# The C Programming Language

Éric Renault

## Organization

- Tutorial 1 — 30/11, 09h-12h30 — INT A07.01
  - Introduction
  - Compilation steps
  - Data types and operators
- Tutorial 2 — 30/11, 14h-17h30 — INT B03
  - Control instructions
- Tutorial 3 — 06/12, 09h-12h30 — INT B313
  - Functions
- Tutorial 4 — 07/12, 14h-17h30 — INT B04
  - Arrays
- Tutorial 5 — 09/01, 14h-17h30 — INT B07
  - Pointers
- Tutorial 6 — 16/01, 14h-17h30 — INT B313
  - Structures, unions,...
- Tutorial 7 — 23/01, 14h-17h30 — INT D012
  - The preprocessor
- Tutorial 8 — 30/01, 14h-17h30 — INT B02
  - Input / output

# The C Programming Language

Éric Renault

Institut National des Télécommunications
9, rue Charles Fourier
91011 Évry Cedex, France



# Contents





# Introduction

The C programming language :

- is linked to the development of UNIX
- was developed at Bell Labs in 1972 by B.W. Kernighan and D.M. Ritchie
- was first developed in assembly language
- was normalized in 1987 by IEEE (norm X3 J-11)

Bibliography :

- B.W. Kernighan and D.M. Ritchie
  The C Programming Language
  Prentice Hall, 1988



# Preliminary

*File*

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

*Function*

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

*Block*

```
{

    printf ( "Hey !\n" ) ;

}
```

A *statement* ends with a semi-column (`;`)

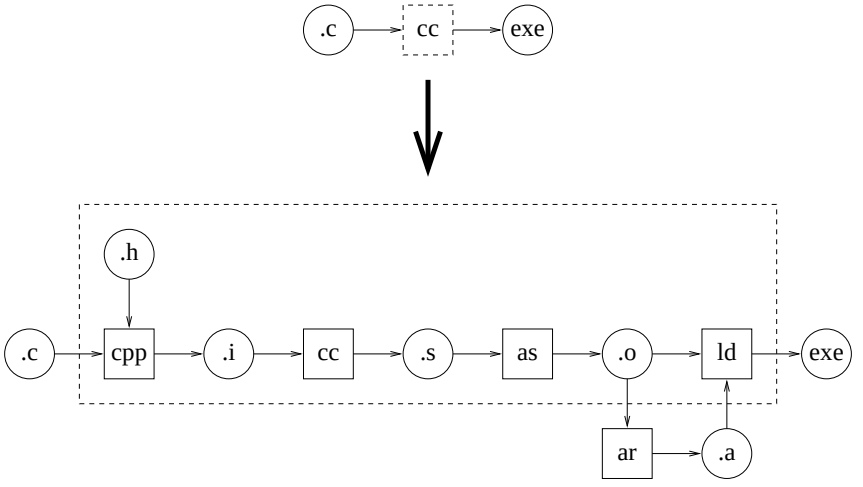
# A simple example

```
# include <stdio.h>

/*
** What a program !
*/
int main ( int argc, char * argv [ ] )
{
    printf ( "Hi there !\n" ) ;
    exit ( 0 ) ;
}
```



# Compilation steps






# Object declaration

- Objects in C are :
  - named constants
  - types
  - variables
  - functions
- An object :
  - MUST be declared before being used
  - cannot be declared twice



# CONSTANTS AND VARIABLES

# Constants

- Integers :
  - decimal : `6`, `-15`, `1752`...
  - octal : `06`, `017`, `03330`...
  - hexadecimal : `0x6`, `0xF`, `0x6D8`...
  - long : `6L`, `017L`, `0x6D8L`...
- Floating-point numbers :
  - decimal : `3.1415927`, `-5.285`...
  - scientific : `0.314116e+1`, `-654.17e-5`...
- Characters :
  - single : `'a'`, `'R'`, `';'`...
  - string : `"Hey !"`, `"a\0"`...



# Data representation

**Register :**

*Binary* 0 1 1 1 0 1 1 0 | 0 1 0 1 0 1 0 0 | 0 0 1 1 0 0 1 0 | 0 0 0 1 0 0 0 0

4th byte 3rd byte 2nd byte 1st byte

**C language notation :**

| *Notation* | Prefix | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Octal* | 0 | 1 | 6 | 6 | 2 | 5 | 0 | 3 | 1 | 0 | 2 | 0 |
| *Hexadecimal* | 0x | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | | | |
| *Decimal* | (none) | 1 | 9 | 8 | 5 | 2 | 2 | 9 | 3 | 2 | 8 | |



# Variables

- Variables are used to store data
- Each variable is associated a type
- Two kinds of type are available :
  - integers
  - floating-point numbers
- Characters are stored using one-byte integers
- Programmers can build their own types
- Collections of data are possible :
  - arrays, structures, unions...
- The value can be updated from a constant or another variable using the = sign



# Data types

- For integers (these may be declared `unsigned`) :

| | | |
|---|---|---|
| `char` | \|_\| | $-2^7 \rightarrow 2^7 - 1$ |
| `short` | \|_\|_\| | $-2^{15} \rightarrow 2^{15} - 1$ |
| `int` (or `long`) | \|_\|_\|_\|_\| | $-2^{31} \rightarrow 2^{31} - 1$ |
| `quad` | \|_\|_\|_\|_\|_\|_\|_\|_\| | $-2^{63} \rightarrow 2^{63} - 1$ |

- Valid values for `unsigned char` are $0 \rightarrow 2^8 - 1$
- For floating-point numbers :

| | | |
|---|---|---|
| `float` | \|_\|_\|_\|_\| | $\pm 10^{-38} \rightarrow \pm 10^{38}$ |
| `double` | \|_\|_\|_\|_\|_\|_\|_\|_\| | $\pm 10^{-300} \rightarrow \pm 10^{300}$ |

# `printf` function

- Used to display data on the standard output
- Defined in `stdio.h`
- To display a string of characters :

```
printf ( "my string\n" ) ;
```

- To display the content of a variable :

```
printf ( "a = %d\n", a ) ;
```

if `a` is defined as an integer



# `scanf` function

- Used to get data from the standard input
- Defined in `stdio.h`
- To update the content of a variable :

```
printf ( "a = " ) ;
scanf ( "%d", & a ) ;
```

if `a` is defined as an integer

- Note that the `&` sign is very important and will be explained later



# Conversion

- For characters :

`%c` (single character) and `%s` (string)

- For integers :

`%d` (decimal), `%o` (octal), `%u` (unsigned decimal) and `%x` (hexadecimal)

- For floating-point numbers :

`%e` (exponential), `%f` (floating-point) and `%g` (shortest from both)



# Escape characters

They all start with a \ sign :

- `\f` : page feed
- `\n` : end of line
- `\t` : tabulation
- `\0` : null character
- `\'` : quote
- `\"` : double-quote
- `\\` : the \ sign

# OPERATORS



# Arithmetics operators

Unary operators :

- unary plus : `+`
- unary minus : `-`

Binary operators :

- addition : `+`
- substraction : `-`
- multiplication : `*`
- division : `/`
- modulo : `%`

- Let $i$ be an integer
- Let $f$ be a floating-point number
- Let $\oplus$ be a binary operator (except `%`)

- $i \oplus i \longrightarrow i$
- $f \oplus f \longrightarrow f$
- $i \oplus f \longrightarrow f$



# Bitwise operators

Logical operators :

- and : `&`
- or : `|`
- exclusive or : `^`
- completion to 1 : `~` (unary operator)

Shifts :

- left : `<<`
- right : `>>`



# Examples

Logical operators :

- `0x1234 & 0xFF` $\longrightarrow$ `0x34`
- `0x1234 | 0xFF` $\longrightarrow$ `0x12FF`
- `0x1234 ^ 0xFF` $\longrightarrow$ `0x12CB`
- `~0x1234` $\longrightarrow$ `0xEDCB`

Shifts :

- `0x1234 << 4` $\longrightarrow$ `0x12340`
- `0x1234 >> 4` $\longrightarrow$ `0x123`

# Composed assignment operators

- Let `a` and `b` be variables (`b` may be a constant)
- Let $\oplus$ be an arithmetics or bitwise operator
- Instead of writing :

  `a = a` $\oplus$ `b`
- One can write :

  `a` $\oplus$`= b`
- New operators : `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`



# Increment and decrement

Two unary operators for integers :

- `++` to increment (add 1)
- `--` to decrement (substract 1)

Two possible positions :

- Before the variable : the operation is done before any other thing in the statement
- After the variable : the operation is done after any other thing in the statement



# Examples

- `int i, j ;`
- `i = 2 ;`
- `j = ++ i ;`

  $\longrightarrow$ `i` = 3, `j` = 3
- `j = i ++ ;`

  $\longrightarrow$ `i` = 4, `j` = 3



# Comparison operators

- Equal : `==`
- Not equal : `!=`
- Less than : `<`
- Greater than : `>`
- Less than or equal to : `<=`
- Greater than or equal to : `>=`

Note :

- the difference between `=` and `==`
- that *greater than or equal to* is `>=` and not `=>`

# Logical operators

Unary operator :

- not : `!`

Binary operators :

- and : `&&`
- or : `||`

- Note the difference between logical and bitwise operators



# What about booleans ?

- There is no boolean type in C

- Comparison and logical expressions are associated a logical state
- Constants, variables and other expressions are also associated a logical state :
  - If the value is 0, the logical state is *false*
  - For any value but 0, the logical state is *true*
- All may be used in logical expressions



# Conditional operator

- It may be used to replace an expression depending upon a condition

- The general form is :

  *expr1* `?` *expr2* `:` *expr3*

  which means that if *expr1* is true then *expr2* is used ; in the other case, *expr3* is used

- Example :

```
min = i < j ?  i :  j ;
```



# The `sizeof` operator

- The size of variables depends upon the underlying hardware
- In order to know the effective size for a data type, the C language provides a specific operator : `sizeof`
- It may be used with either variables or types

Examples :

- `char c ;`

  `sizeof ( c )` $\longrightarrow 1$
- `sizeof ( int )` $\longrightarrow 4$

# CONTROL INSTRUCTIONS



## Control instructions

Two kinds :

- *conditionals* are used to perform a specific set of instructions depending upon the evaluation of an expression
- *loops* are used to repeat a specific set of instructions while/until a condition is valid



## Important

In the following :

- *expr* can be any expression
- Instructions are either a single statement or a block of instructions
- Remember that a statement always ends with a semi-column (`;`) and that blocks are defined by braces (`{ ... }`)



## `if` … `else`

The `if` statement may be used in two different ways :

- `if (` *expr* `)` *true_instr*
- `if (` *expr* `)` *true_instr* `else` *false_instr*

Be careful when `if` statements are nested

→ use braces when there is an ambiguity

# Example

- Display if an integer is an even or odd number

```
int i ;

if ( i & 1 )
    printf ( "This is an odd integer\n" ) ;
else
    printf ( "This is an even integer\n" ) ;
```



# switch

- `switch (` *expr* `) {`

  `case` *constant* `:`
  *instr*

  ...

  `default :`
  *instr*

  `}`

- Instructions are executed up to the end of the block or up to the next `break` instruction



# Example

```
int i ;

switch ( i ) {
case 0 : case 2 : case 4 : case 6 : case 8 :
    printf ( "This is an even integer\n" ) ;
    break ;
case 1 : case 3 : case 5 : case 7 : case 9 :
    printf ( "This is an odd integer\n" ) ;
    break ;
default :
    printf ( "Sorry, I do not know !\n" ) ;
}
```



# while

- `while (` *expr* `)`
  *instr*

- *expr* is evaluated first
- *instr* are performed while *expr* is true

  → *instr* are executed 0, 1 or more times

# Example

- Display integers from 0 to 9 included

```
int i ;

i = 0 ;
while ( i < 10 ) {
    printf ( "%d\n", i ) ;
    i ++ ;
}
```



# `do` … `while`

- `do`
  *instr*
  `while (` *expr* `) ;`

- *expr* is evaluated after the first loop
- Then, *instr* are performed while *expr* is true
  → *instr* are executed 1 or more times



# Example

- Display integers from 0 to 9 included

```
int i ;

i = 0 ;
do {
    printf ( "%d\n", i ) ;
    i ++ ;
} while ( i < 10 ) ;
```



# `for`

- `for (` *instr1* `;` *expr* `;` *instr2* `)`
  *instr*

- *instr1* (called “initialization”) is performed before entering the loop
- *expr* (called “continuation condition”) is evaluated before each occurrence of the loop
- *instr2* (called “incrementation”) is performed after each occurrence of the loop

# Example

- Display integers from 0 to 9 included

```
int i ;

for ( i = 0 ; i < 10 ; i ++ )
    printf ( "%d\n", i ) ;
```



# Sequence break

- `break` : do not execute instructions up to the end of the block and resume on the instruction following the block
- `continue` : do not execute instructions up to the end of the block and resume on the evaluation of the expression
- `goto` : just forget it !



# FUNCTIONS



# Introduction

Functions are used :

- when an operation must be repeated several times
- when one wants to isolate a specific or complex operation

→ This is called *structured programming*

- Functions are returning a value at the end of their execution. If not, the special type `void` must be used
- The default return type is `int`

# Function declaration

- This is also called *function prototype*
- It provides the compiler the way the function is used ; the implementation is supposed to be included later
- It is composed of :
  - the type of the returned value
  - the name of the function
  - the parameter list (parameter names are optional)
- Note that there is no polymorphism (this is C++)



# Example

- Let `min` be the function which returns the minimum of two integers. It may be declared as either :

```
int min ( int a, int b ) ;
int min ( int, int ) ;
```

- Let `display` be the function which displays the $n$ characters following the one given as parameter in the ASCII table. It may be declared as either :

```
void display ( char c, int n ) ;
void display ( char, int ) ;
```



# Function definition

- It provides the compiler what the function should do
- It is composed of :
  - a header
    - the type of the returned value
    - the name of the function
    - the parameter list (parameter names are mandatory)
  - a body (a block of instructions)
- The header must match the function prototype if any
- Keyword `return` specifies which value the function returns. It is always the last operation in a function
- A function cannot be defined inside another function



# Example

- 
```
int min ( int a, int b )
{
  return a < b ?  a :  b ;
}
```

- 
```
void display ( char c, int n )
{
  while ( n )
    printf ( "%c ", c + n -- ) ;
}
```

# Function call

- It tells the compiler which function to execute and what to do with the returned value (if any)
- It is composed of :
  - the name of the function
  - the value associated to each parameter, one value for each parameter and in the same order
- Note that all parameters are mandatory. There is no default value



# Example

```
int main ( int argc, char * argv [ ] )
{
  char c ;
  int n ;

  printf ( "Gimme a char and a number :" ) ;
  scanf ( "%c %d", & c, & n ) ;
  display ( c, min ( n, 10 ) ) ;
}
```



# Variable visibility (1/2)

Inside a block :

- Local variable
- The variable is defined after its definition
- Its use is limited to the block (and inner blocks)

Qualifier (default is `auto`) :

- `auto` : the value is undefined when entering the block
- `static` : the value is saved between two calls



# Variable visibility (2/2)

Outside any block :

- Global variable
- The variable is defined after its definition

Qualifier :

- `extern` : the variable is defined in another file
- `static` : the variable is not accessible from outside the file it is defined
- default : the variable is defined in the file and can be accessed from another file

# Other qualifiers

`register` :

- the variable is preferredly stored in a register
- this may increase performance
- the number of registers is limited

`const` :

- the value cannot be changed (well ... normally)
- this is inherited from C++

- Usable for both local and global variables



# ARRAYS



# Definition

- An array is a collection of objects of the same type
- It is defined by a type, a name and a size
- Square brackets are used both to specify the size and to get an element from the array
- Element are numbered starting from 0
- Arrays may be initialized at creation time (using braces and comma). In this case, the size of the array is optional
- Note that elements of an array are contiguous in the virtual address space



# Examples

- Let `fpn` be an array of 10 floating-point numbers :

```
float fpn [ 10 ] ;
```

- Let `prime` be an array containing the five first prime numbers :

```
int prime [ ] = { 2, 3, 5, 7, 11 } ;
```

→ Note, for the last case, that the list of prime numbers must be given in extension

# Multi-dimensional arrays

- The number of dimensions for an array is not limited
- The size of each dimension must be enclosed between a pair of brackets
- Only the last dimension may be omitted

Example :

- Let `coordinates` be a list of tri-dimensional coordinates :

```
double coordinates [ 50 ] [ 3 ] ;
```



# POINTERS



# Definition

- In C, it is possible to know the address of any object : constants, variables, arrays, functions...
- It is also possible to perform arithmetic operations with them (i.e. considering them like integers)

Definition :

- A *pointer* is a variable that contains the address of another variable

- Pointers are useful to manage lists, sets, stacks, trees...



# Unary operators

The unary operator `&` :

- returns the address of a variable

The unary operator `*` :

- is used to defined a pointer as a variable
- returns the value pointed to by the pointer

- These operators are not confusing with bitwise and arithmetic operators (which are binary operators)

# Examples (1/2)

Use of unary operator `&` :

- Let `i` be an integer
- `int i ;` defines the variable
- `& i` is the address of `i` in the virtual address space

Use of unary operator `*` :

- Let `p` be a pointer to an integer
- `int * p ;` defines the pointer
- `* p` is the value of the variable pointed to by `p`
- Note that, from the definition of the pointer, the type associated to `* p` is `int`



# Examples (2/2)

| Operations | & i | i | & p | p | * p |
|---|---|---|---|---|---|
| int i | 0x1234 | | | | |
| int * p | 0x1234 | | 0x6543 | | |
| i = 3 | 0x1234 | 3 | 0x6543 | | |
| p = & i | 0x1234 | 3 | 0x6543 | 0x1234 | 3 |
| ( * p ) ++ | 0x1234 | 4 | 0x6543 | 0x1234 | 4 |
| i *= 2 | 0x1234 | 8 | 0x6543 | 0x1234 | 8 |



# Pointers and arrays

- Pointers are also associated to arrays
- The address of an array is the address of the first element of the array

Example :

- Let `tab` be an array of 10 integers
- The definition is : `int tab [ 10 ] ;`
- The address of the array is : `& ( tab [ 0 ] )`
- Another notation is the name of the array, i.e. `tab`
- Then, the value of the first element is also : `* tab`



# Pointer arithmetics (1/2)

- As an address is an integer, it is possible to perform arithmetic operations

Is it useful ?

- For example, incrementing and decrementing a pointer allows to change from one element to another one in an array

Any limit ?

- Not all arithmetic operations are interesting
- What does the sum of two pointers mean ?

# Pointer arithmetics (2/2)

How does it work ?

- Like any other variable, just change the value of the pointer (not the value of the variable which is pointed to !)

Example :

- Let `p` be a pointer to an integer
- Suppose the value for `p` is `0x1234`
- The value for `p+1` is `0x1238` as the size of an integer is 4 bytes



# Generic pointers

- Sometimes, a pointer has to be used but the type of the variable pointed to is not defined (or may differ)
- The solution consists in using a generic pointer, i.e. a pointer that can point to any kind of variables
- Special type `void` is associated to generic pointers

Example :

- `void * generic ;`
- As `sizeof ( void )` is equal to 0, pointer arithmetics are not allowed with generic pointers



# Cast operator

- The C language offers the possibility to associate another type to a variable, i.e. to deal with a variable as if it was of a type different from the one it was defined
- Parenthesis are used to specify the new type

Example :

- Let `i` be an integer
- `( double ) i` converts `i` from an integer to a double
- Let `p` be a generic pointer
- `p = ( void * ) & i ;` is a valid instruction



# Dynamic memory allocation

- Sometimes, it is very difficult at compilation time to determine the amount of memory necessary to run a program
- The C language provides functions to increase and decrease the size of the virtual address space

Allocate dynamic memory :

- `void * malloc ( size_t ) ;`

Free dynamic memory :

- `void free ( void * ) ;`

# Example

- Reserve the memory for an array of five integers :

```
int * t ;
t = ( int * ) malloc ( 5 * sizeof ( int ) ) ;
```

- Setup the array with the five first even numbers :

```
for ( i = 0 ; i < 5 ; i ++ )
  t [ i ] = 2 * i ;
```

- Free the memory :

```
free ( t ) ;
```



# String of characters

- There is no type associated to strings of characters
- A string of characters is an array of `char` ending with the special character `'\0'`

Examples :

- `char my_string [ 25 ] ;`
- `char hey [ 6 ] = "hey !" ;`
- `char * hello = "Hello" ;`



# Be careful

- Strings of characters are not basic types

Examples :

- `char my_string [ 10 ] ;`
  `my_string = "Hello !" ;` is not allowed
- `char * s1 = "Data", * s2 = "Data" ;`
  `s1 == s2` is false !

- Lots of functions dedicated to strings of characters are defined in `string.h` (see `man string`)



# DATA STRUCTURES

# Introduction

- If one wants to store information about people (for example the name and the age), one can declare two arrays :

```
char * name [ 50 ] ;
int * age [ 50 ] ;
```

- Then, for the same index, it is supposed it is the same person

- As a name and an age belong to the same person, it is reasonable to group them in a single structure and then create a single array with this structure



# Structures

- Structures are used in order to agregate information which are belonging to a single object

A structure may be used in order to create :

- a new variable having a specific structure
- a new structured type

- For both cases, a block is used to specify elements inside the new structure



# Structured variables

- To create a structure containing information about people :

```
struct {
  char * name ;
  int age ;
} person, * pointer ;
```

- `person` is a new variable which structure is defined above
- `pointer` is a new pointer to an object which structure is defined above



# Data access

To access a specific field, there are two possibilities :

- From a variable, one use '`.`'

  Example : `person . name`
- From a pointer, one use '`->`'

  Example : `pointer -> name`

- Another way to access a field from a pointer is getting the value of the variable pointed to :

  Example : `( * pointer ) . name`

# Structured types

- To create a new structured type about people :

```
struct people {
  char * name ;
  int age ;
} ;
```

- Then, to create variables and/or pointers :

```
struct people person, * pointer ;
```

- Note that `struct people ;` is a prototype for structure `struct people`. This means that the structure can be used and it will be defined later



# Bit fields

- Bit fields are used in order to manipulate bits instead of classical basic types
- This may be helpful :
  - to compress information
  - to match the definition of specific registers
- Example :

```
struct zone {
  int bool :  1 ;
  int indicator :  3 ;
  int character :  8 ;
} ;
```



# Unions

- Unions are used in order to see the same information in different ways
- Unions are used like structures for :
  - the definition of new types
  - the definition of new variables
  - the data access



# Example

- To display the value of each byte for an integer :

```
union {
  int i ;
  char c [ 4 ] ;
} example ;
example .  i = 0x12345678 ;
for ( j = 0 ; j < 4 ; j ++ )
  printf ( "0x%x ", example .  c [ j ] ) ;
```

- Result : `0x12 0x34 0x56 0x78`

# New types

- Reserved keyword `typedef` is used to create new types
- A new type may be created from :
  - a basic type
  - a structure
  - a union
  - a pointer
  - an array of one of the previous cases...
- New types are then used like any other type



# Examples

- From a basic type :

```
typedef int my_int ;
```

- From a structure or a union :

```
typedef struct people people ;
```

- From a pointer :

```
typedef char * string ;
```

- From an array :

```
typedef struct people ten_people [ 10 ] ;
```



# THE PREPROCESSOR



# Introduction

- In the compilation chain, the preprocessor is the first step
- It aims at providing pure C code using directives
- All directive dedicated to the preprocessor must begin with a # sign on the first character of the line
- Operations are :
  - File inclusion
  - Macro definition and interpretation
  - Conditionals

# File inclusion

- This is used to include a file in another one
- Traditionally, a `.c` file never includes another `.c` file
- Only `.h` files should be included
- Usually, `.h` files are defining objects. This avoids each program to redefine everything everytime

Two ways to specify the path associated to a file :

- `# include <filename.h>` for system header files
- `# include "filename.h"` for personal header files



# Macros

- Definition of a macro :

```
# define MY_CONSTANT 0x12345678
# define MIN( X, Y ) ( (X) < (Y) ? (X) : (Y) )
```

- Destruction of a macro :

```
# undef MY_CONSTANT
```

- Does a macro already exist (to be used in tests) :

```
defined ( MY_CONSTANT )
```



# Conditionals

- Conditional compilation are performed using `if ... endif` structures (see the contents of header files)
- `# if` *condition_1*

  *true_instr_1*

  `# elif` *condition_2*

  *true_instr_2*

  `# else`

  *false_instr*

  `# endif`
- This is useful to allow multiple inclusions or to provide the same program for several architectures



# INPUT / OUTPUT

# Introduction

Available operations on files :

- Open and close
- Read and write
- Go to a specific location

Two ways to access a file :

- Using the standard C library
- Using system calls



# Using the standard C library (1/3)

1. Include the corresponding header file :

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

2. Create a file descriptor :

   ```
   FILE * fd ;
   ```

3. Open the file :

   ```
   fd = fopen ( "the path", mode ) ;
   ```

   where the value for `mode` is `r`, `w` or `a` and an optional `+`

4. Do the job
5. Close the file :

   ```
   fclose ( fd ) ;
   ```



# Using the standard C library (2/3)

Read :

- `int fgetc ( FILE * )`
- `int fgets ( char *, int, FILE * )`
- `int fscanf ( FILE *, const char * [ , arg ... ] )`

Write :

- `int fputc ( int, FILE * )`
- `int fputs ( const char *, FILE * )`
- `int fprintf ( FILE *, const char * [ , arg ... ] )`



# Using the standard C library (3/3)

Where am I ?

- `long ftell ( FILE * )`

Go to a specific location :

- `int fseek ( FILE *, long, int )`

  where the last parameter (`whence`) is taken from :
  - `SEEK_SET` : from the beginning of the file
  - `SEEK_CUR` : from the current position
  - `SEEK_END` : from the end of the file

# Using system calls (1/3)

1. Include the corresponding header file :

```
# include <fcntl.h>
```

2. Create a file descriptor :

```
int fd ;
```

3. Open the file :

```
fd = open ( "the path", flag, mode ) ;
```

   See next slide for the value of `flag` and `mode`

4. Do the job
5. Close the file :

```
close ( fd ) ;
```



# Using system calls (2/3)

`flag` :

- `O_CREAT`, `O_TRUNC`, `O_APPEND`
- `O_RDONLY`, `O_RDWR`, `O_WRONLY`
- `O_EXCL`
- `O_BINARY`, `O_TEXT`

`mode` :

- `S_IREAD`, `S_IWRITE`

- To be agregated using `|`



# Using system calls (3/3)

Read :

- `int read ( int, void *, int )`

Write :

- `int write ( int, void *, int )`

Go to a specific location :

- `long lseek ( int, long, int )`