– Typeset by GMNI & FoilTEX **–**

Lenguajes de programación en ingeniería:
PROGRAMACIÓN EN LENGUAJE C y FORTRAN

J. París, F. Navarrina & GMNI

GMNI — GRUPO DE MÉTODOS NUMÉRICOS EN INGENIERÍA

Escuela Técnica Superior de Ingenieros de Caminos, Canales y Puertos
Universidad de A Coruña, España

e-mail: {jose.paris,fermin.navarrina}@udc.es
página web: http://caminos.udc.es/gmni

ÍNDICE

Estructura de un programa (I)

1) Líneas de programa:

- **Fortran**
  - **Formato fijo**
    - Columna 1 → (c,d,!) Línea de comentario
    - Columnas 2 a 5 → Etiqueta numérica de línea
    - Columna 6 → Continuación de línea
    - Columnas 7 a 72 → Espacio para instrucciones
    - Columnas 73 a 80 → Espacio para comentarios
  - No hay formato libre (salvo en versiones modernas de Fortran)

- **C**
  - El formato es libre.
  - Las instrucciones se gestionan y agrupan mediante “;” y “{ }”

Estructura de un programa (II)

2) Comentarios:

Fortran

- `c`, `d` → En la primera columna comenta toda la línea
  La letra `d` se reserva para una opción de compilación
- `!` → Comenta la línea desde la posición en la que se encuentra
  Es lo habitual hoy en día
- cols. 73 a 80 → Son siempre comentarios.

C

- `/*...*/` → Comenta en bloque una o varias líneas
- `//...` → Comenta la línea desde la posición en la que se encuentra
  Estándar en versiones modernas de C y en C++

Estructura de un programa (III)

3) Preprocesador (directivas de compilación) (Fase previa al compilado):

Fortran

- `d` → Se desactivan comentarios con `D`. En desuso.
- `include 'filename'` → Incorpora ficheros de texto externos
  Ej. Ficheros de encabezamiento.
- `parameter(SIMBOLO=valor)` → Remplaza antes de compilar
  el texto `SIMBOLO` por `valor`.

Estructura de un programa (IV)

3) Preprocesador (directivas de compilación) (Fase previa al compilado):

C

`#include <stdio.h>`	→ Incorpora la librería del sistema `stdio.h`
`#include "lib.h"`	→ Incorpora la librería propia `lib` existente en la carpeta actual
`#define SIMBOLO`	→ Define `SIMBOLO` como un parámetro
`#define SIMBOLO valor`	→ Remplaza antes de compilar el texto `SIMBOLO` por `valor`

`#if`
`#elif`
`#endif`
	Permite incorporar condiciones al preprocesador → (Se utiliza, por ejemplo, para actuar en función del sistema operativo existente)

`#define FUNC_SIMB(VAR) F(VAR)`	→ Remplaza `FUNC_SIMB(VAR)` por la función F y a su vez remplaza el texto `VAR` en la expresión de la función F
`#define ...\` `.....`	→ Indica que la definición continúa en la siguiente línea
`#ifdef SIMBOLO` `...`	→ Si está definido `SIMBOLO` entonces `...`

Estructura de un programa (V)

3) Preprocesador (directivas de compilación) (Fase previa al compilado):

Ejemplos de posibles efectos secundarios indeseados:

```
#define COSA(X) (X + X)
        ⋮
j = COSA( i++ );
```

$\longrightarrow \quad j = (i++ \; + \; i++) \neq$

```
j = COSA(i);
i++;
```

```
#define COSA(X) (X * 2.)
        ⋮
u = COSA( a + b );
```

$\longrightarrow \quad u = (a + b * 2.) \left[\neq \quad u = (a + b) * 2. \right]$

```
v = 1./COSA( a );
```

$\longrightarrow \quad v = (1./a * 2.) \left[\neq \quad u = 1./(a * 2.) \right]$

Estructura de un programa (VI)

4) Programa principal:

Fortran

```
!234567
      program nombre      ! Si se omite se denomina por defecto: main
      ...
      end
```

C

```
int main(void)      /* Devuelve un entero de control y no recibe datos */
{
   ...
}
```

ó

C[1]

```
int main(argc, **argv)      /* Devuelve un entero de control
                                  y recibe dos argumentos*/
{
   ...
}
```

(1) Esta opción permite incorporar datos al programa que se escriben directamente en la línea de comandos cuando éste se ejecuta. (Ej. `gfortran programa.f -o programa.exe`)

Estructura de un programa (VII)

5) Grupos de instrucciones:

Fortran $\rightarrow$ No son posibles.

C $\rightarrow$ Sí que son posibles.

- Se definen mediante `{ ... }` que encierran el conjunto de instrucciones.
- Admiten definiciones de variables locales, propias y exclusivas del grupo.

Estructura de un programa (VIII)

6) Legibilidad del código fuente:

Fortran

- Las letras mayúsculas y minúsculas son equivalentes.
- El uso de minúsculas es recomendado (en general).
- Se recomienda usar mayúsculas para definir símbolos.
  (Ej. `PARAMETER (PI=3.1415926535)`)
- Se recomienda usar la primera letra mayúscula para nombres de subrutinas.
  (Ej. `subroutine Producto(...)`)
- El sangrado de las líneas hay que hacerlo utilizando espacios.
- Por motivos de formato reducido no se dejan espacios entre las operaciones.

C

- Las letras mayúsculas y minúsculas son diferentes.
  -Salvo en el nombre de funciones (por compatibilidad con otros lenguajes)
  -No se recomienda utilizar esta distinción en la práctica
- El uso de minúsculas es recomendado (en general).
- Se recomienda usar mayúsculas para definir símbolos.
  (Ej. `#define PI 3.1415926535`)
- Se recomienda usar la primera letra mayúscula para nombres de funciones.
  (Ej. `int Producto(...){...}`)
- El sangrado de las líneas se realiza mediante tabuladores.
- Es práctica habitual dejar espacios entre las operaciones (Ej. `i = j + k;`).

Variables y constantes (I)

Variables:

1) Definición de variable:

Es un nombre simbólico que identifica:

- El nombre de la variable representa la dirección de memoria donde está almacenada.
- Y el espacio de almacenamiento (en función del tipo de variable), esto es: el valor

2) Nombres de variables:

Fortran

- No pueden empezar por un dígito numérico `(0-9)`
- Podemos utilizar los caracteres `a-z` $\equiv$ `A-Z, 0-9, ...,$`
- Se consideran:
  - -Sólo los 31 primeros caracteres para nombres “internos”
  - -Sólo los 6 primeros caracteres para nombres “externos”
- . . . dependiendo del compilador y de las opciones de compilación
- No se admiten:
  - -nombres de instrucciones (`do`, `if`, etc.)
  - -nombres de funciones intrínsecas (`sin`, `max`, `int`, etc.)
- Deben ser mnemónicos
- ¡OJO!: `0` $\neq$ `O` y `1` $\neq \ell$

C

- Se mantienen los mismo criterios que para Fortran
- Hay que tener en cuenta que, en general, `a-z`$\neq$`A-Z`
  - -Excepto en nombres externos de funciones (por compatibilidad)

Variables y constantes (II)

2) Nombres de variables:

Nombres reservados:

C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

asm
fortran
} → dependiendo de la implementación

Los nombres subrayados se incorporaron en el nuevo ANSI standard.

Variables y constantes (III)

3) Tipos de variables (Básicos):

Fortran

- `integer*1` (o byte), `integer*2`, `integer*4`, `integer*8`
  `integer` (Lo más habitual) (lógico CPUs 64 bits)
- `real*4`, `real*8`, `real*16`
  real, double precision, quadruple precision
- `complex*8`, `complex*16`, `complex*32`
  complex,
- `logical*1`, `logical*2`, `logical*4`, `logical*8`
  `logical` ⟶ Esto es lo más recomendable en la práctica
- `character *(n) texto` → Crea una variable alfanumérica de n caracteres

Variables y constantes (IV)

3) Tipos de variables (Básicos):

- C
  - `unsigned`
    - `char` → 1 byte
    - `short (int)` → variable entera $\geq$ `char` (Normalmente 2 bytes)
    - `int` → variable entera $\geq$ `short` (Normalmente 4 bytes)
    - `long (int)` → variable entera $\geq$ `int` (Normalmente 4 bytes)
    - `long long` → variable entera $\geq$ `long` (Y nada más a priori)
  - `char` → 1 byte
  - `short (int)` → variable entera $\geq$ `char` (Normalmente 2 bytes)
  - `int` → variable entera $\geq$ `short` (Normalmente 4 bytes)
  - `long (int)` → variable entera $\geq$ `int` (Normalmente 4 bytes)
  - `long long` → variable entera $\geq$ `long` (Y nada más a priori)
  - `float` → real en simple precisión (pero número de bytes desconocido)
  - `double` → real en doble precisión (pero número de bytes desconocido)
  - `long double` → real en cuádruple precisión (número de bytes desconocido)

Variables y constantes (V)

3) Tipos de variables: Variables locales (automáticas) y externas

V. Locales

- Son variables internas a un módulo
- El nombre (dirección de memoria) es desconocido para otros módulos
- El espacio de almacenamiento tampoco.
  - -Información permanente en prog. principal $\rightarrow$ valor garantizado
  - -Información temporal en otros módulos $\rightarrow$ valor no garantizado en llamadas sucesivas

V. Externas

- Son variables comunes a varios módulos
- El nombre (dirección de memoria) es conocido en varios módulos
- El espacio de almacenamiento es:
  - -Permanente en varios módulos $\rightarrow$ valor garantizado

Variables y constantes (VI)

3) Tipos de variables: Variables locales (automáticas) y externas

ForTran

Todas las variables son locales a módulos, excepto

- Argumentos de subrutinas y funciones (se envían por referencia, no por valor)
- Bloques COMMON

```
common /nombre/ vble_1, vble_2, ...
```

C

Todas las variables son locales a módulos o grupos (`{ ... }`) , excepto

- Las que se declaran como `extern`
  - Las declaradas antes de: `int main(void)`
  - Las declaradas dentro de cada función como `extern` (*)

```
extern variable_externa
```

(*) Si las funciones están todas en el mismo archivo no es necesario declararlas en todas.

Variables y constantes (VII)

Ej. Fortran

```
!234567
      implicit real*8(a-h,o-z)

      n=2
      x=5.
      call calc(x,n,y)
      print*,y
      call exit(0)
      end
!
!-------------------------
!
      subroutine calc(a,i,b)
      implicit real*8(a-h,o-z)

      z=a+1.
      b=z**i
      return
      end
```

```
!234567
      implicit real*8(a-h,o-z)
      common /exponente/ n
      n=2
      x=5.
      call calc(x,y)
      print*,y
      call exit(0)
      end
!
!-------------------------
!
      subroutine calc(a,b)
      implicit real*8(a-h,o-z)
      common /exponente/ i
      z=a+1.
      b=z**i
      return
      end
```

Variables y constantes (VIII)

Ej. Lenguaje C

```
void main(void)
{
     void calc(double, int, double *)     /* Prototipo de función */
     int n;                               /* Declaración de variable n */
     double x, y;                         /* Declaración de variables x e y */
     n=2;
     x=5.;
     calc(x,n,&y);                        /* Llamada a la función calc */
     print("%f",y);                       /* Impresión por pantalla del resultado */
     exit(0);
}
void calc(double a, int i, double *pb)
{
     double z;
     z = a + 1.;
     *pb = pow(z,i);
}
```

- El prototipo define el tipo de función y de las variables que se transmiten.
- A priori, las variables que se modifican en la subrutina se envían con `&` y se reciben con $*$.

Variables y constantes (IX)

4) Constantes:

Fortran

- Enteras:

  $\pm$ [nº en base decimal] $\rightarrow$ cifra entre 0 y 9

  Se guardan en el tipo entero por defecto

  `integer*2`
  `integer*4`
  `integer*8`
  } $\rightarrow$ Dependiendo de la opción de compilación

- Reales:

$\pm$123.4	$\rightarrow$	REAL (Normalmente `real*4`)
$\pm$.1234e+3	$\rightarrow$	REAL (Normalmente `real*4`)
$\pm$.1234d+3	$\rightarrow$	DOUBLE PRECISION (Normalmente `real*8`)
$\pm$.1234q+3	$\rightarrow$	QUADRUPLE PRECISION (Normalmente `real*16`)

  Depende de las opciones de compilación

  Se guardan en el tipo correspondiente o por defecto (real*4, real*8, real*16)

Variables y constantes (X)

4) Constantes:

Fortran

- Alfanuméricas:

No son variables como tales. Son Descriptores.

Contienen internamente → { La dirección de memoria del inicio del “string”
La longitud del “string” (número de caracteres)

```
'hello, world'
12Hhello, world
```

} → hello, world

12 caract.

↑

Formato Hollerith

- Lógicas:

```
.true.
.false.
```

Variables y constantes (XI)

4) Constantes:

Lenguaje C

- Enteras: (ver el archivo `limits.h en las librerías del compilador`)

Se almacenan en forma de (`unsigned` / `signed`) { `char` / `short` / `int` / `long` / `long long` }

`char`

`'x'`	→	código ASCII del carácter x	`'\r'`	→	carriage return
`'\0'`	→	null	`'\f'`	→	form feed
`'\n'`	→	newline	`'\a'`	→	bell (beep)
`'\t'`	→	horizontal tab	`'\\'`	→	backslash
`'\v'`	→	vertical tab	`'\?'`	→	question mark
`'\b'`	→	backspace	`'\''`	→	single quote
`'%%'`	→	símbolo %	`'\"'`	→	double quote

`'\o'` / `'\oo'` / `'\ooo'` → octal { o → 1 cifra octal; oo → 2 cifras octales; ooo → 3 cifras octales }

`'\xh'` / `'\xhh'` → hexadecimal { h → 1 cifra hexadec.; hh → 2 cifras hexadec. }

Se guardan en { unsigned char / signed char } según la implementación

Variables y constantes (XII)

4) Constantes:

Lenguaje C

`short, int, long, long long`

$\pm$ [número en base decimal] $\rightsquigarrow$ cifras (0-9), primera $\neq 0$, salvo para el número cero.

$\pm$ 0[número en base octal] $\rightsquigarrow$ cifras (0-7), la primera cifra es un 0.

$\pm$ 0x[número en hexadecimal] $\rightsquigarrow$ cifras (0-9), letras (a-f) $\equiv$ (A-F)

Se guardan en (`unsigned` / `signed`) { `char` / `short` / `int` / `long` / `long long` }

↓ según el signo o lo que se indique

↓ lo mínimo necesario o lo que se indique

constante + { u / U } → unsigned Ej. `347U`

constante + { l / L } → long Ej. `2598L`

} Se pueden combinar (Ej. `0xFUL` $\equiv$ 15)

NOTA: Las expresiones con constantes enteras se evalúan al compilar (no al ejecutar)

Variables y constantes (XIII)

4) Constantes:

Lenguaje C

- Reales: (ver el archivo `float.h en las librerías del compilador`)

`float, double, long double`

$$\left\{ \begin{array}{c} \pm 123.4\text{f} \\ \pm 123.4\text{F} \end{array} \right\}, \left\{ \begin{array}{c} \pm .1234\text{e+3f} \\ \pm .1234\text{e+3F} \end{array} \right\} \rightsquigarrow \texttt{float}$$

$\pm 123.4, \pm .1234\text{e+3} \quad \rightsquigarrow \quad$ `double` (opción por defecto)

$$\left\{ \begin{array}{c} \pm 123.4\ell \\ \pm 123.4\text{L} \end{array} \right\}, \left\{ \begin{array}{c} \pm .1234\text{e+3}\ell \\ \pm .1234\text{e+3L} \end{array} \right\} \rightsquigarrow \texttt{long double}$$

Variables y constantes (XIV)

4) Constantes:

Lenguaje C

- Alfanuméricas: (Strings) Se tratan como vectores de caracteres.

$$\begin{cases} \texttt{"hello, world"} \Leftrightarrow \underbrace{\texttt{"hello,"}\ \texttt{" world"}}_{\text{Se concatenan}} \rightsquigarrow \text{hello, world} \\ \texttt{""} \Leftrightarrow \text{String vacío} \end{cases}$$

  - Consisten en un vector de caracteres tipo `char con un '\0' (null) al final.`
  - `Se definen por el vector donde se almacena y su longitud finaliza en el primer '\0'.`
  - `El '\0' final se establece por convenio.`

Variables y constantes (XV)

4) Constantes:

Lenguaje C

- Enum: (Permite crear conjuntos de variables con valores constantes asignados)

```
enum boolean {NO,YES};
```
$\longrightarrow$ $\left\{ \begin{array}{l} \text{NO} \longleftrightarrow 0 \\ \text{YES} \longleftrightarrow 1 \end{array} \right.$

```
enum escapes {BEL='\a', BACKSPACE='\b', ..., RETURN='\r'};
```

```
enum meses {ENE=1, FEB, MAR, ABR,... DIC};
```
$\longrightarrow$ $\left\{ \begin{array}{l} \text{ENE} = 1 \\ \vdots \\ \text{DIC} = 12 \end{array} \right.$

Los valores se guardan en variables tipo `int`. Forma de uso:

```
enum boolean {NO, YES};
enum boolean siono, aceptado;
⋮
siono = NO; aceptado = YES;
```

También:

```
enum boolean {NO,YES} siono, aceptado;
enum {NO,YES} siono, aceptado;
```

Variables y constantes (XVI)

5) Cambios de tipos:

Fortran: Se realizan mediante funciones

Funciones:
- `int(variable)`
- `real(variable)`
- `dble(variable)`
- `float(variable)` *[en desuso]*
- `dfloat(variable)` *[en desuso]*

→ vble.tipo2 = f(vble.tipo1)

Ej. `i=int(x)`
Ej. `z=real(j)`

Lenguaje C: Se realiza mediante un operador de conversión (“cast operator”))

Casts:

(tipo) variable /* Cambia `variable` al nuevo tipo*/
/* Sólo se aplica al argumento inmediato */

```
Ej.   i = (int) x;   // Convierte x en entero
Ej.   z = (float) j;   // Convierte j en real
```

Variables y constantes (XVII)

6) Inicialización de variables:

Fortran →

No es posible salvo con la instrucción `data`

```
data vble1,vble2,vble3 /valor1, valor2, valor3/
```

Ej.
```
data m,n,x,y /10,20,2.5,2.5/
   data m/10/, n/20/, x,y /2*2.5/
   real v(100)
   data v/100*0.0/      ! 100 componentes con valor 0.0
```

Las variables globales se inicializan una vez y al principio

Las variables locales, ¿se inicializan cada vez? → SI

Variables y constantes (XVIII)

6) Inicialización de variables:

C →

Se pueden inicializar directamente al declararlas:

```
int i;          int i = 3;
int i, j;       int i = 3, j = 4;
float pi;       float pi = 3.1415926535;
char letra;     char letra = 'x'
```

Se inicializan:
- Una sóla vez las permanentes o globales
- Cada vez en cada módulo las variables locales

Si se definen como constantes se inicializan y no pueden ser modificadas a posteriori

```
const int i = 3;
```

Se pueden inicializar también vectores al declararlos:

```
int v[5] = { 1, 2, 3, 4, 5 };
char texto[6] = {'L', 'u', 'n', 'e', 's', '\0'};
char texto[6] = "Lunes";
```

Variables y constantes (XIX)

7) Asignación de valores a variables:

Fortran →
- Asignación directa a variables. Ej. `variable=valor`
- En vectores, componente a componente. Ej. `vector(i)=valor`

C →

Asignación directa a variables.

```
i = 3;
i = j = k = 6;   ⟷   { k = 6;
                       j = k;
                       i = j; }
```

En vectores:

```
int v[5];
v = { 1, 2, 3, 4, 5 };          ⟶ No es correcto
int v[5] = { 1, 2, 3, 4, 5 };   ⟶ Es correcto
char texto[6];
texto[6] = "Lunes";             ⟶ No es correcto
char texto[6] = "Lunes";        ⟶ Es correcto
```

OJO: Hay que reservar un caracter adicional para el `'\0'`

```
float x, z;
int y, i, j, k;
x = y = z = 3.2;   // ¡Ojo a las conversiones de tipo automáticas!
i = j++ = k = 8;   // ¿Estamos seguros del resultado?
```

Operadores (I)

Operadores

	Fortran	C
Aritméticos	Sí	Sí
Relacionales	Sí	Sí
Lógicos	Sí	Sí
Incrementales	No	Sí
Bitwise logical	No	Sí
Otros	Concatenación	Ternario

Operadores (II)

1) Operadores aritméticos:

Fortran:

- { Unarios: `-` — Cambio de signo. Afecta a una variable.
  Binarios: `+, -, *, /` — Operaciones elementales. Afectan a 2 variables.

- Se aplican a `integer, real, complex`

- Prioridad { 1) − (unario)
  2) `*, /`
  3) `+, -` } → Puede alterarse con paréntesis.

  Ej. `a * b + c / -d` ⟷ `(a * b) + ( c / (-d))`

◇ ¡OJO! El compilador puede tomar decisiones. No hay que fiarse.

Si se quiere forzar un resultado es mejor utilizar variables intermedias

Ej. ¿`( a + b ) + c = a + ( b + c )`? si { a = -1.
b = +1.
c = 10^{-24}

Operadores (III)

1) Operadores aritméticos:

C:

- Son los mismos que en Fortran y además:
  - ▷ División modular (`%`): Función resto entero de la división de enteros
  - ▷ Abreviaturas: operadores abreviados
- { Unarios: `-` Cambio de signo. Afecta a una variable.
  Binarios: `+, -, *, /, %` Afectan a dos variables.
- Se aplican a: { `char, short, int, long, long long`
  `float, double, long double` (excepto la división modular `%`)
- Prioridad { 1) - (unario)
  2) `*, /, %`
  3) `+, -` } → Puede alterarse con paréntesis.

Ej. `a * b + c / -d` ⟷ `(a * b) + ( c / (-d))`

◇ ¡OJO! El compilador puede tomar decisiones. No hay que fiarse.

Si se quiere forzar un resultado es mejor utilizar variables intermedias

Ej. ¿`( a + b ) + c = a + ( b + c )`? si { a = -1.
b = +1.
c = 10^{-24}

Operadores (IV)

1) Operadores aritméticos:

C:

- Abreviaturas:

  x op= expresión $\longleftrightarrow$ x = x op (expresión)

  siendo op una operación $\rightarrow$ op $\in$ {`+, -, *, /, %`} (también {`<<, >>, &, ^, |`})

  ¡OJO ! $\rightarrow$ $\begin{cases} \text{Procurar no mezclar variables de distinto tipo} \\ \text{Forzar un cambio adecuado de tipos usando reglas de promoción (casts)} \end{cases}$

  Ej. $\begin{cases} \texttt{xmod += v[i] * v[i];} \longleftrightarrow \\ \texttt{v[i] /= xmod;} \end{cases}$

Operadores (V)

2) Operadores relacionales:

Fortran:

- `.gt.`, `.ge.`, `.lt.`, `.le.` → Mayor, mayor o igual que, menor, menor o igual que.
  `.eq.`, `.ne.` → Igual, no igual.
- Sólo se comparan escalares.
  Si los escalares son de distintos tipos se promueven al tipo más complejo, pero no es recomendable.
- Ej.
```
if (i.eq.1)

if (x.gt.5)
```
  ! Si `x` es `real*8` → `5` se convierte a `real*8`
- Tienen menos prioridad que las operaciones aritméticas:

```
if (x+y.gt.x*y) ⟷ if ((x+y).gt.(x*y))
```

De todos modos es mejor usar paréntesis para evitar ambigüedades.

Operadores (VI)

2) Operadores relacionales:

C:

- Estos operadores son en realidad numéricos:

  $\left\{ \begin{array}{l} \texttt{>, >=, <, <=} \\ \texttt{==, !=} \end{array} \right.$ $\downarrow$ Prioridad creciente.

- `( x < y )` vale $\left\{ \begin{array}{l} \text{0 si no es verdad (0} \equiv \text{ FALSE)} \\ \text{1 si es verdad (1} \equiv \text{ TRUE)} \end{array} \right.$

- En general $\left\{ \begin{array}{l} \text{0} \equiv \text{ FALSE} \\ \text{1} \equiv \text{ TRUE (cualquier valor no nulo indica TRUE)} \end{array} \right.$

  ¡OJO! No confudir $\left\{ \begin{array}{ll} \texttt{x = y} & \rightarrow \text{ Asigna el valor de } \texttt{y} \text{ en } \texttt{x} \\ \text{con} & \\ \texttt{x == y} & \rightarrow \text{ Devuelve el valor 0 si } \texttt{x} \text{ no coincide con } \texttt{y} \\ & \text{y el valor 1 si } \texttt{x} \text{ coincide con } \texttt{y} \end{array} \right.$

- Tienen más prioridad que las aritméticas (al contrario que en Fortran)

  Es recomendable utilizar siempre paréntesis para evitar conflictos.

  Ej. `a + b != c;` $\Longleftrightarrow$ `a + ( b != c );` $\rightarrow \left\{ \begin{array}{ll} \text{Vale } \texttt{a} & \text{si } \texttt{b} \text{ es igual a } \texttt{c} \\ \text{Vale } \texttt{a+1} & \text{si } \texttt{b} \text{ no es igual a } \texttt{c} \end{array} \right.$

Operadores (VII)

3) Operadores lógicos:

Fortran:

- `.and.`, `.or.` $\longrightarrow$ binario
  `.not.` $\longrightarrow$ unario
  `.eqv.`, `.neqv.`
- Tablas de verdad:

`(a).eqv.(b)` es `.true.` $\leftrightarrow$ { `a` y `b` son `.true.` / `a` y `b` son `.false.` }

`.neqv.` es equivalente a `.not..eqv.`

.EQV.	`a .true.`	`a .false.`
`b .true.`	`.true.`	`.false.`
`b .false.`	`.false.`	`.true.`

.NEQV.	`a .true.`	`a .false.`
`b .true.`	`.false.`	`.true.`
`b .false.`	`.true.`	`.false.`

Operadores (VIII)

3) Operadores lógicos:

Fortran:

- Prioridad de operadores $\rightarrow$ $\left\{ \begin{array}{ll} \texttt{.not.} & + \\ \texttt{.and.} & \uparrow \\ \texttt{.or.} & \downarrow \\ \texttt{.eqv.}—\texttt{.neqv.} & - \end{array} \right.$

- En caso de duda, se recomienda utilizar paréntesis

  Ej. ¿Qué quiere decir `(a.or.b.and.c)` ó `(a.or.(b.and.c))`?

Operadores (IX)

3) Operadores lógicos:

C:

- En realidad son operadores numéricos

  Binarios: `&&`, `||` (and y or)
  Unarios: `!` (negación)

  Prioridad → `!` (+) `&&` (↕) `||` (−)

  Luego `! a && b` ⟷ `( ! a ) && b`

  Se recomienda utilizar paréntesis para evitar confusiones

  Duda: ¿`if ( ! valido )...` ó `if ( valido == 0 )...`?

- Ej.: para ver si el año (year) es bisiesto:

```
if ( ( (year % 4) == 0 && (year % 100) != 0 ) || (year % 400) == 0)
```

ó

```
if ( ( ! (year % 4) && ( year % 100 ) ) || ! ( year % 400 ) )
```

Operadores (X)

4) Operadores Incrementales:

C:

`++, --` $\longrightarrow$ por $\begin{cases} \text{delante} & \rightarrow \text{El incremento es anterior a las operaciones} \\ \text{detrás} & \rightarrow \text{El incremento es posterior a las operaciones} \end{cases}$

¡OJO! `x++ = --y + z++;` $\longrightarrow$

```
y = y - 1;
x = y + z;
x = x + 1;
z = z + 1;
```

- Sólo puede aplicarse a variables.
- No puede aplicarse a expresiones.

```
z = ( x + y )++; // No es una expresión válida.
```

En caso de duda, evitar las confusiones mediante paréntesis.

Ej. ¿Qué significan?

```
a[i] = i++;
a[i] = ++i;
x = power(++n,n);
```

Operadores (XI)

5) Operadores *Bitwise Logical*:

C:

`&`	`bitwise and`
`|`	`bitwise or`
$\wedge$	`bitwise exclusive or`
`<<`	`left shift` (Desplaza los bits hacia la izqda. las posiciones indicadas)
`>>`	`right shift` (Desplaza los bits hacia la dcha. las posiciones indicadas)
$\sim$	complemento a uno (unario)

Ej.: `c = n & 0177;` $\rightarrow$ pone a cero todo menos los últimos 7 bits de n, que no se modifican

$$\overset{\downarrow}{(01111111)_2}$$

$$\begin{array}{rr} & (01111111)_2 \\ \& & (10101001)_2 \\ \hline & (00101001)_2 \end{array} \rightarrow \begin{cases} 0 \text{ “\&” } 0 \rightarrow 0 \\ 0 \text{ “\&” } 1 \rightarrow 0 \\ 1 \text{ “\&” } 0 \rightarrow 0 \\ 1 \text{ “\&” } 1 \rightarrow 1 \end{cases}$$

Operadores (XII)

5) Operadores *Bitwise Logical*:

Ej.: `c = n | MASK;` → pone a uno todos los bits de n que son uno en MASK y no cambia el resto

$$\begin{array}{ll} \text{MASK} & (01111101)_2 \\ n \quad | & (10101001)_2 \\ \hline & (11111101)_2 \end{array} \rightarrow \begin{cases} 0 \text{ “|” } 0 \rightarrow 0 \\ 0 \text{ “|” } 1 \rightarrow 1 \\ 1 \text{ “|” } 0 \rightarrow 1 \\ 1 \text{ “|” } 1 \rightarrow 1 \end{cases}$$

`c = n ^ MASK;` → Uno si los bits de n y MASK son distintos y cero si son iguales

$$\begin{array}{ll} \text{MASK} & (01111101)_2 \\ n \quad \wedge & (10101001)_2 \\ \hline & (11010100)_2 \end{array} \rightarrow \begin{cases} 0 \text{ “}\widehat{}\text{” } 0 \rightarrow 0 \\ 0 \text{ “}\widehat{}\text{” } 1 \rightarrow 1 \\ 1 \text{ “}\widehat{}\text{” } 0 \rightarrow 1 \\ 1 \text{ “}\widehat{}\text{” } 1 \rightarrow 0 \end{cases}$$

`x = x << 3;` $\longleftrightarrow$ x = x * 2^3
`x = x >> 3;` $\longleftrightarrow$ x = x / 2^3
} Si x es entero (y no se viola el rango de la variable)

`0177` = $(01111111)_2$
$\sim$`0177` = $(10000000)_2$

`c = n & (~ 0177);` → Pone a cero los últimos 7 bits de n

¡ OJO !

`x = 1`
`y = 2`
} $\leftrightarrow$ { `x && y` es 1 ; `x & y` es 0

(TRUE and TRUE = TRUE)

$(1)_{10} = (00000001)_2$
& $(2)_{10} = (00000010)_2$
} $\rightarrow (00000000)_2 = (0)_{10}$

Operadores (XIII)

6) Otros operadores:

Fortran:

- `//` $\rightarrow$ Concatenación de strings

  Ej. `'Hola '//'Amigo'`

C:

- Ternarios $\longrightarrow$ e_1 ? e_2 : e_3

  Ej. `z = ( a > b ) ? c : d;` $\longrightarrow$ $\begin{cases} \text{Si a} > \text{b} & \rightarrow & \text{z = c} \\ \text{Si a} \leq \text{b} & \rightarrow & \text{z = d} \end{cases}$

7) Recomendaciones finales:

- No fiarse del orden. ¿ $\left\{ \begin{array}{l} \texttt{a[i] = i++;} \\ \texttt{a[i] = ++i;} \end{array} \right\}$?
- En caso de duda utilizar múltiples instrucciones y utilizar los paréntesis

Sentencias de control (I)

Sentencias de control

- **Fortran**
  - if aritmético
  - if lógico
  - if — then — else — endif
  - do — enddo
  - do while — enddo

  → Bucles
  - goto (incondicional)
- **C**
  - if
  - if — else
  - if — else —if
  - switch

  → Sentencias de ejecución condicionales
  - for
  - while
  - do — while

  → Bucles
  - break
  - continue
  - goto

  → Sentencias de control incondicionales

Sentencias de control (II)

1.1) IF – ELSE:

Permiten la ejecución de un conjunto de instrucciones si se cumple la condición:

```
if ( exp )
   sentencia1;
else
   sentencia2;
```

Opcional

```
if ( exp ) {
   .....;
   .....;
   .....;
} else {
   .....;
   .....;
}
```

¡ OJO ! El `else` se asocia al `if` más cercano, luego

```
if ( n > 0 )
   if ( a > b )
      z = a;
   else
      z = b;
```

$\neq$

```
if ( n > 0 ){
   if ( a > b )
      z = a;
} else
   z = b;
```

A veces se abrevia como:

if (exp) $\longleftrightarrow$ if (exp != 0)

Ej. `if ( error )` $\longleftrightarrow$ `if ( error != 0 )`

Sentencias de control (III)

1.2) IF – ELSE IF:

```
if ( exp1 )
   .........;
else if ( exp2 )
   .........;
else if ( exp3 )
   .........;
else
   .........;
```

Opcional

```
if ( exp1 ) {
   .........;
   .........;
}
else if ( exp2 ) {
   .........;
   .........;
}
else if ( exp3 ) {
   .........;
   .........;
}
else {
   .........;
   .........;
}
```

Opcional

Sentencias de control (IV)

1.3) SWITCH

- Permite establecer un selector de opciones según condiciones

```
switch ( expr. entera ){
   case int1:
      .......;
      .......;
   case int2:
      .........;
      .........;
      break;
   case .... :
   default :
}
```

- Las sentencias `case:` y `default` se pueden colocar en cualquier orden
- La ejecución se desvía al `case` cuyo valor coincide con el valor de `expr. entera`
- Una vez en el `case` correspondiente, la ejecución continúa hasta el final de la instrucción `switch`
- Si ningún valor de los `case` coincide con el valor de `expr. entera` se ejecuta desde `default`
- Para que cada `case` ejecute sólo sus instrucciones hay que colocar un `break` al final
- Se recomienda no utilizarlo. Pueden ejecturse instrucciones que no queremos.

Sentencias de control (V)

2.1) WHILE

- Ejecuta las instrucciones repetitivamente mientras se cumpla la expresión.

```
while ( exp )
  sentencia;
```

```
while ( exp ){
  .......;
  .......;
}
```

Sentencias de control (VI)

2.2) FOR

- Sentencia de repetición de instrucciones

Inicialización de variables (opcional)	condición de repetición	Instrucciones de ejecución al final
↓	↓	↓
for (exp1 ;	exp2 ;	exp3)
sentencia;		

```
for ( exp1 ; exp2 ; exp3 )
   sentencia;
```

$\longleftrightarrow$

```
exp1;
while ( exp2 ) {
   sentencia;
   exp3;
}
```

- ¡ OJO ! La condición se comprueba al principio, antes de cada interación
- ¡ OJO ! En este caso, las comas garantizan el orden de evaluación.

Ej.

```
for ( i = 0 ; i < n ; i++ )

for ( i = 0 , j = 0 ; i < n && j < m ; i++, j++ )
```

Sentencias de control (VII)

2.2) FOR

- Ejemplos:

```
for (i = n - 1 ;   i   >= 0 ; i--) {
       v[  i] =   i;                    /* => i = n-1, ..., 0 */
}

for (i = n     ; --i   >= 0 ;    ) {
       v[  i] =   i;                    /* => i = n-1, ..., 0 */
}

for (i = n     ;   i-- >  0 ;    ) {
  v[  i] =   i;                         /* => i = n-1, ..., 0 */
}

for (i = n     ;   i   >  0 ;    ) {
  i--; v[  i] =   i;                    /* => i = n-1, ..., 0 */
}

for (i = n     ;   i   >  0 ;    ) {
       v[ --i] =   i;                   /* => ¿¿¿¿¿¿   ?????? */
}

for (i = n     ;   i   >  0 ;    ) {
       v[  i] =  --i;                   /* => ¿¿¿¿¿¿   ?????? */
}
```

Sentencias de control (VIII)

2.3) DO – WHILE

- Instrucción de repetición con estructura:

```
do                              do {
   sentencia;          ó            sentencias;
while ( exp );                  } while ( exp );
```

Se repiten las sentencias siempre que se cumpla la condición indicada en `exp`

¡ OJO ! La condición se comprueba al final. El bucle se ejecuta al menos una vez.

Su uso no es muy habitual por el anterior motivo.

Sentencias de control (IX)

2.4) OTRAS INSTRUCCIONES DE CONTROL

- `break;` → rompe y sale del bucle que se está ejecutando.
- `continue;` → Obvia la ejecución del resto de sentencias de esa iteración del bucle.
  Pero no finaliza el bucle. Salta las sentencias de esa iteración

Ej.:

```
for ( i = 0 ; i < n ; i++ ){
  if ( a[i] < 0 )
    continue;
  // Sentencias que se ejecutan para términos a[i] positivos
  .....;
  .....;
}
```

- GOTO → Desvía la ejecución a otro punto del programa.

  `goto` *etiqueta*
  sentencias;
  etiqueta:

  No se deben usar a menos que sean imprescindible

  La sentencia `goto` y la línea con *etiqueta* deben estar en la misma función

  No se puede saltar a una etiqueta de un bucle desde fuera del bucle

Punteros y vectores (I)

1) CONSIDERACIONES PREVIAS

- Direcciones:

  `int i` $\rightarrow$
  - Reserva espacio en memoria para un entero
  - Guarda la posición de memoria donde se almacena el entero
  - `i` contiene el valor que se almacena en ese espacio de memoria

  `i = 5;` $\rightarrow$ Almacena el valor entero `5` en el espacio de memoria de `i`

  `&i` $\rightarrow$ Extrae la dirección de memoria donde comienza a almacenarse `i`

- Vectores:

  `int a[10];` $\rightarrow$
  - Reserva espacio en memoria para 10 enteros tipo `int` (a[0] , a[1], ... , a[9])
  - `a[i]` es el contenido del espacio de la $(i+1)$-ésima componente.
  - `a` es la dirección de memoria de la primera componente del vector

    `a` $\Longleftrightarrow$ `&a[0]`

  Ej.:

  `a[3] = 5;` $\rightarrow$ Almacena el valor 5 en el espacio de la cuarta componente.

  `double b[4] = { 1., -2., 7., -5.};` $\rightarrow$ Declara el vector `b` y le asigna valores iniciales

- Podemos obtener la dirección y definir vectores de todos los tipos de variables

Punteros y vectores (II)

2) PUNTEROS

- Son variables especiales: almacenan posiciones de memoria de otros tipos de variables.

`int *p;` $\rightarrow$
- Reserva espacio para la dirección de memoria de un entero
- `p` es una variable puntero (o puntero) a un entero `int`
- ¡¡ El entero no tiene porqué existir !!

```
p = &i;   // Extrae la posición en memoria de la variable i y la almacena en p
i = *p;   // Busca el valor almacenado en la posición de memoria
          //      que indica p y lo guarda en i
```

Aritmética de punteros

- Incrementar la posición de memoria: `p += 3;`
- Decrementar la posición de memoria: `p -= 7;`
- Restar posiciones de memoria: `n = p - q;`

- Es consistente. (Tiene en cuenta el número de posiciones de memoria del tipo de variable en cuestión, avanza o retrocede tantos bytes como ocupe cada tipo de variable)
- Todas las demás operaciones están prohibidas (por lógica)
- Uso de punteros
  - Pasar o enviar variables a funciones
  - Manejar, dimensionar, ... vectores

Punteros y vectores (III)

2) PUNTEROS

- Relación entre punteros y vectores

  El nombre de un vector es un puntero que indica la posición de la primera componente

  ```
  int a[10], * p;
  p = a;      ( ⟷  p = &a[0]; )
  a[i]   ⟷  *(a+i) // El valor de la dirección de memoria (a) se incrementa
                   en i posiciones y luego se obtiene su valor con (*)
  ```

  OJO:
  - `p` es un puntero variable. Su valor (dirección a la que apunta) se puede modificar
  - `a` es un puntero constante. Su valor ya está asignado y el espacio en memoria reservado

  La primera componente de un vector en C es la 0: (`int v[100]` → {`v[0], ...,v[99]`})

  Ej.:

```
int a[10], *p;                    int a[10], *p;
int i;                            int i;
p = a;                            p = a;
for (i = 0; i < 10; i++)    ⇔     for (i = 0; i < 10; p++, i++)
  printf("%d\n",p[i]);              printf("%d\n",*p);
```

Punteros y vectores (IV)

2) PUNTEROS

- Ejemplos de aplicación

```
int main(void)
{
  int suma ( int , int * );
  int a[10], sa, n;
  ...
  sa = suma( n, a );
  ...
}

int suma ( int n , int *a );
{
  int s = 0;
  int i;
  for ( i = 0 ; i < n ; i++ )
  {
    s += a[i];
  }
  return s;
}
```

$\longleftrightarrow$

Sin perder el puntero `a`

```
      ↑
for ( p = a + n ; p > a ; )
{
  s += *(--p);
}
      ↑

int suma ( int n , int *a );
{
  int s = 0;
  int *p;
  for ( p = a + n ; a < p ; a++ )
  {
    s += *a;
  }
  return s;
}
```

Punteros y vectores (V)

3) STRINGS

```
char nombre[5] =   "pepe";
```

↑

`{'p', 'e', 'p', 'e', '\0'};` → Acaba en un Null

OJO !! Un string es un vector, no una variable

No podemos hacer:

```
char nombre[5];
nombre = "pepe";
```

Tendríamos que hacer:

```
char nombre[5];
nombre[0] = 'p';
nombre[1] = 'e';
nombre[2] = 'p';
nombre[3] = 'e';
nombre[4] = '\0';
```

O bien:

```
char * nombre;
nombre = "pepe"; /* <- ¡Interesante!  ¿Por qué funciona?  */
```

La manipulación de strings se hace a través de funciones

Punteros y vectores (VI)

4) VECTORES MULTIDIMENSIONALES

- `int m[N][M];` → { Reserva espacio para $N \times M$ enteros tipo `int` (*) ; `m` es el puntero a un vector de punteros (**) }

(*)
```
m[0][0], m[0][1], m[0][2], ..., m[0][M]
m[1][0], m[1][1], m[1][2], ..., m[1][M]
...
m[N][0], m[N][1], m[N][2], ..., m[N][M]
```

(**) `m` → Puntero a un vector de punteros (m[0], m[1], ..., m[N])

⇓

m[0] ⇒	m[0][0]	m[0][1]	m[0][2]	...	m[0][M]
m[1] ⇒	m[1][0]	m[1][1]	m[1][2]	...	m[1][M]
⋮	⋮	⋮	⋮		⋮
m[N] ⇒	m[N][0]	m[N][1]	m[N][2]	...	m[N][M]

↑

Componentes del vector de punteros cuyos valores indican donde comienza cada fila de la matriz de datos

Punteros y vectores (VII)

4) VECTORES MULTIDIMENSIONALES

¡¡ OJO !!

`m[i][j]` $\longleftrightarrow$ es un entero: `int n; n = m[i][j];`

`(m[i])` $\longleftrightarrow$ `int *p;` $\begin{cases} \texttt{p} \longleftrightarrow \texttt{\&m[i][0];} \\ \texttt{p} \longleftrightarrow \texttt{m[i];} \end{cases}$

`m` $\longleftrightarrow$ `int (*p)[M];` $\begin{cases} \texttt{p} \longleftrightarrow \texttt{\&m[i];} \\ \texttt{p} \longleftrightarrow \texttt{m;} \end{cases}$

Luego:

`m`	$\longleftrightarrow$ `&m[0]`	`m[0]`	$\longleftrightarrow$ `&m[0][0]`
`m + 1`	$\longleftrightarrow$ `&m[1]`	`m[1]`	$\longleftrightarrow$ `&m[1][0]`

El compilador traduce

`m[i][j]` $\longleftrightarrow$ `*(m[i] + j)` $\longleftrightarrow$ `*(*(m+i) + j)`

- Las filas se almacenan internamente una a continuación de otra, aunque no necesariamente tiene que ser así.

Dimensionamiento de arrays (I)

1) DIMENSIONAMIENTO ESTÁTICO (STATIC ALLOCATION)

- Reserva espacio en memoria de forma estática e inmutable para una un programa.
- Se realiza al mismo tiempo que se declara el array
- El sistema reserva una parte de la memoria para albergar el contenido del array y guarda la dirección de la primera componente del array en el puntero que la caracteriza.
- El espacio en memoria se reserva de la parte de la parte de memoria denominada STACK. Su existencia está garantizada al arrancar el programa.
- En los sistemas actuales este STACK es de tamaño reducido y no se recomienda dimensionar arrays grandes de este modo.

  FORTRAN:

  Ej.:
  ```
  real *8 v
  dimension v(100)
  ```

  C:

  Ej.:
  ```
  double v[100];
  ```

Dimensionamiento de arrays (II)

2) DIMENSIONAMIENTO DINÁMICO (DYNAMIC ALLOCATION)

- Permite asignar espacio de memoria de forma dinámica en tiempo de ejecución.
- La reserva de memoria se realiza durante la ejecución y no está garantizada la disponibilidad.

FORTRAN

En el encabezamiento del programa principal es necesario declarar las variables como “asignables dinámicamente”

```
implicit real*8(a-h,o-z)
allocatable v(:,:)      !  Indica que el array v se dimensiona
                        !  dinámicamente y tendrá dos índices
                        !  (fila y columna, por ejemplo)
```

Posteriormente en el programa se asigna memoria dinámicamente como:

```
allocate(v(nx,ny), STAT=ist)   !  Asigna nx*ny componentes a v
                               !  Si ist=0, la asignación se ha realizado
correctamente
```

- Este procedimiento sólo es aplicable para variables globales en el programa principal
- El dimensionamiento dinámico de variables globales en subrutinas es más complejo
- La liberación de la memoria asignada se realiza como: `deallocate(v)`

Dimensionamiento de arrays (III)

2) DIMENSIONAMIENTO DINÁMICO (DYNAMIC ALLOCATION)

LENGUAJE C

Se realiza creando una variable puntero del tipo que corresponda a los datos a almacenar

```
double * pv;
```

Posteriormente en el programa se asigna memoria dinámicamente como:

```
pv = (double *) malloc( n * sizeof(double));
```

reserva n x (bytes de un double) y almacena la posición en el puntero

`sizeof( tipo );` devuelve el número de bytes que ocupa el tipo de variable indicado

- Este procedimiento sólo es aplicable para dimensionamiento dinámico en el programa principal
- Para liberar la memoria una vez deja de utilizarse:

```
free(pv);
```

Dimensionamiento de arrays (IV)

2) DIMENSIONAMIENTO DINÁMICO (DYNAMIC ALLOCATION)

LENGUAJE C

- La utilización de dimensionamiento dinámico en funciones es más complejo dado que se envían y reciben copias de los valores de las variables (de los punteros también) y la instrucción `malloc` modifica el valor del puntero.
- La solución consiste en enviar el puntero del puntero que representará al vector.

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  void dyndim( int, double **);
  double ** v;
  int n;

  scanf("%d",&n);

  dyndim(n,v);
}

void dyndim(int n, double ** v)
{
  (*v) = (double*) malloc(n*sizeof(double));
}
```

```
#include <stdio.h>
#include <stdlib.h>

void main(void)
{
  void dyndim( int, double **);
  double * v;
  int n;

  scanf("%d",&n);

  dyndim(n,&v);
}

void dyndim(int n, double ** v)
{
  (*v) = (double*) malloc(n*sizeof(double));
}
```

Funciones (I)

1) DESCRIPCION GENERAL

- Las funciones son subprogramas encargados de realizar las operaciones de un algoritmo o de parte del mismo. Convenientemente enlazadas y definidas conforman un programa de ordenador

  Funciones del sistema: son funciones propias del compilador que se encuentran en las librerías del sistema. (`<stdio.h>`, `<stdlib.h>`, `<math.h>`)

  Funciones propias: son funciones propias desarrolladas por el usuario.

- Definición:

  tipo_retorno nombre_funcion(*declaracion_parametros, si es necesario*)
  {
  Declaraciones;
  Sentencias;
  }

- Partes:
  - *tipo_retorno:* tipo de valor que devuelve la función al finalizar (int, float, int *, ...)
  - *Nombre_función:* nombre con el que se identifica la función (OJO: a-z $\equiv$ A-Z)
  - *declaracion_parametros:* conjunto de tipos y variables asociadas que recibe la función

Funciones (II)

1) DESCRIPCION GENERAL

Ej.:

```
int power(int base, int n)     // Eleva la base al exponente n
{
  int i, p = 1;

  for (i = 1; i <= n; ++i)
    p = p * base;

  return p;          // Devuelve el valor de p como valor de retorno
}
```

Esta función recibe dos argumentos de tipo entero (`base` y `n`)

Y devuelve un argumento de tipo entero (En este caso con el valor de `p`)

Funciones (III)

2) DECLARACIÓN DE PARÁMETROS

- Los parámetros se reciben y se envían desde la función de origen en el orden en que se indica.
- Los parámetros se envían desde la función de origen por valor (En Fortran, por referencia).
- La función recibe como parámetros copias de los valores de la función de origen.
- Si se modifican en la función no se modifican en la función original.

3) PROTOTIPOS DE FUNCIONES

- Antes de realizar la llamada a una función debe indicarse en la función de origen un prototipo de la misma. Se definen en el encabezamiento, antes de la función `main` o en ficheros externos de tipo “header” (*.h)
- Los prototipos tienen la misma estructura que las definiciones de función pero sin nombres de variables. Sólo indican tipos (tanto de retorno como de parámetros)

  Ej.:

```
int power(int , int );     // Los parámetros deben ser dos
                                    variables int y se devuelve otro int
```

Funciones (IV)

4) LLAMADAS A FUNCIONES

- La definición de los parámetros sólo incorpora el nombre de las variables (sin los tipos)
- El valor del tipo de retorno se almacena en una variable de tipo adecuado (salvo en el caso de funciones con tipo de retorno `void`)

```
#include <stdio.h>
int power(int , int);     // Prototipo de funciones
void main(void)
{
  int i, j = 2, k = -3;
  for (i = 0; i < 10; ++i )
    printf("%d %d %d"\n, i, power(2,i), power(-3,i));
}

int power(int base, int n)
{
  int i, p = 1;
  for (i = 1; i <= n; ++i)
    p = p * base;
  return p;
}
```

5) FUNCIONES Y RECURSIVIDAD

- Se dice que una función es recursiva cuando al llamarse a sí misma desarrolla un algoritmo determinado
- Las sentencias de la función contienen una llamada a la propia función.
- No es en absoluto recomendable para cálculo científico.
- OJO ! El espacio en memoria (stack) necesario para la ejecución aumenta peligrosamente y no se puede evitar con esta técnica.

Función normal

```
int factorial(int n)
{
  int i, f;
  for ( i = 1, f = 1; i <= n; i++)
    f *= i;
  return(f);
}
```

Función recursiva

```
int Rfactorial(int n)
{
  return( n < 2 ?  1 :  n * Rfactorial(n-1));
}
```

5) FUNCIONES MATEMÁTICAS

- Se introducen incorporando los prototipos de las librerías del sistema:

```
#define <stdlib.h>
#define <math.h>
```

```
abs(i),          labs(l),    fabs(d),
exp(f),          log(d),     log10(d),
pow(x,y),        sqrt(d),
srand(iseed),    ⟶           rand(),
cos(d),          sin(d),     tan(d),
acos(d),         asin(d),    atan(d),     atan2(s,c),
cosh(d),         sinh(d),    tanh(d),
```

- En algunos casos el compilador de GNU puede requerir la opción de compilación

```
-lm
```

para que las funciones matemáticas tengan efecto.

Uso de ficheros en C (I)

1) ACCESO A FICHEROS

- Apertura de ficheros:

  Se realiza mediante la instrucción `fopen` como:

  ```
  fp = fopen ("nombre", "modo_apertura")
  ```

  donde:

`fp`	puntero a un fichero (FILE *)
`"nombre"`	nombre completo del fichero a abrir (y su ruta completa si es necesario)
`"modo"`	`"r"` → Apertura para lectura (“read only”) `"w"` → Apertura para escritura (“write”) `"a"` → Añadir al fichero actual (“append”) `"b"` → Lectura o escritura en binario (“binary”)

- Cierre de ficheros:
  - Se realiza mediante la instrucción `fclose` como:

    ```
    fclose( fp );
    ```

Ej.:

```
FILE *fp;
...
fp = fopen("resultados.txt","w");
...
fclose(fp);
```

Lectura y escritura de datos (I)

1) CONSIDERACIONES GENERALES

- La lectura y escritura de datos se realiza mediante funciones del sistema cuyos prototipos se encuentran en el fichero de encabezamientos (header):

```
#include <stdio.h>
```

Por este motivo es necesario incluir estos ficheros en los programas para poder utilizar esas funciones

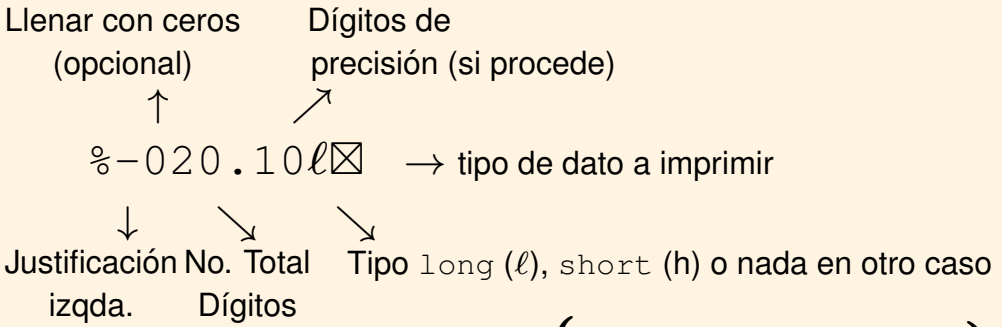
Lectura y escritura de datos (II)

2) STANDARD OUTPUT (stdout)

- Escritura en el “standard output” (por defecto, la pantalla):

```
int printf("formato", variables[si procede]);
```

El “formato” contiene el texto a escribir
Si se desea imprimir el valor de variables se indica su tipo en el formato precedido de `%`
Devuelve un `int` que indica el número de elementos escritos correctamente



⊠ →
- `d` Entera en base decimal
- `f` Real en coma fija
- `e` Real y notación científica
- `g` Lo más breve entre `f` y `e`

- `u` Decimal
- `o` Octal
- `x` Hexadecimal

} `unsigned`

- `c` `int` como `char`
- `s` string hasta NULL (puntero a vector de char’s)

`%%` → Imprime el símbolo %

Lectura y escritura de datos (III)

3) STANDARD INPUT (stdin)

- Lectura a través del “standard input” (por defecto, el teclado):

```
int scanf("formato",punteros a variables[si procede]);
```

El formato es similar al utilizado en `printf`

Devuelve un `int` que indica el número de elementos leídos correctamente (aunque a veces se omite)

OJO!:

El formato `long` se indica `ld`

El formato `double` se indica `lf`

El formato `long double` se indica `LF`

Se pasan siempre punteros

Lectura y escritura de datos (IV)

4) LECTURA Y ESCRITURA EN STRINGS

- Permiten realizar las mismas operaciones que con el stdin y el stdout pero en strings de caracteres.

Escritura (prototipo de la función)

```
int sprintf(string (char *), "formato", variables[si procede]);
```

Escribe los datos en el string de caracteres cuyo puntero se indica.
Devuelve el número de elementos escritos correctamente.

Lectura (prototipo de la función)

```
int sscanf(string (char *), "formato", punteros a variables[si procede]);
```

Lee los datos del string de caracteres cuyo puntero se indica y se almacenan en las variables

Lectura y escritura de datos (V)

5) LECTURA Y ESCRITURA DE `CHAR`

- Permiten leer y escribir bytes (`char`) en stdin y stdout respectivamente

Escritura

```
int putchar( int )     // Ej.:  c2 = putchar( c );
```

Imprime el caracter almacenado en una variable de tipo `int` en el stdout
Devuelve el mismo caracter si no hay errores o bien `EOF` (End Of File) si hay errores
EOF ← CTRL + Z en entornos Windows
EOF ← CTRL + D en entornos Unix/Linux

Lectura

```
int getchar()     Ej.:  c = getchar();
```

Lee byte a byte (char a char) los datos del stdin.
En cada ejecución lee un byte y se coloca para la lectura del siguiente.
El byte leído se devuelve como un `int`

Lectura y escritura de datos (VI)

6) LECTURA Y ESCRITURA EN FICHEROS

- Permiten realizar las mismas operaciones que `printf` y `scanf` pero en ficheros (de texto o binarios).

Escritura

```
int fprintf( FILE * ,"formato",variables[si procede]);
```

Escribe los datos en el fichero cuyo puntero (FILE *) se indica.
Devuelve el número de elementos escritos correctamente.

Lectura

```
int fscanf(FILE *,"formato",punteros a variables[si procede]);
```

Lee los datos del fichero cuyo puntero (FILE *) se indica y se almacenan en las variables

Lectura y escritura de datos (VII)

7) LECTURA Y ESCRITURA DE `CHAR` EN FICHEROS

- Permiten leer y escribir bytes (`char`) en o de ficheros

Escritura

```
int putc( int, FILE * )     // Ej.:  c2 = putc( c , fp );
```

Imprime el caracter almacenado en la variable de tipo `int` en el fichero del puntero indicado
Devuelve el mismo caracter si no hay errores o bien `EOF` (End Of File) si hay errores
EOF ← CTRL + Z en entornos Windows
EOF ← CTRL + D en entornos Unix/Linux

Lectura

```
int getc( FILE * )     // Ej.:  c = getc( fp );
```

Lee byte a byte (char a char) los datos del fichero cuyo puntero se indica.
En cada ejecución lee un byte y se coloca para la lectura del siguiente.
El byte leído se devuelve como un `int`

Uso del command line (I)

1) USO DEL COMMAND-LINE

- Los programas en Lenguaje C son normalmente de tipo:

```
int main(void)
{
  ...
}
```

- Sin embargo, el lenguaje C ofrece la posibilidad de que el usuario introduzca datos a un programa en la misma línea de comandos desde la que se lanza la ejecución.

  Ej.:

```
$> factorial 5

$> copy archivo1.f archivo2.f

$> gcc hola.c -o hola
```

Uso del command line (II)

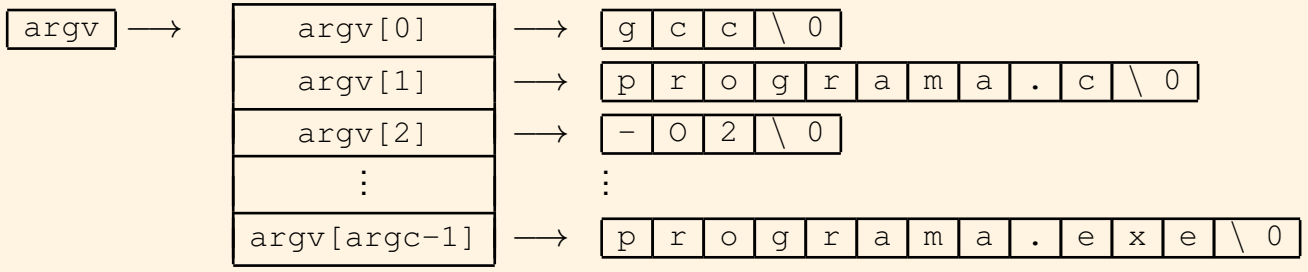
1) USO DEL COMMAND-LINE

- Para utilizar el command-line, los programas son de tipo:

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

donde:

- `argc`: contiene el número de argumentos separados por espacios introducidos en el "command-line"
- `argv[]`: es un vector (puntero) de strings(punteros) que almacena los textos de los argumentos introducidos en el "command line".
  `argv` tiene tantas componentes de tipo string (vector de char's) como indique `argc`.
  `argv[0]` es el puntero al string en el que se almacena el propio nombre del programa que se ejecuta.



Uso del command line (III)

1) USO DEL COMMAND-LINE

Consideraciones importantes:

- Las componentes de `argv` son de tipo string (vector de caracteres)
- Si desean utilizarse datos del command-line como valores numéricos es necesario utilizar funciones para transformarlos.

  Por ejemplo, `sscanf()` lectura en strings ó `atoi()`: “ascii to integer”
- Dado que `argv` es un vector (puntero) de punteros a strings también se puede indicar como:

```
int main(int argc, char ** argv)
{
  ...
}
```

En este caso,

```
*(argv+0)   ⟺   argv[0]
*(argv+1)   ⟺   argv[1]
*(argv+2)   ⟺   argv[2]
    ⋮               ⋮
```

Estructuras (I)

ESTRUCTURAS

Son variables especiales que se componen internamente de otras variables de cualquier tipo

```
struct nombre_estructura{variable1; variable2; ...};
```

- Declaración de estructuras:

```
struct {                        // Crea la estructura de datos
          float x;              // coordenadas, formada por dos float (x e y)
          float y;
       }  coordenadas;
```

```
struct {   COORDENADAS          // Crea el tipo de estructura de datos
          float x;              // COORDENADAS, formado por dos float (x e y)
          float y;
       }
```

```
struct   COORDENADAS   coordenadas;
            ↑              ↑
          tag          estructura
```

Estructuras (I)

ESTRUCTURAS

- Miembros de las estructuras:
  - Se gestionan como: `nombre_estructura.nombre_variable`

    Ej.: { `coordenadas.x` / `coordenadas.y`

- Punteros:

```
struct COORDENADAS coordenadas;

struct COORDENADAS *c;

c = &coordenadas;
```

`(*c).x` ↔ `c->x` ↔ `coordenadas.x`

`(*c).y` ↔ `c->y` ↔ `coordenadas.y`

Union (I)

UNION

- Funcionan igual que las estructuras
- Pero sólo existe 1 de sus miembros (el que queramos) en cada caso
- La variable `union` puede almacenar variables de distinto tipo

Ej.:

```
union {
          int i;
          float x;
}         cosa
```

→ `cosa.i` puede almacenar un entero
`cosa.x` puede almacenar un real

y ocupan la misma posición de memoria !!!
(aunque su tamaño puede ser distinto)

Fields y definición de tipos (I)

FIELDS

```
unsigned bits :  3;
```
// bits es una variable de 3 bits sin signo

TYPEDEF

```
typedef int (*pf)();

pf funcion;
```
→ `funcion` es un puntero a una función que devuelve un entero

```
typedef int Length;

Length len, maxlen;
```
→ `len` y `maxlen` son `int`

```
typedef char * String;

String line[MX];
```
→ `line[MX]` es un `string`

```
typedef struct COORDENADAS Punto;

Punto p;
```
→ es una estructura de tipo `COORDENADAS`

PROGRAMACIÓN EN C Y FORTRAN (XXXI)

- Bibliografía:

- *The C programming Language*, B.W. Kernighan, D.M. Ritchie, 2^{nd} Edition, Prentice Hall Software Series, Upper Saddle River, NJ, USA, 1978
- *Fortran 77 for engineers and scientists with an introduction to Fortran 90*, Larry Nyhoff y Sandford Leestma, Prentice Hall, Upper Saddle River, NJ, USA, 1996
- *Aprenda Fortran 8.0 como si estuviera en primero*, Javier García de Jalón, Franciso de Asís de Ribera, E.T.S. Ingenieros Industriales, Universidad Politécnica de Madrid, 2005