	13	31
1	5	43	42	11	41	12	24	28	20	36	35	34
2	45	15	44	3	1	39	19	40	18	47	23	14
3	16	21	8	10	22	37	9	46	4	27	38	33

Un ejemplo de la matriz `baraja`

```
  Cinco de Espadas        Tres de Oros        Cuatro de Espadas
  Nueve de Bastos           As de Copas        Siete de Oros
  Nueve de Oros           Tres de Bastos       Siete de Bastos
 Cuatro de Bastos        Cuatro de Copas        Seis de Copas
Caballo de Oros            Rey de Espadas       Dos de Espadas
     As de Bastos         Cinco de Oros        Nueve de Espadas
  Siete de Espadas        Nueve de Copas         Dos de Bastos
  Cinco de Bastos      Caballo de Espadas       Siete de Copas
    Dos de Oros           Ocho de Oros          Sota de Bastos
   Ocho de Copas            As de Oros        Cuatro de Oros
    Rey de Oros           Seis de Oros           Rey de Bastos
    Rey de Copas       Caballo de Copas         Sota de Copas
   Seis de Bastos      Caballo de Bastos        Seis de Espadas
   Ocho de Espadas       Cinco de Copas         Tres de Copas
    Dos de Copas          Tres de Espadas          As de Espadas
   Ocho de Bastos         Sota de Espadas        Sota de Oros
```

```
/*********************************************/
// Fichero: baraja.cpp
// Mezcla aleatoriamente un mazo de cartas.
/*********************************************/

#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>

using namespace std;

const int PALOS = 4;                  // Num. Palos
const int CARTAS_PALO = 12;           // Cartas/palo
const NUM_CARTAS = PALOS*CARTAS_PALO // Num. Cartas

const int COLS = 3; // Columnas para el resultado
```

```
void barajar (int baraja[PALOS][CARTAS_PALO]);
void mostrar (int baraja[PALOS][CARTAS_PALO], char *palos[PALOS],
              char *nombres[CARTAS_PALO]);

/*****************************************************************/

int main (void)
{

 // Declaracion e iniciacion de "nombres" y "palos"

char *palos[PALOS] = {"Oros", "Copas",  "Espadas", "Bastos"};
char *nombres[CARTAS_PALO] = {"As", "Dos", "Tres", "Cuatro",
                              "Cinco", "Seis", "Siete", "Ocho",
                              "Nueve", "Sota", "Caballo", "Rey"};
```

```
   // Decl. e inic. de "baraja": determina el orden de las cartas

   int baraja [PALOS][CARTAS_PALO] = {0};

   barajar (baraja);  // barajar el mazo
   mostrar (baraja, palos, nombres); // resultado

   return (0);
}

/*****************************************************************/
// Funcion: barajar
// Tarea:   Ordena las cartas aleatoriamente.
// Recibe:  int baraja[][CARTAS_PALO], matriz 2D de enteros que determina
//          el orden final. P.e. si baraja[0][0] (As de Oros) es 20, el As
//          de Oros sera la carta numero 20.
/*****************************************************************/
```

```
void barajar  (int baraja[][CARTAS_PALO])
{
   int carta, fila, col;
   time_t t;

  // Inicializa el generador con el reloj del sistema
   srand ((int) time(&t));

   // para todas las cartas del mazo

   for (carta=1; carta <= NUM_CARTAS; carta++) {

      // asigna un numero (orden) a cada carta si baraja[fila][col]==0

      do { col  = rand() % CARTAS_PALO; // columna
           fila = rand() % PALOS;       // fila
      } while (baraja[fila][col] != 0);
```

```
      baraja[fila][col] = carta;

      // numero de orden de la carta de palo "fila" y nombre "col".
   }
}

/*****************************************************************/
// Funcion:  mostrar
// Tarea:    Muestra el mazo (en COLS columnas)despues de barajar.
// Recibe:   int baraja[][CARTAS_PALO], matriz 2D que determina el orden.
//           char **palos, matriz de cadenas con los nombres de los palos.
//           char **nombres, matriz de cadenas con los nombres de las cartas.
/*****************************************************************/

void mostrar (int baraja[][CARTAS_PALO], char **palos, char **nombres)
{
   char c;           // puede ser '\n' o '\t'
```

```
   // para todas las cartas del mazo

   for (int carta=1; carta <= NUM_CARTAS; carta++)

      for (int fila=0; fila<PALOS; fila++)

         for (int col=0; col<CARTAS_PALO; col++)

            // Mostrar si en la casilla [fila][col] esta el numero buscado.

            if (baraja[fila][col] == carta) {
               c = (((carta % COLS == 0)||(carta == NUM_CARTAS)) ? '\n':'\t');
               cout << setw(8) << nombres[col] << " de " << setw(10)
                    << palos[fila] << " " << c;
            }
         cout << endl;
}
```

Una mejora:

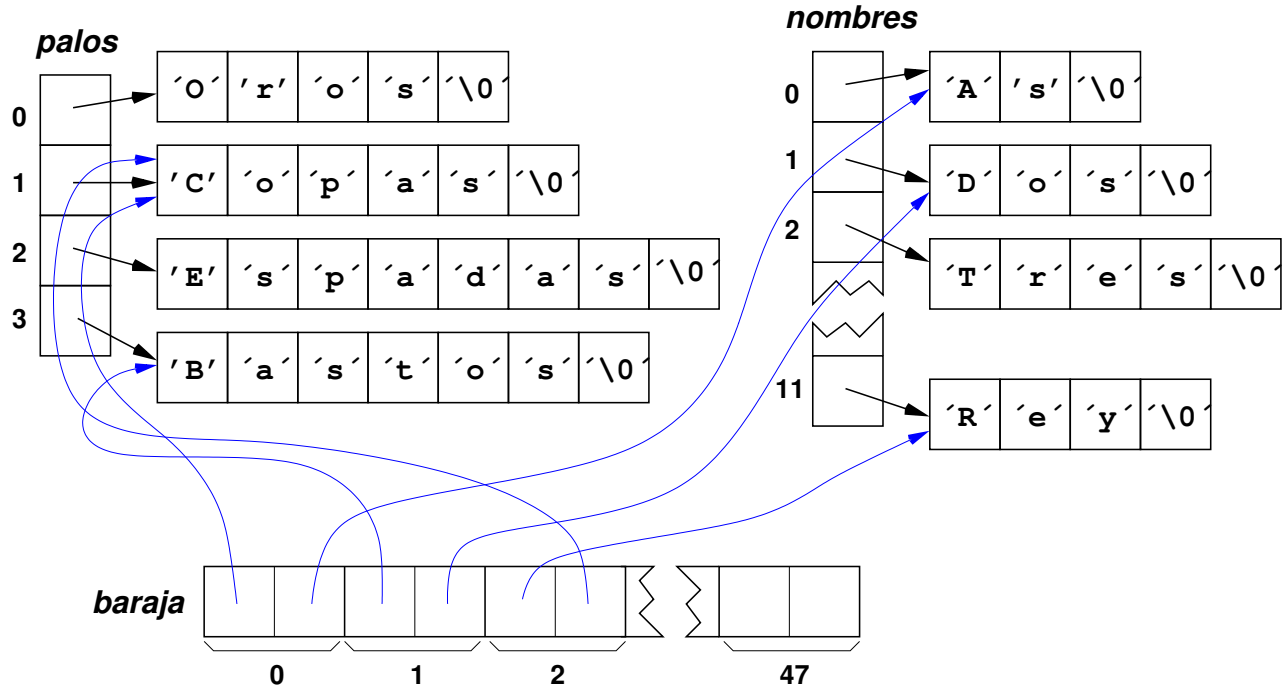
- Propuesta: representar cada carta por una estructura (`struct`) de forma que los datos sean:
  - el palo (`palo`)
  - el nombre de la carta (`nombre`)

  y los valores de cada campo sean cadenas de caracteres (punteros a las cadenas estáticas referenciadas por las matrices `palos` y `nombres`):

```
struct carta {

   char *palo;   // punt. a cadena de "palos"
   char *nombre; // punt. a cad.de "nombres"
};
```

- Ahora, una baraja es un vector de `NUM_CARTAS` estructuras de tipo `carta`:

```
carta baraja[NUM_CARTAS];
```



Estado de la memoria para una ordenación particular de cartas

- A la hora de barajar, procesamos las catras secuencialmente: desde la primera (*As de Oros*) a la última (*Rey de bastos*), generando un número aleatorio comprendido entre 0 y `NUM_CARTAS-1`.

- Antes de "incluir" la carta, ver si la posición ya está ocupada:

```
int control[NUM_CARTAS] = {0};
```

  Si `control[i] == 0`, libre,
  si `control[i] == 1`, ocupada.

```
/*************************************************************/
// Fichero: baraja2.c
// Mezcla aleatoriamente un mazo de cartas.
// Otra implementacion utilizando un vector de estructuras.
/*************************************************************/
#include <iostream>
#include <iomanip>
#include <cstdlib>
#include <ctime>
using namespace std;

const int PALOS = 4;                    // Palos
const int CARTAS_PALO = 12;             // Cartas/palo
const NUM_CARTAS = PALOS*CARTAS_PALO;// Num. Cartas
const int COLS = 3; // Num. columnas del resultado
```

```
struct carta { // Definicion del tipo de datos carta

 char *palo; // punt. a una cadena de "palos"
 char *nombre; // punt. a cadena de "nombres"

};

void barajar (carta baraja[NUM_CARTAS],
     char *palos[PALOS],
     char *nombres[CARTAS_PALO]);

void mostrar (carta baraja[NUM_CARTAS],
     char *palos[PALOS],
     char *nombres[CARTAS_PALO]);
```

```
int main (void)
{
   // Declaracion e iniciacion de "nombres" y "palos"
   char *palos[PALOS]={"Oros", "Copas", "Espadas", "Bastos"};
   char *nombres[CARTAS_PALO] =
         {"As", "Dos", "Tres", "Cuatro", "Cinco", "Seis",
          "Siete", "Ocho", "Nueve", "Sota", "Caballo", "Rey"};

   // Declaracion de "baraja": orden de cartas
   carta baraja[NUM_CARTAS];

   barajar (baraja, palos, nombres); // barajar
   mostrar (baraja, palos, nombres); // resultado
   return (0);
}
```

```
/*****************************************************************/
// Funcion:  barajar
// Tarea:    Ordena las cartas aleatoriamente.
// Recibe:   carta baraja[NUM_CARTAS], vector de estructuras
//           de tipo "carta" que determina el orden final.
//   char **palos, matriz de cadenas (palos).
//   char **nombres, matriz de cadenas (cartas).
/*****************************************************************/

void barajar (carta baraja[NUM_CARTAS],
              char **palos, char **nombres)
{
   int p, c, orden;
   int control[NUM_CARTAS] = {0};
   time_t t;
```

```
// Iniciar el generador con el reloj del sistema.
srand ((int) time(&t));

// Para todas las cartas del mazo en orden sec.:
// desde el As de Oros, hasta el Rey de Bastos.

for (p=0; p < PALOS; p++)  // palos

   for (c=0; c < CARTAS_PALO; c++) { // nombres
      // Asigna un numero a cada carta si no ha
      // sido procesada (control[orden] == 0)

      do {
         orden  = rand() % NUM_CARTAS;
      } while (control[orden] != 0);
```

```
            (baraja[orden]).palo   = palos[p];
            (baraja[orden]).nombre = nombres[c];
            control[orden] = 1;
        }
}

/*************************************************************/
// Funcion:  mostrar
// Tarea:    Muestra el mazo de carta barajado
// Recibe:   carta baraja[NUM_CARTAS], vector de estruct. de tipo
//           "carta" cuyo orden determina el orden.
//           char **palos, matriz de cad. (palos).
//           char **nombres, matriz de cad. (cartas).
/*************************************************************/
```

```
void mostrar (carta baraja[NUM_CARTAS],char **palos,char **nombres)
{

  // Para todas las cartas del mazo

  for (int n=1; n <= NUM_CARTAS; n++)

    cout << (baraja[n-1]).nombre << " de "
         << left << setw(10) << (baraja[n-1]).palo
         << (((n%COLS==0)||(n==NUM_CARTAS))?'\n':'\t');

}
```

0.18 Paso de args. en la línea de órdenes

- **Motivación**.

  Manejamos habitualmente programas a los que se les proporcionan los datos sobre los que operan desde la línea de órdenes (*prompt* del sistema) al mismo tiempo que se invoca su ejecución.

```
% echo hola
hola
% echo adios
adios
```

- **Implementación**.

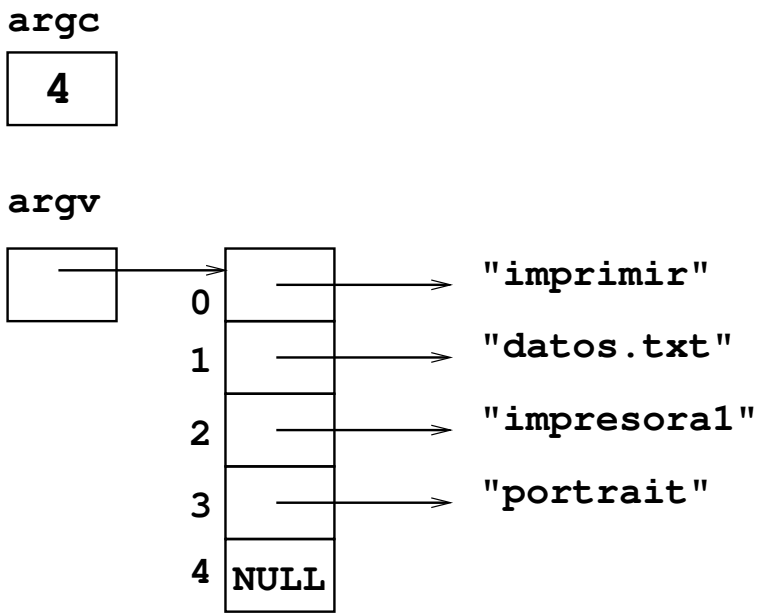
```
int main (int argc, char *argv[])
```

`int argc`	El número de argumentos suministrados, incluído el nombre del programa.
`char *argv[]`	Proporciona los argumentos en sí.

Ejemplos

```
% imprimir datos.txt impresora1 portrait
```

```
argc  = 4
argv [0] = "imprimir"
argv [1] = "datos.txt"
argv [2] = "impresora1"
argv [3] = "portrait"
argv [argc] = argv [4] = 0
```



```
/*******************************************/
// Programa: echo1.cpp
// Una implementacion de echo
/*******************************************/
#include <iostream>
using namespace std;

int main (int argc, char **argv)
{
   for (int i = 1; i < argc; i++)
      cout << argv[i] << ((i < argc-1) ? " ": "");

   cout << endl;
   return (0);
}
```

```
/*********************************************/
// Programa: echo2.cpp
// Otra implementacion de echo
/*********************************************/
#include <iostream>
using namespace std;

int main (int argc, char **argv)
{
   while (--argc > 0)
      cout << *++argv << ((argc > 1) ? " ": "");

   cout << endl;
   return (0);
}
```

0.19 Funciones *inline*

Motivación

Cuando una función es llamada múltiples veces, se debe acceder cada una de las veces a donde está el código de definición (guardar estado, salto, copia en la pila, etc.). Sería más eficiente, si no se realizara la llamada, sino que se ejecutara el código de la función.

- En C el proceso de que el compilador sitúe el código donde se realiza la llamada, es mediante macros definidas con `#define`
- En C++ también está permitido el uso de macros, pero no se recomienda.

C++ proporciona las funciones *inline*, que permite ubicar el código de la función en el lugar de la llamada (sustitución), si el compilador lo considera oportuno.

Ventaja de las funciones *inline*: La etapa de ejecución es más rápida, aunque aumenta el volumen del ejecutable.

Formato: preceder la declaración con `inline`.

Cuando las funciones miembro de una clase son definidas dentro de la propia clase, no necesitan que se les antepongan la palabra *inline*, implícitamente son consideradas *inline*.

Sólo deberían definirse funciones *inline*, cuando la definición contenga muy poco código.

```
#include <iostream>

const TIPO = 16;
inline float iva (float a) {return ((TIPO*a)/100);}

int void()
{
  float alquila1 = 23,67;
  float alquila2 = 65,78;
  float alquila3 = 34,52;
  .....
  cout << iva(alquila1) << " " << iva(alquila2)
       << " " << iva(alquila3) << " " << endl;
  ....
}
```

0.20 Parámetros por defecto.

En C++ se permite indicar en el prototipo de una función, valores por defecto de algunos o todos los parámetros formales, que se asumirán en la llamada a la función como argumentos por defecto.

Se usan cuando se realizan múltiples llamadas a una función, casi siempre con los mismos argumentos.

En el prototipo de la función, los valores de los parámetros formales por defecto deben aparecer a la derecha de los que no son por defecto.

Los parámetros por defecto en la definición de la función, aparecerán como parámetros normales.

El valor por defecto de una declaración puede ser una constante, una constante global, una variable global e incluso una llamada a una función.

```
void fun(int a, long b = 2, float c) {...} // INCORRECTO
```

```
void fun(int a, long b = 2, float c = 5) {...} // CORECTO
```

Cuando se realice una llamada a una función con parámetros por defecto, todos los argumentos por defecto más a la derecha que no aparezcan en la llamada, serán los especificados en la cabecera del prototipo.

```
double potencia (long base, int exponente=2)
{
  double resultado;
  if (exponente == 2) resultado = base * base;
  else
     for(int i=1, resultado=base; i < exponente; i++)
        resultado *= base;
  return resultado;
}
int main()
{
  cout << "potencia(5) = " <<
  potencia(5) << endl;
  cout << "potencia(4, 3)= " << potencia(4,3) << endl;
}
```

0.21 Sobrecarga de funciones

C++ permite el uso de funciones sobrecargadas, que son aquellas que realizan prácticamente la misma tarea, tienen el mismo nombre de función, y se diferencian en los tipos y/ó número y/ó orden de los parámetros formales.

Cuando se realiza una llamada a una función sobrecargada, se activarán en el siguiente orden:

- Coincidan exactamente los argumentos, con el número y tipo de sus parámetros formales
- Solo se tengan que realizar promociones de tipos (de menor a mayor)

- Tenga que realizar conversiones de tipos propios del compilador
- Necesite conversiones de tipos definidas por el usuario.

Si no ocurre ninguna de las situaciones anteriores, se obtendrá un error.

La **firma** de una función es el conjunto de su nombre y la lista ordenada de los tipos de parámetros formales (obviando el modificador `const` y el símbolo de referencia `&`)

No se pueden sobrecargar funciones que solo se diferencien en el tipo devuelto, y las firmas no pueden coincidir, por lo que no pueder ser considerada como una diferencia aquella que esté basada en el modificador `const` o en el símbolo de referencia `&`.

Un caso típico de sobrecarga de funciones, son algunas funciones matemáticas que se definen para distintos tipos de datos: `int`, `float`, etc.