– **Typeset by GMNI &** FoilTEX –

PROGRAMMING IN C AND FORTRAN

Fermín Navarrina, José París

GMNI — GROUP OF NUMERICAL METHODS IN ENGINEERING

School of Civil Engineering
Technological Innovation Centre in Building and Civil Engineering (CITEEC)
University of A Coruña

GMNI - Group of Numerical Methods in Engineering
http://caminos.udc.es/gmni

Index

Structure of a program (I)

1) Code lines:

- **Fortran**
  - **Formato fijo**
    - Column 1 → (c,d,!) Comment line
    - Columns 2 to 5 Line numbering →
    - Column 6 → Line continuation
    - Columns 7 to 72 → Space available for instructions
    - Columns 73 to 80 → Space for additional comments
  - There is no free format (except in more modern versions of Fortran).
- **C**
  - The format is free.
  - Sentences are managed and grouped using ";" and "{ }"

Structure of a program (II)

2) Comments:

Fortran

`c`, `d`	→	In the first column, sets the entire line as a comment. Letter `d` is reserved for a compilation option.
`!`	→	comments on the line from the position in which it is found It's the common way nowadays
cols. 73 a 80	→	They are always comments.

C

`/*...*/`	→	It comments on one or more lines as a block
`//...`	→	It comments on the line from the current position Standard in modern version of C and C++

Structure of a program (III)

3) Preprocessor (compilation directives) (Pre-compilation phase):

Fortran

`d`	→ Comments with `D` are disabled.
`include 'filename'`	→ Incorporate external text files Ex. Header files.
`parameter(SYMBOL=value)`	→ Replace the text `SYMBOL` by the `value` before compiling.

Structure of a program (IV)

3) Preprocessor (compilation directives) (Pre-compilation phase):

C

`#include <stdio.h>`	→ It incorporates the system library `stdio.h`
`#include "lib.h"`	→ It incorporates a user library named `lib` that exists in the current folder
`#define SYMBOL`	→ It defines `SYMBOL` as a parameter
`#define SYMBOL value`	→ Replace before compiling the text `SYMBOL` by the `value`

`#if`
`#elif`
`#endif`
	→ It allows to incorporate conditions to the pre-processor (It is used, for instance, to work depending on the existing operating system)

`#define FUNC_SYMB(VAR) F(VAR)`	→ It replaces `FUNC_SYMB(VAR)` by the function F and it replaces the text `VAR` in the expression of the function F
`#define ...\` `.....`	→ Ii indicates that the definition continues in the next line
`#ifdef SYMBOL` `...`	→ If `SYMBOL` is defined then ...

Structure of a program (V)

3) Pre-processor (compilation directives) (Pre-compilation phase):

Examples of possible unwanted secondary effects:

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

$$\longrightarrow \quad j = (i++ \;+\; i++) \neq \left\{ \begin{array}{l} \texttt{j = ACME(i);} \\ \texttt{i++;} \end{array} \right.$$

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

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

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

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

Structure of a program (VI)

4) Main program:

Fortran

```
!234567
      program name      ! If omitted it is stated by default as: main
      ...
      end
```

C

```
int main(void)     /* It does not receive any data and it returns an integer */
{
   ...
}
```

ó

C[1]

```
int main(argc, **argv)     /* It returns an integer control value
                              and it receives two arguments*/
{
   ...
}
```

(1) This option allows to incorporate data into the program that is written directly into the command line when it is executed. (Ex. `gfortran program.f -o program.exe`)

Structure of a program (VII)

5) Groups of sentences:

Fortran $\rightarrow$ It is not possible.

C $\rightarrow$ They can be defined.

- They are defined by `{...}`, that contain the set of sentences.
- They allow definitions of local variables that are specific and exclusive to that group of sentences.

Structure of a program (VIII)

6) Source code readability:

Fortran

- Uppercase and lowercase letters are equivalent.
- The use of lowercase letters is recommended (in general).
- It is recommended to use capital letters to define symbols.
  (Ex. `PARAMETER (PI=3.1415926535)`)
- It is recommended to use the first letter capitalized for subroutine names.
  (Ex. `subroutine Product(...)`)
- Line indentation must be done using spaces.
- Due to the reduced format available, no spaces are left between operations.

C

- Uppercase and lowercase letters are different.
  -Except in function names (for compatibility with other languages)
  -It is not recommended to use this distinction in practice.
- The use of lowercase letters is recommended (in general).
- It is recommended to use capital letters to define symbols.
  (Ex. `#define PI 3.1415926535`)
- It is recommended to use the first letter capitalized for function names.
  (Ex. `int Product(...){...}`)
- The indentation of the lines is done using tabs.
- It is common practice to leave spaces between operations. (Ex. `i = j + k;`).

Variables and constants (I)

Variables:

1) Definition of variable:

It is a symbolic name that identifies:

- the memory address where the information associated with that name is saved.
- And the storage space (depending on the type of variable), that is: the value

2) Names for variables:

Fortran

They cannot start with a numeric digit. (0-9)

We can use the characters a-z $\equiv$ A-Z, 0-9, ...,$

Considerations to keep in mind:

- Only the first 31 characters for "internal" names
- Only the first 6 characters for "external" names
- They depend on the compiler and the compilation options

Names cannot correspond to instruction names (do, max, int, ...)

- They should be mnemonic
- ATTENTION: 0 $\neq$ O and 1 $\neq \ell$

C

The same criteria as for Fortran apply

It should be noted that, in general, a-z$\neq$A-Z

- Except in external function names (for compatibility)

Variables and constants (II)

2) Names of variables:

Reserved names:

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 → depending on the implementation

The underlined names were incorporated into the new ANSI standard.

Variables and constants (III)

3) Types of variables (Basic ones):

Fortran

- `integer*1` (o byte), `integer*2`, `integer*4`, `integer*8`
  `integer` (The most common) (logic for 64 bits CPUs)
- `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` $\longrightarrow$ This is the most advisable approach in practice
- `character *(n) text` $\rightarrow$ It creates an alphanumeric variable of n characters

Variables and constants (IV)

3) Types of variables (Basic ones):

C

- `unsigned`
  - `char` → 1 byte
  - `short (int)` → integer variable ≥ `char` (Normally 2 bytes)
  - `int` → integer variable ≥ `short` (Normally 4 bytes)
  - `long (int)` → integer variable ≥ `int` (Normally 4 bytes)
  - `long long` → integer variable ≥ `long` (and nothing else, a priori)
- `char` → 1 byte
- `short (int)` → integer variable ≥ `char` (Normally 2 bytes)
- `int` → integer variable ≥ `short` (Normally 4 bytes)
- `long (int)` → integer variable ≥ `int` (Normally 4 bytes)
- `long long` → integer variable ≥ `long` (and nothing else, a priori)
- `float` → real en simple precision (but unknown number of bytes)
- `double` → real in double precision (but unknown number of bytes)
- `long double` → real in quadruple precision (but unknown number of bytes)

Variables and constants (V)

3) Types of variables: Local variables (automatic ones) and external ones

Local V.

- They are internal variables to a module
- The name (memory address) is unknown to other modules
- The storage space is unknown to other modules
  - -Permanent information in main program $\rightarrow$ guaranteed value
  - -Temporary information in other modules $\rightarrow$ value not guaranteed on subsequent calls

External V.

- These are variables common to several modules
- The name (memory address) is known to other modules
- The storage space is:
  - -Permanent in different modules $\rightarrow$ guaranteed value

Variables and constants (VI)

3) Types of variables: Local variables (automatic ones) and external ones

ForTran

All variables are local to modules, except

- Subroutine and function arguments (sent by reference, not by value)
- COMMON blocks

```
common /name/ vble_1, vble_2, ...
```

C language

All variables are local to modules or groups (`{...}`), except

- The ones declared as `extern`
  - The ones declared before: `int main(void)`
  - Those declared within each function as `extern` (*)

```
extern external_variable
```

(*) If the functions are all in the same file, it is not necessary to declare them all.

Variables and constants (VII)

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

Variables and constants (VIII)

Ex. C language

```
void main(void)
{
     void calc(double, int, double *)     /* Prototype of the function */
     int n;                               /* Declaration of variable n */
     double x, y;                         /* Declaration of variables x and y */
     n=2;
     x=5.;
     calc(x,n,&y);                        /* Call to the function calc */
     print("%f",y);                       /* Printing the result on screen */
     exit(0);
}
void calc(double a, int i, double *pb)
{
     double z;
     z = a + 1.;
     *pb = pow(z,i);
}
```

- The prototype defines the type of function and the variables that are passed.
- A priori, variables modified in the subroutine are sent with & and received with $*$.

Variables and constants (IX)

4) Constants:

Fortran

- Integer:

  $\pm$ [number in decimal form] $\rightarrow$ digit between 0 and 9

  They are stored in the integer type by default

  `integer*2`
`integer*4`
`integer*8` } $\rightarrow$ Depending on the compilation option

- Real:

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

  It depends on the compilation options

  They are saved in the corresponding type or by default (real*4, real*8, real*16)

Variables and constants (X)

4) Constants:

Fortran

- Alphanumeric constants:

They are not variables as such. They are Descriptors.

They contain internally → { The memory address of the beginning of the string
The length of the string (number of characters)

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

} → hello, world

12 charact.

↑

Hollerith format

- Logic constants:

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

Variables and constants (XI)

4) Constants:

C language

- Integer constants: (see the file `limits.h` in the compiler libraries)

  Char:

  They are saved as (`unsigned` / `signed`) { `char`, `short`, `int`, `long`, `long long` }

`'x'`	→	unsigned char (1 byte) Saves the ASCII code of x	`'\r'`	→	carriage return
			`'\f'`	→	form feed
`'\0'`	→	null	`'\a'`	→	bell (beep)
`'\n'`	→	newline	`'\\'`	→	backslash
`'\t'`	→	horizontal tab	`'\?'`	→	question mark
`'\v'`	→	vertical tab	`'\''`	→	single quote
`'\b'`	→	backspace	`'\"'`	→	double quote
			`'%%'`	→	Symbol %

`'\o'`, `'\oo'`, `'\ooo'` → octal { o → 1 octal number; oo → 2 octals numbers; ooo → 3 octal numbers }

`'\xh'`, `'\xhh'` → hexadecimal { h → 1 hexadec. number; hh → 2 hexadec. numbers }

They are saved in { unsigned char / signed char } depending on the implementation

Variables and constants (XII)

4) Constants:

C language

`short, int, long, long long`

$\pm$ [number in decimal form] $\rightsquigarrow$ digits (0-9), first $\neq 0$, except for the zero.
$\pm$ 0[number in octal form] $\rightsquigarrow$ digits (0-7), first digit is a 0.
$\pm$ 0x[number in hexadecimal form] $\rightsquigarrow$ digits (0-9), letters (a-f) $\equiv$ (A-F)

They are saved in $\left(\begin{array}{c}\texttt{unsigned}\\ \texttt{signed}\end{array}\right)$ $\left\{\begin{array}{c}\texttt{char}\\ \texttt{short}\\ \texttt{int}\\ \texttt{long}\\ \texttt{long long}\end{array}\right\}$

$\downarrow$ depending on the sign or as indicated

$\downarrow$ the bare minimum or as indicated

constant + $\left\{\begin{array}{c}\texttt{u}\\ \texttt{U}\end{array}\right\}$ $\rightarrow$ unsigned Ex. `347U`

constant + $\left\{\begin{array}{c}\texttt{l}\\ \texttt{L}\end{array}\right\}$ $\rightarrow$ long long Ex. `2598L`

They can be combined (Ex. `0xFUL` $\equiv$ 15)

NOTE: Expressions with integer constants are evaluated at compile time (not at runtime)

Variables and constants (XIII)

4) Constants:

C language

- Real: (See the file `float.h in the compiler libraries`)

`float, double, long double`

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

± 123.4, $\pm .1234\text{e}{+}3$ $\rightsquigarrow$ `double` (Default option)

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

Variables and constants (XIV)

4) Constants:

C language

- Alphanumeric: (Strings) They are treated as character arrays.

$$\begin{cases} \texttt{"hello, world"} \Leftrightarrow \underbrace{\texttt{"hello,"}\ \texttt{" world"}}_{\text{They are concatenated}} \rightsquigarrow \text{hello, world} \\ \texttt{""} \Leftrightarrow \text{empty string} \end{cases}$$

  - They consist of a vector of characters of type `char` with a `'\0'` (`null`) at the end.
  - They are defined by the vector where they are stored, and their length ends at the first `'\0'`.
  - The final `'\0'` is established by agreement.

Variables and constants (XV)

4) Constants:

C language

- Enum: (It allows creating sets of variables with assigned constant values)

```
enum boolean {NO,YES};
```
$\longrightarrow \begin{cases} \text{NO} \longleftrightarrow 0 \\ \text{YES} \longleftrightarrow 1 \end{cases}$

```
enum escapes {BEL='\a', BACKSPACE='\b', ..., RETURN='\r'};
enum meses {JAN=1, FEB, MAR, APR,... DEC};
```
$\longrightarrow \begin{cases} \text{JAN} = 1 \\ \vdots \\ \text{DEC} = 12 \end{cases}$

Values are stored in variables of type `int`

Declaración:

```
enum boolean {NO,YES};
enum boolean yesorno, acepted;
⋮
yesorno = NO; accepted = YES;

enum boolean {NO,YES} yesorno, accepted;
enum {NO,YES} yesorno, accepted;
```

Variables and constants (XVI)

5) Changes of variable types:

Fortran: They are performed using functions

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

$\rightarrow$ vble.type2 = f(vble.type1)

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

C language: It is done using casts (assignments).

Casts:

(type) variable /* It assigns the value of `variable` to the new variable type. */
/* Applies only to the argument immediately following */

```
Ej.   i = (int) x;   // It saves the integer part of x in i
Ej.   z = (float) j;   // It saves the integer value j in z
```

Variables and constants (XVII)

6) Variables initialization:

Fortran →

It is not possible except with the instruction `data`

```
data vble1,vble2,vble3 /value1, value2, value3/
```

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 components of value 0.0
```

Global variables are initialized once and at the beginning

Are local variables initialized each time? → YES

Variables and constants (XVIII)

6) Variables initialization:

C →

They can be initialized directly when declared:

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

They are initialized:
- Once, the permanent or global ones
- Each time in each module, the local variables

If they are defined as constants and
they are initialized and cannot be modified afterwards

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

Arrays can also be initialized when declaring them:

```
char v[5] = { 1 , 2 , 3 , 4 , 5 };
char text[7] = { 'M','o','n','d','a','y','\0' };
char text[7] = "Monday"; char texto2[5] = "five";
```

Variables and constants (XIX)

7) Assigning values to variables:

Fortran → { Direct assignment to variables. Ex. `variable=value`
In arrays, component by component. Ex. `vector(i)=value`

C →

Direct assignment to variables.

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

In arrays:

```
int v[5];
v = { 1 , 2 , 3 , 4 , 5 };          ⟶ Not correct
int v[5] = { 1 , 2 , 3 , 4 , 5 };   ⟶ Correct
char text[7];
texto[7] = "Monday";                ⟶ Not correct
char text[7] = "Monday";            ⟶ Correct
```

Attention: An additional character must be reserved for null '\0'

```
float x, z;
int y, j, k;
x = y = z = 3.2;   ¿¿??
i = j++ = k = 8;   ¿¿??
```

Operators (I)

Operators

	Fortran	C
Arithmetic	Yes	Yes
Relational	Yes	Yes
Logic	Yes	Yes
Incremental	No	Yes
Bitwise logical	No	Yes
Others	Concatenation	Ternary

Operators (II)

1) Arithmetic operators:

Fortran:

- { Unary: `-` — Sign change. Affects one variable.
  Binarios: `+, -, *, /` — Basic operations. They affect 2 variables

- They are applied to `integer, real, complex`

- Priority { 1) - (unary)
  2) *, /
  3) +, - } → It can be altered with parentheses.

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

◇ ¡Attention! The compiler can make decisions. Don't trust it.

If you want to force a result, it is better to use intermediate variables.

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

Operators (III)

1) Arithmetic operators:

C:

- They are the same as in Fortran, plus::
  - ▷ Modulo division (%): Remainder of the division of one integer by another
  - ▷ Abbreviations: shortcut operators
- { Unary: `-` Sign change. Affects one variable.
  { Binary: `+, -, *, /, %` They affect two variables.
- They are applied to: { `char, short, int, long, long long`
  { `float, double, long double` (except modulo division %)
- Priority { 1) - (unary)
  { 2) `*, /, %`
  { 3) `+, -` } → It can be altered with parentheses.

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

◇ ¡Attention! The compiler can make decisions. Don't trust it.

If you want to force a result, it is better to use intermediate variables

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

Operators (IV)

1) Arithmetic operators:

C:

- Abbreviations:

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

  being op one operation $\rightarrow$ op $\in$ {+, -, *, /, %} (also {<<, >>, &, ^, |})

  ¡Attention ! $\rightarrow$ { Try not to mix variables of different types
  Force an appropriate rate change using promotion rules (casts)

  Ex. { `xmod += v[i] * v[i];` $\longleftrightarrow$ xmod = xmod + v[i] * v[i]
  `v[i] /= xmod;` $\longleftrightarrow$ v[i] = v[i] / xmod

Operators (V)

2) Relational operators:

Fortran:

- `.gt.`, `.ge.`, `.lt.`, `.le.` → Greater than, greater than or equal to, less than, less than o
  `.eq.`, `.ne.` → Equal, not equal.
- Only scalars are compared.
- If the scalars are of different types, they are promoted to the most complex type but it is not advisable.
- Ex.
```
if (i.eq.1)
if (x.gt.5)   ! If x is real*8 → 5 it is converted to real*8
```
- They have lower priority than arithmetic operations:

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

  In any case, it is better to use parentheses to avoid ambiguity.

Operators (VI)

2) Relational operators:

C:

- These operators are actually numeric:

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

- `( x < y )` takes the value $\left\{ \begin{array}{l} 0 \text{ if not true } (0 \equiv \text{ FALSE}) \\ 1 \text{ if it is true } (1 \equiv \text{ TRUE}) \end{array} \right.$

- Generally, $\left\{ \begin{array}{l} 0 \equiv \text{FALSE} \\ 1 \equiv \text{TRUE (any non-null value indicates TRUE)} \end{array} \right.$

  ¡Attention! Do not confuse

  $\left\{ \begin{array}{lcl} \texttt{x = y} & \rightarrow & \text{It assigns the value of } \texttt{y} \text{ in } \texttt{x} \\ \text{with} & & \\ \texttt{x == y} & \rightarrow & \text{Returns the value 0 if } \texttt{x} \text{ is not equal to } \texttt{y} \\ & & \text{and the value 1 if } \texttt{x} \text{ is equal to } \texttt{y} \end{array} \right.$

- They have higher priority than arithmetic operations (unlike in Fortran)

  It is advisable to always use parentheses to avoid conflicts.

  Ej. `a + b != c;` $\Longleftrightarrow$ `a + ( b != c );` $\rightarrow \left\{ \begin{array}{ll} \text{is equal to a} & \text{if b is equal to c} \\ \text{is equal to a+1} & \text{if b} \neq \text{c} \end{array} \right.$

Operators (VII)

3) Logic operators:

Fortran:

- $\left\{ \begin{array}{ll} \texttt{.and.}, \texttt{.or.} & \longrightarrow \quad \text{binary} \\ \texttt{.not.} & \longrightarrow \quad \text{unary} \\ \texttt{.eqv.}, \texttt{.neqv.} & \end{array} \right.$
- Truth table:

  `(a).eqv.(b)` is `.true.` $\leftrightarrow$ $\left\{ \begin{array}{l} \texttt{a} \text{ and } \texttt{b} \text{ are } \texttt{.true.} \\ \texttt{a} \text{ and } \texttt{b} \text{ are } \texttt{.false.} \end{array} \right.$

  `.neqv.` is equivalent to `.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.`

Operators (VIII)

3) Logical operators:

Fortran:

- Priority of operators $\rightarrow$ $\left\{ \begin{array}{ll} \texttt{.not.} & + \\ \texttt{.and.} & \uparrow \\ \texttt{.or.} & \downarrow \\ \texttt{.eqv.}—\texttt{.neqv.} & - \end{array} \right.$
- When in doubt, it is recommended to use parentheses

  Ej. ¿What does it mean `(a.or.b.and.c)` or `(a.or.(b.and.c))`?

Operators (IX)

3) Logic operators:

C:

- They are actually numeric operators

$$\begin{cases} \text{Binary:} & \&\&,\ \| & \text{(and y or)} \\ \text{Unary:} & ! & \text{(denial)} \end{cases}$$

$$\text{Priority} \rightarrow \begin{cases} ! & + \\ \&\& & \updownarrow \\ \| & - \end{cases}$$

Then `! a && b` $\longleftrightarrow$ `( ! a ) && b`

It is recommended to use parentheses to avoid confusion

Doubt: ¿`if ( ! valid )...` or `if ( valid == 0 )...`?

- Ex.: to check if the (year) is a leap year:

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

or

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

Operators (X)

4) Incremental operators:

C:

`++`, `--` $\longrightarrow$ is $\begin{cases} \text{ahead} & \rightarrow \text{The increase precedes the operations} \\ \text{behind} & \rightarrow \text{The increase follows the operations} \end{cases}$

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

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

- Can only be applied to variables.
- Cannot be applied to expressions.

```
z = ( x + y )++; // Not a valid expression.
```

When in doubt, avoid confusion by using parentheses.

Ex. What do they mean?

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

Operators (XI)

5) *Bitwise Logical* operators:

C:

$\&$	`bitwise and`
$\mid$	`bitwise or`
$\wedge$	`bitwise exclusive or`
$<<$	`left shift` (Shift the bits to the left by the indicated positions)
$>>$	`right shift` (Shift the bits to the right by the indicated positions)
$\sim$	complement to one (unary)

Ex.: `c = n & 0177;` $\rightarrow$ resets everything except the last 7 bits of n, that are not modified

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

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

Operators (XII)

5) Operators *Bitwise Logical*:

Ej.: `c = n | MASK;` $\rightarrow$ sets all bits of n that are 1 in 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;` $\rightarrow$ 1 if the bits of n and MASK are different, and 0 if they are the same

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

$$\left.\begin{array}{lcl} \texttt{x = x << 3;} & \longleftrightarrow & x = x * 2^3 \\ \texttt{x = x >> 3;} & \longleftrightarrow & x = x / 2^3 \end{array}\right\} \text{If x is an integer (and the ange is not violated)}$$

$$\begin{aligned} 0177 &= (01111111)_2 \\ \sim 0177 &= (10000000)_2 \end{aligned}$$

`c = n & (~ 0177);` $\rightarrow$ Resets the last 7 bits of n

¡Attention! $\left.\begin{array}{l} \texttt{x = 1} \\ \texttt{y = 2} \end{array}\right\} \leftrightarrow \begin{cases} \texttt{x \&\& y} & \text{is 1} \quad \text{(TRUE and TRUE = TRUE)} \\ \texttt{x \& y} & \text{is 0} \quad \left.\begin{array}{l} (1)_{10} = (00000001)_2 \\ \& \quad (2)_{10} = (00000010)_2 \end{array}\right\} \rightarrow (00000000)_2 = (0)_{10} \end{cases}$

Operators (XIII)

6) Other operators:

Fortran:

- `//` $\rightarrow$ Strings concatenation

  Ex. `'Hello '//'my friend'`

C:

- Ternary $\longrightarrow$ e_1 ? e_2 : e_3

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

7) Final Suggestions:

- Don't trust the order. $\left\{ \begin{array}{l} \texttt{a[i] = i++;} \\ \texttt{a[i] = ++i;} \end{array} \right\}$?
- When in doubt, use multiple instructions and use parentheses

Control sentences (I)

Control sentences

- Fortran
  - Arithmetic if
  - Logic if
  - if — then — else — endif
  - do — enddo
  - do while — enddo

  → Loops
  - goto (unconditional)
- C
  - if
  - if — else
  - if — else —if
  - switch

  → Conditional execution sentences
  - for
  - while
  - do — while

  → Loops
  - break
  - continue
  - goto

  → Unconditional execution sentences

Control sentences (II)

1.1) IF – ELSE:

They allow a set of instructions to be executed if the condition is satisfied:

```
if ( exp )
   sentence1;
else
   sentence2;
```

} Optional

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

¡ Attention ! The `else` is linked to the closest `if`, thus

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

$\neq$

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

It is sometimes abbreviated as:

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

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

Control sentences (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
   .........;           }
}
```

Control sentences (IV)

1.3) SWITCH

- It allows you to set up a selection of options based on conditions

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

  - The sentences `case:` and `default` can be placed in any order
  - The execution is diverted to the `case` whose value matches the value of the `integer expr.`
  - Once in the corresponding `case`, execution continues until the end of the instruction `switch`
  - If no value in the `case` matches the value of `integer expr.`, it is executed from `default` onwards.
  - In order for each `case` to execute only its instructions, a `break` must be placed at the end of them.
  - It is recommended not to use it. Instructions that we do not want may be carried out.

Control sentences (V)

2.1) WHILE

- Execute the instructions repeatedly as long as the expression is true.

```
while ( exp )
  sentence;
```

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

Control sentences (VI)

2.2) FOR

- Sentence for repeating instructions

Initialization of variables (optional)	condition for repeating	instructions to be executed at the end
↓	↓	↓
for (exp1 ;	exp2 ;	exp3)
sentence;		

```
for ( exp1 ; exp2 ; exp3 )          <-->   exp1;
   sentence;                               while ( exp2 ) {
                                              sentence;
                                              exp3;
                                           }
```

- ▷ ¡ Attention ! The condition is checked at the beginning, before each iteration.
- ▷ ¡ Attention ! In this case, commas ensure the order of evaluation.

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

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

Control sentences (VII)

2.2) FOR

- Examples:

```
for ( i = 0 ; i < n ; i++ )
  v[i] = i;

for ( i = 0 ; i < n ;     ){
  --i;          // The program fails due to access to v[-1]. Incorrect loop
  v[i] = i;
}

for ( i = n ; i > 0 ;     ){
  ¿v[i] = --i;?   or   ¿v[--i] = i;?   →   Not clear.
}

for ( i = n ; --i >= 0 ;   ){    →   Ends in i=-1.
  v[i] = i;
}

for ( i = n ; i-- > 0 ;   ){    →   Ends in i=-1.
  v[i] = i;
}
```

Control sentences (VIII)

2.3) DO – WHILE

- Repeat instruction with structure:

```
do                                  do {
   sentence;          or               sentences;
while ( exp );                      } while ( exp );
```

The sentences are repeated as long as the condition indicated in `exp` is met

¡ Attention ! The condition is checked at the end. The loop is executed at least once.

Its use is not very common for the above reason.

Control sentences (IX)

2.4) OTHER CONTROL INSTRUCTIONS

- `break;` → Breaks and exits the loop that is running.
- `continue;` → Skips the execution of the remaining statements in that iteration. But the loop does not end. Skip the statements in that iteration.

Ex.:

```
for ( i = 0 ; i < n ; i++ ){
  if ( a[i] < 0 )
    continue;
  // Statements that are executed for terms a[i] possitive
  .....;
  .....;
}
```

- GOTO → Diverts execution to another point in the program.

$$\begin{cases} \texttt{goto}\ label \\ sentences; \\ label: \end{cases}$$

Should not be used unless absolutely necessary

The sentence `goto` and the line with $label$: must be in the same function

Labels should never be placed inside loops

Pointers and vectors (I)

1) PREVIOUS CONSIDERATIONS

- Directions:

  `int i` $\rightarrow$
  - Reserve memory space for an integer
  - Saves the memory location where the integer is stored
  - `i` contains the value stored in that memory space

  `i = 5;` $\rightarrow$ Store the integer value 5 in the memory space of `i`

  `&i` $\rightarrow$ Extract the memory address where the storage of `i` begins

- Vectors:

  `int a[10];` $\rightarrow$
  - Reserve memory space for 10 integers of type `int` (a[0] , a[1], ... , a[9])
  - `a[i]` is the content of the $(i+1)$-th component.
  - `a` is the memory address of the first component of the vector

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

  Ex.:

  `a[3] = 5;` $\rightarrow$ Store the value 5 in the fourth component.

  `double b[4] = { 1., -2., 7., -5.};` $\rightarrow$ Declares the vector `b` and assigns initial values to it

- We can obtain the direction and define vectors for all types of variables

Pointers and vectors (II)

2) POINTERS

- They are special variables: store memory locations of other types of variables.

`int *p;` $\rightarrow$ { Reserves space for the memory address of an integer; `p` is a pointer variable (or pointer) to an integer `int`; The integer does not necessarily have to exist !!

```
p = &i;   // Extracts the position in memory of variable i and stores it in p
i = *p;   // Search for the value stored in the memory location
          //  that indicates p and saves it in i
```

Pointers arithmetic

- { Increase the memory position: `p += 3;`; Decrease the memory position: `p -= 7;`; Substract memory positions: `n = p - q;`
- It is consistent. (It takes into account the number of memory locations for the variable type in question, and advances or retreats as many bytes as each variable type occupies)
- All other operations are prohibited (by logic).
- Use of pointers { Passing or sending variables to functions; Manage, allocate, ... vectors

Pointers and vectors (III)

2) POINTERS

- Relationship between pointers and vectors

  The name of a vector is a pointer that indicates the position of the first component

  ```
  int a[10], * p;
  p = a;      ( ⟷  p = &a[0]; )
  a[i]   ⟷  *(a+i) // The value of memory address (a) is incremented
                      by i positions and then its value is obtained with (*)
  ```

  Attention:
  - `p` is a variable pointer. Its value (direction that points out) can be modified
  - `a` is a constant pointer. Its value is already defined and the space in the memory reserved

  The first component of a vector in C is 0
  (`int v[100]` → {`v[0], ...,v[99]`})

  Ex.:

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

2) POINTERS

- Application examples

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

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

$\longleftrightarrow$

Without loosing the pointer a

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

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

Pointers and vectors (V)

3) STRINGS

```
char name[5] =  "john";
```

`{'j', 'o', 'h', 'n', '\0'};` → ends up with a Null

Attention !! A string is a vector, not a variable

We can not write:

```
char name[5];
name = "john";
```

We could write:

```
char name[5];
name[0] = 'j';
name[1] = 'o';
name[2] = 'h';
name[3] = 'n';
name[4] = '\0';
```

Or:

```
char * name;
name = "john";
```

String manipulation is done through functions

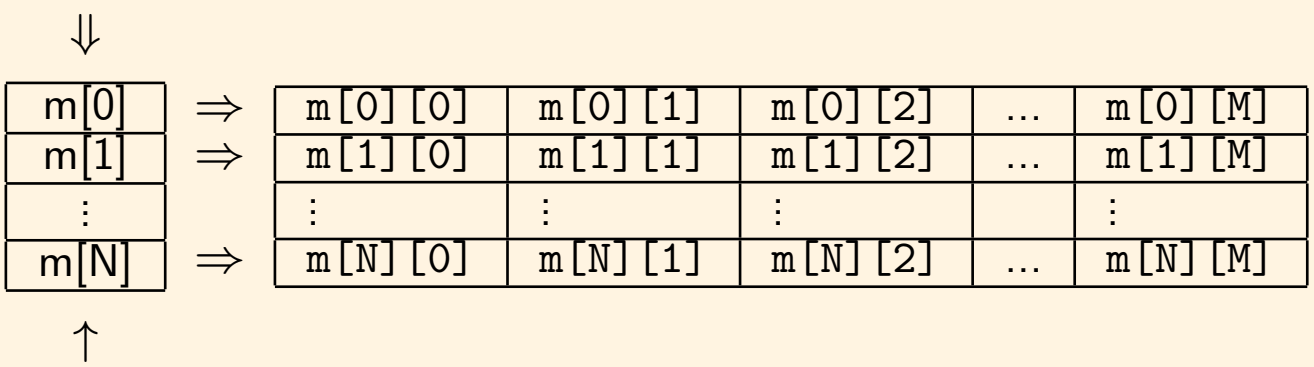
Pointers and vectors (VI)

4) MULTIDIMENSIONAL VECTORS

- `int m[N][M];` → { Reserves space for $N \times M$ integers type `int` (*)
  `m` is the pointer to a vector of pointers (**)

(*)
```
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 → Pointer to a vector of pointers (m[0], m[1], ..., m[N])



Components of the vector of pointers whose values indicate where each row of the matrix starts

4) MULTIDIMENSIONAL VECTORS

¡¡ OJO !!

`m[i][j]` $\longleftrightarrow$ is an integer: `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]`

The compiler translates

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

- The rows are stored internally one after another, although this does not necessarily have to be the case.

Arrays allocation (I)

1) STATIC ALLOCATION

- Reserves memory space in a static and immutable manner for a program
- It is reserved at the same time as the array is declared
- The system reserves a portion of memory to store the contents of the array and saves the address of the first component of the array in the pointer that locates it.
- The memory space is reserved from the part of memory called STACK. Its existence is guaranteed when the program starts up.
- In current systems, this STACK memory is small in size, and it is not recommended to allocate large arrays in this way.

  FORTRAN:

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

  C:

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

Arrays allocation (II)

2) DYNAMIC ALLOCATION

- It allows to allocate memory space dynamically at runtime.
- Memory reservation is performed during execution and availability is not guaranteed.

FORTRAN

In the header of the main program, it is necessary to declare the variables as "dynamically allocatable."

```
implicit real*8(a-h,o-z)
allocatable v(:,:)     !  Indicates that the array v is dynamically
                       !  allocated and will have two indices
                       !  (row and column, for instance)
```

Later in the program, memory is allocated dynamically as:

```
allocate(v(nx,ny), STAT=ist) !  It allocates nx*ny components for v
                             !  ist=0 means correct allocation
```

- This procedure is only applicable for dynamic dimensioning in the main program
- The use of dynamic sizing in subroutines is more complex.
- The release of allocated memory is performed as follows: `deallocate(v)`

Arrays allocation (III)

2) DYNAMIC ALLOCATION

C LANGUAGE

This is done by creating a pointer variable of the type corresponding to the data to be stored

```
double * pv;
```

Later in the program, memory is allocated dynamically as:

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

reserves n x (bytes of a double) and stores the location in the pointer

`sizeof( type );` returns the number of bytes occupied by the specified variable type

- This procedure is only applicable for dynamic dimensioning in the main program
- To free up memory once it is no longer in use:

```
free(pv);
```

Arrays allocation (IV)

2) DYNAMIC ALLOCATION

C language

- The use of dynamic allocation in functions is more complex since copies of variable values (including pointers) are sent and received, and the `malloc` instruction modifies the value of the pointer.
- The solution consists of sending the pointer of the pointer that will represent the vector.

```c
#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));
}
```

```c
#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));
}
```

Functions (I)

1) GENERAL DESCRIPTION

- Functions are subprograms responsible for performing the operations of an algorithm or part of it. Conveniently linked and defined, they form a computer program

  System functions: these are functions specific to the compiler that are found in the system libraries. (`<stdio.h>`, `<stdlib.h>`, `<math.h>`)

  Native functions: these are functions developed by the user

- Definition:

  return_type function_name(*variables_declaration, if needed*)
  {
  Declarations;
  Sentences;
  }

- Parts:
  - *return_type:* type of value returned by the function upon completion (int, float, int *, ...)
  - *Function_name:* name that identifies the function (Attention: a-z ≡ A-Z)
  - *variables_declaration:* set of types and associated variables that the function receives

Functions (II)

1) GENERAL DESCRIPTION

Ex.:

```
int power(int base, int n)      // Raise the base to the exponent n
{
  int i, p = 1;

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

  return p;           // Returns the value of p as the return value
}
```

This function takes two arguments of type integer (base and n)

And returns an integer argument (in this case with the value of p)

Functions (III)

2) VARIABLES DECLARATION

- Parameters are received and sent from the source function in the order indicated.
- Parameters are passed from the source function by value (in Fortran, by reference)..
- The function receives copies of the values from the source function as parameters.
- If they are modified in the function, they are not modified in the source function

3) PROTOTYPES OF FUNCTIONS

- Before calling a function, a prototype of that function must be specified in the source function. They are defined in the header, before the `main` function or in external header files (*.h)
- Prototypes have the same structure as function definitions but without variable names. They only indicate types (both return types and parameter types).

  Ex.:

```
int power(int , int );      // The arguments must be two int variables
                            // and the return value is other int
```

4) FUNCTIONS CALLS

- The definition of the parameters only includes the names of the variables (without the types)
- The value of the return type is stored in a variable of the appropriate type (except in the case of functions with a `void` return type)

```
#include <stdio.h>
int power(int , int);      // Prototype of functions
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) FUNCTIONS AND RECURSIVITY

- A function is said to be recursive when it calls itself to develop a specific algorithm
- The statements in the function contain a call to the function itself.
- It is not at all recommended for scientific calculations.
- Attention ! The memory space (stack) required for execution increases dangerously and it cannot be avoided with this technique.

Normal function

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

Recursive function

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

5) MATHEMATICAL FUNCTIONS

- They are introduced by incorporating the prototypes of the system libraries:

```
#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),
```

- In some cases, the GNU compiler may require the compilation option

  `-lm` $(-\ell m)$

  for mathematical functions to take effect.

Use of files in C (I)

1) ACCESS TO FILES

- Files opening:

  This is done using the `fopen` command as follows:

  ```
  fp = fopen ("name", "opening_mode")
  ```

  where:

`fp`	ppointer to a file (FILE *)
`"name"`	complete name of the file to be opened (and its complete file route if needed)
`"mode"`	`"r"` → Open for reading ("read only") `"w"` → Open for writting ("write") `"a"` → Append to the existing file ("append") `"b"` → Reading or writting in binary ("binary")

- Files closing:
  - This is done using the `fclose` command as follows:

    ```
    fclose( fp );
    ```

Ej.:

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

Reading and writing data (I)

1) GENERAL CONSIDERATIONS

- Data reading and writing is performed using system functions whose prototypes are found in the header file (header):

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

For this reason, it is necessary to include these files in the programs in order to use these functions.

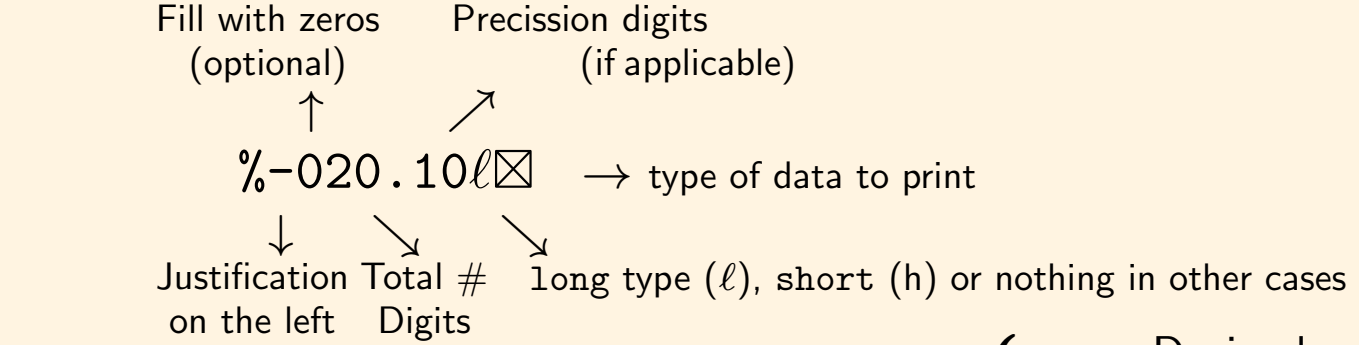
Reading and writing data (II)

2) STANDARD OUTPUT (STDOUT)

- Writing to the standard output (by default, the screen):

```
int printf("format", variables[if applicable]);
```

The "format" indicates the text to be written
If you want to print the value of variables, indicate their type in the "format" preceded by %
It returns an `int` indicating the number of elements written correctly



⊠ →

d	Integer in decimal form
f	Fixed-point real number
e	Real number and scientific notation
g	The shortest between f and e

u	Decimal	unsigned
o	Octal	
x	Hexadecimal	
c	int as char	
s	string until NULL (pointer to a vector of chars)	

%% → prints the symbol %

Reading and writing data (III)

3) STANDARD INPUT (STDIN)

- Reading data in the standard input (by default, the keyboard):

```
int scanf("format",pointers to variables[if applicable]);
```

The format is similar to the used in `printf`

It returns an `int` indicating the number of elements read correctly (although sometimes is omitted)

OJO!:

The `long` format is indicated as `ld`

The `double` format is indicated as `lf`

The `long double` format is indicated as `LF`

Pointers to the variables are sent to the function.

Reading and writing data (IV)

4) READING AND WRITING IN STRINGS

- They allow you to perform the same operations as with STDIN and STDOUT, but on character strings.

Writing (prototype of the function)

```
int sprintf(string (char *),"format",variables[if applicable]);
```

It writes the data to the character string whose pointer is indicated.
It returns the number of correctly written elements.

Reading (prototype of the function)

```
int sscanf(string (char *),"format",pointers to variables[if
applicable]);
```

It reads the data from the character string whose pointer is indicated and stores the values in the variables

Reading and writing data (V)

5) READING AND WRITING OF `CHAR`

- Allow reading and writing bytes (`char`) in the STDIN and STDOUT respectively

Writing

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

It prints the character stored in a variable of type `int` to STDOUT
It returns the same character if there are no errors, or `EOF` (End Of File) if the are errors
EOF $\leftarrow$ CTRL + Z in Windows OS
EOF $\leftarrow$ CTRL + D in Unix/Linux OS

Reading

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

It reads the data from STDIN byte by byte (char by char).
Each time it runs, it reads one byte and positions itself to read the next one.
The byte read is returned as an `int`

Reading and writing data (VI)

6) READING AND WRITING IN FILES

- They allow you to perform the same operations as `printf` and `scanf`, but on files (text or binary).

Writing

```
int fprintf( FILE * ,"format",variables[if applicable]);
```

It writes the data to the file whose pointer (FILE *) is indicated.
It returns the number of correctly written elements.

Reading

```
int fscanf(FILE *,"format",pointers to variables[if applicable]);
```

It reads the data from the file of pointer (FILE *) is indicated and stores the values in the variables

7) READING AND WRITING OF CHAR IN FILES

- It allows reading and writing bytes (`char`) in or from files

Writing

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

It prints the character stored in the variable of type `int` in the file pointed to by the pointer `fp`
It returns the same character if there are no errors, or `EOF` (End Of File) if there are errors
EOF $\leftarrow$ CTRL + Z in Windows OS
EOF $\leftarrow$ CTRL + D in Unix/Linux OS

Reading

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

Reads the data from the file whose pointer is indicated byte by byte (char by char).
Each time it runs, it reads one byte and positions itself to read the next one.

Reading and writing data (VII)

The byte read is returned as an `int`

Use of the command line (I)

1) USE OF THE COMMAND-LINE

- Programs in C language are usually of the following type:

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

- However, the C language offers the possibility for the user to enter data into a program on the same command line from which the execution is launched.

  Ej.:

```
$> factorial 5

$> copy archive1.f archive2.f

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

Use of the command line (II)

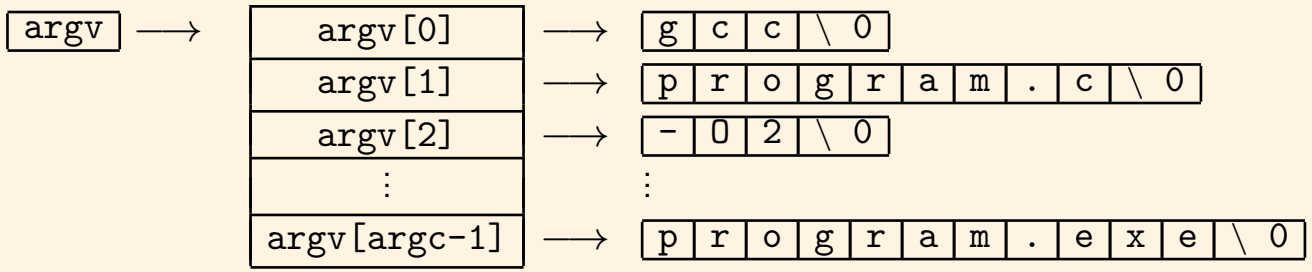
1) USE OF THE COMMAND-LINE

- To use the command line, the programs are of the type:

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

donde:

- `argc`: contains the number of arguments separated by spaces entered on the command line.
- `argv[]`: is a vector (pointer) of strings (pointers) that stores the texts of the arguments indicated in the command line..
  `argv` has as many string-type components (vector of chars) as indicated by `argc`.
  `argv[0]` is the pointer to the string where the name of the program being executed is stored.



Use of the command line (III)

1) USE OF THE COMMAND-LINE

Important considerations:

- The components of `argv` are of type string (char vector)
- If you want to use command-line data as numerical values, you need to use functions to transform them.

  For example, `sscanf()` reading of strings or `atoi()`: "ascii to integer"
- Given that `argv` is a vector (pointer) of pointers to strings it can be also used as:

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

In this case,

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

Structures (I)

STRUCTURES

They are special variables that are internally composed of other variables of any type

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

- Structures declaration:

```
struct {                        // Creates the structure coordinates
           float x;             // with two float (x and y)
           float y;
       }   coordinates;

struct   COORDINATES {          // Creates the type of structure named
         float x;               // COORDINATES with two float
         float y;
     }

struct   COORDINATES   coordinates;
              ↑              ↑
            tag          structure
```

Structures (II)

STRUCTURES

- Members of the structures:
  - They are managed as: `structure_name.variable_name`

    Ex.: $\begin{cases} \texttt{coordinates.x} \\ \texttt{coordinates.y} \end{cases}$

- Pointers:

```
struct COORDINATES coordinates;

struct COORDINATES *c;

c = &coordinates;
```

(*c).x ↔ c->x ↔ coordinates.x

(*c).y ↔ c->y ↔ coordinates.y

Union (I)

UNION

- They work the same as structures
- But there is only one of its members (the one we want) in each case.
- The `union` variable can store variables of different types

Ex.:

```
union {
          int i;
          float x;          →
}         union_example
```

`union_example.i` can save an integer
`union_example.x` can save a real

and tehy occupy the same memory space position !!!
(although its size can be different)

Fields and types definition (I)

FIELDS

```
unsigned bits :  3;          // bits is an unsigned variable of 3 bits
```

TYPEDEF

```
typedef int (*pf)();
pf function;
```
→ `function` is a pointer to a function that it returns an integer

```
typedef int Length;
Length len, maxlen;
```
→ `len` and `maxlen` are `int`

```
typedef char * String;
String line[MX];
```
→ `line[MX]` is a `string`

```
typedef struct COORDINATES Point;
Point p;
```
→ is a structure of type `COORDINATES`

- Bibliography:

- *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