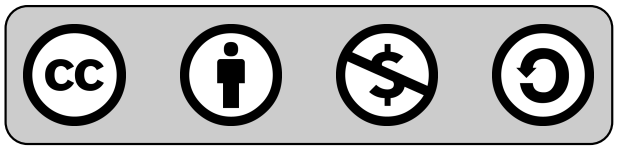
# The C Programming Language

Jérôme Hugues
ISAE/DMIA – jerome.hugues@isae.fr

# License CC BY-NC-SA 3.0



# Objective

- Review basic constructs of the C programming language: typing system, control flow, functions prototyping;
- Explore advances concepts: memory management and pointers,
- Highlight common pitfalls of the language;
- Present specific patterns for embedded systems.

# References I


Darnell, P.A. and P.E. Margolis (1991).
*C, a software engineering approach*.
Springer books on professional computing.
Springer-Verlag.
ISBN: 9783540973898.
http://books.google.com/books?id=VptQAAAAMAAJ.

Kernighan, Brian W. and Dennis M. Ritchie (Mar. 1988).
*C Programming Language (2nd Edition)*.
2nd ed.
Englewood Cliffs, NJ: Prentice Hall.
ISBN: 0131103628.
http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0131103628.

# References II


Dowek, G. (2009).
*Principles of programming languages*.
Springer.
ISBN: 9781848820319.
http://books.google.com/books?id=1nfDlRj9aj0C.

WG14/N1124 (2005).
*C Reference Manual ISO ISO/IEC 9899:TC2*.
ISO/IEC.


You may also consult:
http://www.open-std.org/jtc1/sc22/wg14/,
http://en.wikibooks.org/wiki/C_Programming and
https://www.securecoding.cert.org

# References III

All examples from these slides are available, contact the author for more details.

# Outline

Part I

# Overview of the C Programming Language

# Outline

# Outline

# A little bit of history

C originated from the Bell Labs, closely tied to the development of the Unix operating system and the inability of existing language to access to low-level features proposed by the processor.

C emerged as a widely used language for building operating systems, and low-level programming.

Actually, C is both *portable* to several processor architectures, while offering *low-level constructs* to manipulate the processor. In some occasions, C is referred to as a *"portable assembly"* language.

# A little bit of history (cont'd)

C is now standardised at ISO, by the working group ISO/IEC JTC1/SC22/WG14, with various evolutions:

- "K&R" C, as Kernighan and Richie, is the authors attempts to define the structure and the semantics of the language in 1978;
- ANSI C, is the standardized version of C, after joint work by ANSI and ISO. This release is also known as C89;
- C99 (1999) clarified some aspects related to types;
- C11 (12/2011) is the latest release of the standard, and added multi-tasking as key features.

In this lecture notes, we will focus on *ANSI C*, with some emphasis on relevant elements of *C99*.

# C vs. C++

C++ emerged from early works from 1979 to extend C with the concepts of object-oriented languages (classes, inheritance, encapsulation, etc.).

C++ was first a superset of C with some extensions before becoming an independent language, defined in a separated standard. C++ took many elements from the C syntax, and changed some like one line comments using '//'.

**Note:** in some occasions, people mix the two languages. This can cause many difficulties when building the full application.

# Hello World!

Hello World is the canonical C example, exhibiting all C basic constructs

```c
#include<stdio.h>

int main (int argc, char **argv) {
  printf("Hello World!\n");

  return 0;
}
```

# Hello World!

Hello World is the canonical C example, exhibiting all C basic constructs

- *include* (libraries), main entrypoint, return value

```
#include<stdio.h>

int main (int argc, char **argv) {
  printf("Hello World!\n");

  return 0;
}
```

# Hello World!

Hello World is the canonical C example, exhibiting all C basic constructs

- *include* (libraries), main entrypoint, return value
- calling printf from stdlio.h

```
#include<stdio.h>

int main (int argc, char **argv) {
  printf("Hello World!\n");

  return 0;
}
```

# Compiling Hello World!

In the following, we suppose the source code of the "Hello World!" example is in file hello.c.
To compile it, we issue the following command[1]:

```
gcc -Wall -Werror -std=c99 -o hello hello.c
```

- -o hello is the name of the program to create;
- -std=c99 forces the use of the C99 rules;
- -Wall -Werror enforces stricter syntactic rules.

```
neraka:c hugues$ ./hello
Hello World!
```

[1] Assuming a GNU compilation toolchain

# Some notes on C

- C is a **compiled** language (as opposed to Python), without virtual machine (like e.g. Java);
- C comes with a specific compilation process for large systems, relying on a preprocessor, external libraries and dependences management;
- C is **case-sensitive** (foo and Foo are different);
- Comments start with "/*" and end with "*/", or start with "//" to the end of the line ($C++$ style);
- C is **weakly-typed**, meaning it is easy to mix types without explicit conversions.

# Outline

# Anatomy of a C program

Every C program is made of

- **identifiers**: named entities, e.g. variables, functions, types, labels. They must be defined prior to their use;
- **keywords**: reserved words defined by the standard, e.g. $\mathrm{int}$, $\mathrm{const}$, $\mathrm{default}$, ...
- **constants**
- **operators**: $+$, $++$, $>>$, ...
- **punctuations**

*Note:* comments are not seen by the C compiler, they are removed by the preprocessor.

# Identifiers

Identifiers are used to name entities: variables, functions, types, labels. They must be defined prior to their use;
An identifier is a set of characters made of

- letters (upper and lower case, no accent, special characters, etc.);
- digits;
- underscore ('_').

The first character cannot be a digit

# Keywords

Keywords are reserved by the language to convey particular concepts:

- memory: auto, register, static, extern, typedef;
- types: char, double, enum, float, int, long, short, signed, struct, union, unsigned, void;
- type qualifiers: const, volatile;
- control flow: break, case, continue, default, do, else, for, goto, if, switch, while;
- miscellaneous: return, sizeof.

# Expressions and instructions

An *expression* is a series of syntactically correct elements, that evaluates to a types
e.g. x=0 or $(i > 0)$ && $(i <= 42)$. They can be concatenated using ",".
An *instruction* is an expression completed by a ";" or a block delimited by "{" and "}".

```
int a, b, c;        /* integer variables */
b = 10, c = 32;
a = b + c;

if (b != 0) {
  a = c / b;
 }
```

# Definition of subprograms

Function implementations follow a regular pattern: function declaration (e.g. a header file), declaration of local variables and then instructions:

```
type_identifier function_identifier (parameters) {
  /* local variables */
  /* instructions */
}
```

*"type_identifier"* denotes the type returned by the function. If no value is returned, then this identifier must be void.

# A simple subprogram

```
/* Return the minimum value of two integers */
int min (int a, int b) {
  int result;

  if (a <= b)
    result = a;
  else
    result = b;

  return result;
}
```

*Notes:* In this example, we use blocks of only one expression, block delimiters are optional.

# About the main function

The main function has a default signature, where

- argc: denotes the number of parameters on the command line, starting from 0;
- argv: arrays of strings, one per argument;
- return value: 0 if the program exits correctly (or use EXIT_SUCCESS defined in stdlib.h)

```c
/* Print the list of arguments from the command line */
#include<stdio.h>
int main (int argc, char **argv) {
  int i;
  for (i = 0; i < argc; i++)
    printf ("argument #%d: %s\n", i, argv[i]);
  return 0;
}
```

# Outline

# C predefined types

C is a **weakly-typed** language: one can mix homogeneous types. i.e. a type defines both the size and the representation of a data in memory.

C defines a set of predefined types, using one of the following keywords:

- character type: char;
- integer types: int, short, long, unsigned;
- floating point types: float, double

# char: character type

The char type denotes a 8-bits type used to represent ASCII characters. char variables can be either signed (value in -128 .. 127) or unsigned (0 .. 255).

```
#include<stdio.h>
int main (int argc, char **argv) {
  unsigned char c = 'A';
  printf ("%d %c\n", c, c); /* Prints "65 A" */
  return 0;
}
```

*Note:* maximum values of all types are defined in limits.h. UCHAR_MAX and CHAR_MAX define the maximum values for unsigned char and char.

# ASCII table

The memory representation of characters is a digit, hence there is a correspondance between characters 'a' and a integer. An ASCII table provides one such representation, Unicode extends it to non-latin languages.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 32 | 0 | 48 | @ | 64 | P | 80 | ` | 96 | p | 112 |
| ! | 33 | 1 | 49 | A | 65 | Q | 81 | a | 97 | q | 113 |
| " | 34 | 2 | 50 | B | 66 | R | 82 | b | 98 | r | 114 |
| # | 35 | 3 | 51 | C | 67 | S | 83 | c | 99 | s | 115 |
| $ | 36 | 4 | 52 | D | 68 | T | 84 | d | 100 | t | 116 |
| % | 37 | 5 | 53 | E | 69 | U | 85 | e | 101 | u | 117 |
| & | 38 | 6 | 54 | F | 70 | V | 86 | f | 102 | v | 118 |
| ' | 39 | 7 | 55 | G | 71 | W | 87 | g | 103 | w | 119 |
| ( | 40 | 8 | 56 | H | 72 | X | 88 | h | 104 | x | 120 |
| ) | 41 | 9 | 57 | I | 73 | Y | 89 | i | 105 | y | 121 |
| * | 42 | : | 58 | J | 74 | Z | 90 | j | 106 | z | 122 |
| + | 43 | ; | 59 | K | 75 | [ | 91 | k | 107 | { | 123 |
| , | 44 | < | 60 | L | 76 | \ | 92 | l | 108 | \| | 124 |
| - | 45 | = | 61 | M | 77 | ] | 93 | m | 109 | } | 125 |
| . | 46 | > | 62 | N | 78 | ^ | 94 | n | 110 | ~ | 126 |
| / | 47 | ? | 63 | O | 79 | _ | 95 | o | 111 | | |

# ASCII special characters

Characters code below 32 have special meaning, here is a small list:

- 0x0, NUL, Null character, string termination
- 0x7, BEL, Bell
- 0x8, BS, Backspace
- 0x9, HT, Horizontal Tab
- 0xA, LF, Line Feed
- 0xD, CR, Carriage Return

They are defined with particular characters in C:
\n new line \r carriage return \t horizontal tab
\f new page \v vertical tab \a bell
The '\' character is used to escape,
e.g. **char** backslash ='\"';,
but also to represent characters in the form '\x<hexa-code>'
e.g. **char** c ='\xFF'; or '\<octal-code>'

# String constants

C does not have a native type for strings. Strings are nothing but an array of characters, defined as follow:

```
char a_string[] = "Hello World!";
```

**Note:** as a consequence, there is no language defined attributes for manipulating strings, one must use predefined functions, see string.h for more details.

```
/* from string.h */
char    *strcat(char *, const char *);
char    *strchr(const char *, int);
int      strcmp(const char *, const char *);
int      strcoll(const char *, const char *);
char    *strcpy(char *, const char *);
size_t   strlen(const char *);
```

# Integer types

They are defined over the interval $[-2^{n/2}..2^{n/2}-1]$, where n is the size of the type in bits; it is a multiple of 8.
C defines integer types: char, int, and modifiers:

- short int: smaller integer types;
- long int and long long int: larger integer types;
- unsigned [char|int]: shift range to $0..2^n-1$

Per construction, the size in bytes (returned by the operator sizeof) of all types respects:
*char* $<$ *short int* $\leq$ *int* $\leq$ *long int* $<$ *long long int*
See source/c/test_sizeof.c for more details.

# Printing strings

C is a low-level language, there is a no support for manipulating strings in the language. This is supported through dedicated libraries.

stdio.h is a *header file* that defines functions for input/output. printf is used to print a string.

```
printf ("control chain", exp1, exp2, ..);
```

where "control chain" is a string with special characters, exp-n the set of expressions to print.

# Printing strings (cont'd)

Control chain uses special characters to insert data to print:

```c
#include<stdio.h>

int main (int argc, char **argv) {
  printf ("Hello World!\n"); /* \n to flush the output */
  printf ("Universe constant %d\n", 42); /* print integer */
  printf ("Hexadecimal %x\n", 42); /* print integer */
  return 0;
}
```

# Printing strings (cont'd)

| format | conversion | data displayed |
|---|---|---|
| %d | int | signed decimal |
| %u | unsigned int | unsigned decimal |
| %o | unsigned int | unsigned octal |
| %x | unsigned int | unsigned hexadecimal |
| %f | double | floating point |
| %e | double | floating point, exp notation |
| %c | unsigned char | character |
| %s | char* | string |
| %p | void* | pointer |

The prefix "l" must be used for long int or double.

# C99 Integer types

*C*99 introduces integer types whose size is clearly defined, in file stdint.h

- int8_t, int16_t, int32_t, int64_t: signed integers of size 8, 16, 32, 64 bits;
- uint8_t, uint16_t, uint32_t, uint64_t:
  unsigned integers of size 8, 16, 32, 64 bits;
- size_t is the unsigned integer type of the result of the sizeof operator (ISO C99 Section 7.17.)
  The sizeof operator yields the size (in bytes) of its operand, which may be an expression or the parenthesized name of a type. The size is determined from the type of the operand. The result is an integer. The value of the result is implementation-defined, and its type (an unsigned integer type) is size_t (ISO C99 Section 6.5.3.4.)

# C99 Boolean types

Standard C did not include the definition of a boolean type until *C*99. Booleans are defined in stdbool.h.

```
#include<stdbool.h>

bool status = true;
```

Note: the canonical definition of 'false' is an integer constant whose value is 0, 'true' is 1

# Floating point types

C defines three floating point types, they usually follow IEEE754 definition for floating types:

$$(-1)^{sign} \times c \times b^{exponent}$$

where c is coefficient, b is usually 2.

- float: simple precision;
- double: double precision;
- long double: quadruple precision.

# Numerical constants

C allows the definition of numerical constants. Prefixes and suffixes are used to enforce a particular base for computation, and for defining precision:

```
int           integer = 123;          /* integer */
int           octal   = 0123;         /* octal (base 8) */
int           hexa    = 0xc00fee;     /* hexadecimal */
long          a_long  = 123456789L;   /* long integer */
unsigned int  u_int   = 1234U;        /* unsigned integer */
unsigned long int uli = 123456789UL; /* unsigned long integer */
double        a_double = 3.14159;     /* a double */
float         a_float  = 3.14156F;    /* a float */
long double   l_double = 3.14159L;    /* a long double */
/* The following is a GNU gcc extension */
int             binary = 0b1010;      /* binary number */
```

# Outline

# C operators

C is a **weakly-typed** language. The affectation operator '=' casts (converts) the value to the type of the left hand-side part of the expression.
Explicit cast can be enforced using "(type) expression"

```
#include<stdio.h>

int main(int argc, char **argv) {
  int i, j = 2;
  float x = 2.6, y = 2.7;
  i = x + y;                    /* i = 5 */
  j = (int) x + (int) y;        /* j = 4 */
  x = i + 2.6;                  /* x = 7.6 */
  printf("i = %d,  j = %d, x = %f \n",i, j, x);
  return 0;
}
```

# C operators (cont'd)

C uses typical symbols for operators: $+$, $-$, $*$, $/$.
The % sign is for modular division.
C has no operator for exponentiation. Similarly, typical mathematical functions are not built-in.

```
#include<math.h> /* mathematical functions */
#include<stdio.h>

int main (int argc, char **argv) {
  double s, s2;
  s = sin (M_PI / 4);
  s2 = sqrt (2.0) / 2.0;
  printf ("%f %f\n", s, s2);
  return 0;
}
```

# C operators and type conversion

**Warning**: C uses the same / operator for integer and float division. The type of the operands has an impact on the precision of the result:

```
#include<stdio.h>

int main (int argc, char **argv) {
   float x, x2;
   x = 3 / 2;      /* x = 1.0 */
   x2 = 3 / 2.0;   /* x = 1.5 */
   printf ("%f %f\n", x, x2);
   return 0;
}
```

# Type promotion

When performing computations, integer types smaller than int are implicitely converted to this type, otherwise to unsigned int

```
#include <stdio.h>

int main (int argc, char **argv) {
  signed char cresult, c1, c2, c3;
  c1 = 100; c2 = 3; c3 = 4;
  cresult = c1 * c2 / c3;
  printf ("%d\n", cresult); /* returns 75 */
  c1 = 100; c2 = 3; c3 = 4;
  cresult = c1 * c2;
  cresult  /= c3;
  printf ("%d\n", cresult); /* returns 11 */
  return 0;
}
```

# Integer Conversion Rank

- No two different signed integer types have the same rank, even if they have the same representation.
- The rank of a signed integer type is greater than the rank of any signed integer type with less precision.
- The rank of long long int is greater than the rank of long int, which is greater than the rank of int, which is greater than the rank of short int, which is greater than the rank of signed char.
- The rank of any unsigned integer type is equal to the rank of the corresponding signed integer type, if any.
- The rank of any standard integer type is greater than the rank of any extended integer type with the same width.
- The rank of char is equal to the rank of signed char and unsigned char.
- The rank of any extended signed integer type relative to another extended signed integer type with the same precision is implementation defined but still subject to the other rules for determining the integer conversion rank.
- For all integer types T1, T2, and T3, if T1 has greater rank than T2, and T2 has greater rank than T3, then T1 has greater rank than T3.

# Rules for type promotion

1. If both operands have the same type, no further conversion is needed.
2. If both operands are of the same integer type (signed or unsigned), the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
3. If the operand that has unsigned integer type has rank greater than or equal to the rank of the type of the other operand, the operand with signed integer type is converted to the type of the operand with unsigned integer type.
4. If the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, the operand with unsigned integer type is converted to the type of the operand with signed integer type.
5. Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type. Specific operations can add to or modify the semantics of the usual arithmetic operations.

# Type promotion example

"Rules for type promotion are not logical, there are the rules, period." (from Internet).

```
#include <stdio.h>
int main (int argc, char **argv) {
  int si = -1;
  unsigned int ui = 1;
  printf("%d\n", si < ui);
  printf("%d\n", si < (int)ui);
  return 0;
}
```

**Exercise:** explain the result displayed.

# C relational and binary operators

Relational operators: $>$, $>=$, $<$, $<=$, $==$, $!=$
**Warning:** A typical error is to mix "=" (affectation) and "==" (test).
Logic operators: && (logic AND), || (logic OR), ! (negation)
**Note:** the default value for true is 1, false is 0.

```
int x = (i >= 0) && (i <= 9) && !(i == 5);
```

# Other operators

C defines short operators: $++, --$

- a++; is equivalent to a = a + 1;
- These operators can be used as *suffix* or *prefix*
  a=1, c = ++a; is equivalent to a++, c = a;
  a=1, c = a++; is equivalent to c = a, a++;

$+=, -=, *=, /=, \%=$

- a $*$ = 10; is equivalent to a = a $*$ 10;

# Ternary operator

The ternary operator can be used for one-liner expression evaluation:

**unsigned char** bit = ( x % 2 == 0) ? '0' : '1';

which reads: "if x modulo 2 equals to 0, then return the character '0', else returns '1'."

**Note:** this notation is compact, and should be used for simple expression evaluation only, e.g. computing a min or max value.

# Outline

# Control flow instructions

C proposes control flow instructions similar to those from Java:

- conditional branching: if else,
- multiple branching: switch case,
- loops: while, do/while, for,
- unconditional branching: break, continue, goto

# Conditional branching: if else

The generic form of an if/else instruction is

```
if ( expression-1 )
  instruction-1
else if ( expression-2 )
  instruction-2
/* .. */
else
  instruction-n
```

```
/* shorter variant */
if ( expression-1 )
  instruction-1
```

where "expression-k" is a boolean expression, and "instruction-k" one instruction, or a block.

# Multiple branching: switch case

The generic form of a switch/case instruction is

```
switch ( expr ) {
  case constant-1: /* if expression == constant-1 */
    instruction-1
    break; /* mandatory, else executes
              the following block */
  case constant-2:
    instruction-2
    break;
  default:
    /* executed if expr does not match any constant */
    instruction-n
    break;
}
```

# while and do/while loops

while and do/while loops differs in semantics:

```
while (expression)
  instruction
```

is executed as long as expression in true. It may not be executed at all, whereas

```
do
  instruction
while (expression)
```

is always executed at least once.

# for loops

A for loop has the following form

```
for ( expr 1 ; expr 2 ; expr 3)
  instruction
```

it is equivalent to

```
expr1;
while (expr2) {
  instruction;
  expr3;
}
```

# Example

```c
/* Sieve Of Eratosthenes: find prime numbers less than SIZE */

#include <stdio.h>

#define SIZE 100
static int sieve[SIZE]; /* By default, array initialized to 0 */

void eratosthenes (void) {
  int i,j;
  for (i=2; i*i <= SIZE; i++)
    if (!sieve[i])     /* i-th entry is a prime number */
      for(j = i+i; j < SIZE; j += i)
        sieve[j] = 1; /* non-prime are multiple of i */
}
```

# Unconditional branching: break, continue, goto

These instructions are used to control the execution of loops:

- break: stop the execution of the inner loop;
- continue: stop the execution of the current step in a loop, and execute the next iteration;
- goto: jump to a particular label in source code. Usually **forbidden** by coding guidelines.

# Part II

# C Advanced topics

# Outline

# Outline

# A digression: coding versus engineering

*"Coding is the process of transforming the comprehensible into the incomprehensible, and is relevant only where machines are programmed in less abstract (or less meaningful) terms [..]"* from *"The Word 'Coding' Considered Harmful"* by Brian Tooby.
Coding is often used for software-related activities, it should not be considered this way: large systems relies on software to achieve their mission. Hence, *engineering* it correctly is important. This means

- select proper style guidelines: naming conventions;
- design the architecture: types hierarchy, libraries;
- testing strategies, ...

# Style guidelines

We spend 90% of the time *reading* source code. Style guidelines are important to ease reading and navigation:

1. one instruction per line, ';' being the last character;
2. the layout of the program should be obvious, braces should be alone on a line, or the last character on a line;
3. use text editors features to indent source;
4. there should be a whitespace character between keywords, and the following '(', and binary operators;
5. there is no whitespace between a unary operator and the operand.

# Code quality

*"Quality and Assessment"* of code is an important aspect of the software engineering cycle. Quality stems from

- proper type and function definitions, well isolated in modules;
- proper documentation to remove ambiguity in function semantics;
- proper testing of all artifacts.

All these elements are detailed in depth in standards for building critical systems like DO-178B (avionics), ECSS-E-40-B (space), ...

Example of C coding guidelines from NASA's JPL:
http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf

# Outline

# User-defined types

From basic types (integer, float and character), one can build more complex types. Usually, these are defined to match a particular problem space. C proposes:

- static arrays[2];
- structures: aggregates of types;
- enumerations: lists of tags;
- union: multiple views on the same area of memory.

In addition, the user can name its types.

[2] dynamic arrays are discussed with pointers

# Static arrays

The general form for declaring static arrays is
type id[const_size]
where type is the type of an elements, id the name of the type and const_size a constant integer denoting the number of elements in the array.
By convention, elements indexes are 0..*const_size* − 1.

```
void f (void) {
  int i;
  unsigned char tab[10]; /* tab is an array of size 10 */

  for (i = 0; i < 10; i++) /* indices are 0 to 9       */
    tab[i] = i;            /* affecting values to tab  */
}
```

# Static arrays (cont'd)

**Note**: Using [] for manipulating arrays is a handy notation. But in C, they do not define a type.

- **int** tab[10]; simply allocates storage space to store 10 integers. tab is an identifier to refer to this area.
- the = or == operators do not operate on the content of tab, but to the *address* of this storage space.

**Note:** specific routines for copying or comparing arrays must be used, like memcpy(), memcmp().

# Structures

The definition of structures is simple:

```
#include<math.h>

struct point {    /* definition of a struct :                    */
  double x, y;    /* two doubles: x, y                           */
};

struct point origin = { 0., 0. };          /* a variable          */

double norm (struct point p) {
  return (sqrt (p.x * p.x + p.y * p.y)); /* manipulation          */
}
```

# Enumerations

An enumeration is a type defined by an ordered list of literals:

```
#include<stdio.h>

enum days { sun, mon, tue, wed, thur, fri, sat };

int main (int argc, char **argv) {
  enum days a_day = sun;
   printf ("%d\n", a_day);  /* prints "0" */
   return 0;
}
```

Per construction, a literal is mapped to an integer.

By default, the first value is 0, successor is 1, etc.

# Unions

A union is a set of variables all set at the same memory position. Each element of the union is a distinct view on the same address:

```
#include<stdio.h>

union foo { int a; float b; };

int main (int argc, char **argv) {
  union foo bar;
  bar.b = 42.0;
  printf ("%d %f\n", bar.a, bar.b);
  /* prints "1109917696 42.000000" */
  return 0;
}
```

# typedef: naming a type

To ease writing of program, one can use typedef to define an *alias* to a type, like:

```
struct _point { float x,y; };

typedef struct _point point;

point a_point = { 0.5, 1.0 };

int main (int argc, char **argv) {
  a_point.x = 3.14;

  return 0;
}
```

# Outline

# Object value and address

In C, an entity is defined by its *address* and its *value*.

- the $\&$ operator returns the address of a variable,
- $<\text{type}>$ *$<\text{id}>$; defines a *pointer* variable. A variable whose content is the address of another variable of type $<\text{type}>$,
- the * operator returns the content pointed by a pointer variable.

Manipulating pointers is useful for accessing memory area through aliases. They are mandatory for many C idiomatic constructs: arrays, parameter passing (structs, out mode), and pointers to functions.

# Manipulating object and pointers

```c
#include<stdio.h>

int main (int argc, char **argv) {
  int i = 42, *p;
  p = &i;
  printf ("i : %d %p\n" , i, &i);
  printf ("p : %p %p\n" , p, &p);
  printf ("*p : %p %d\n" , p, *p);
  /* printf (on a 64bit machine)
     i : 42 0x7fff5fbff8ec
     p : 0x7fff5fbff8ec 0x7fff5fbff8e0
     *p : 0x7fff5fbff8ec 42 */
  return 0;
}
```

# Memory allocation

Memory is divided into different segments depending on their usage:

- Variables local to a subprogram are allocated on the *stack*. This memory is reclaimed when exiting the subprogram.
- Global variables are allocated on the *data segment*.
- To handle requirements for dynamicity in memory usage (e.g. creation/destruction of requests), one will allocate memory in the heap.

# malloc() and free()

C has no automatic memory management capabilities.

- malloc() allocates a chunk of memory;
- free() releases this part.

```
#include <stdlib.h> /* for malloc() */
#include <stdio.h>
int main (int argc, char **argv) {
  int *memory = malloc (100 * sizeof (int)); /* allocate
*/
  printf ("Memory allocated at %p\n", memory);
  free (memory); /* free allocated memory
*/
  return 0;
}
```

**Warning:** be careful not to introduce memory leaks.

# About free()

free(p) deallocates the memory pointed by *p but does not change the value of p. A good practice is to always set the value of p to NULL:

```
free (p);
p = NULL;

#define FREE(x) free(x) ; x = NULL
```

See funny videos: Pointer fun with Binky at Stanford: http://cslibrary.stanford.edu/104/

# Pointers arithmetics

A pointer is the address of a variable in memory. It is represented as an integer (or long integer depending on the CPU).
Pointers support arithmetics operations.
Let p be a pointer to $\mathrm{type}$, i an integer:

- the address of p= p + i; is
  $\mathrm{p} + \mathrm{i} * \mathrm{sizeof}(\mathrm{type})$,
- operators -, $++$, $--$, $==$ are also defined,
- the $\mathrm{NULL}$ macro defines an invalid pointer. This is the default value of all pointers.

# Arrays and pointers

In C, arrays are just a syntactic convention. An array is a memory area storing n items of the same type, it is therefore equivalent to a constant pointer whose value is the first element of the array.

Hence, p[i] = *(p+i);

Allocating a dynamic array is equivalent to allocating the corresponding chunk of memory using malloc():

**int** *tab = (**int** *) malloc (n * **sizeof** (**int** ));

# Multi-dimension arrays

Arrays of dimension 2 or more follow the same logic: the array is split as an array of arrays, like in the following

```
#include<stdlib.h>
int main (int argc, char **argv) {
  int i, **array;

 array = (int **) malloc (10 * sizeof (int *));
 for (i = 0; i < 10; i++)
   array[i] = (int *) malloc (10  * sizeof (int ));

 for (i = 0; i < 10; i++)
   free (array[i]);
 free (array);

 return 0;
}
```

# Strings and pointers

A string is an array of characters, hence a pointer. The ANSI/C convention for strings is that a string is an array terminated by the character '\0' (NUL).

```
#include <stdio.h>

int main (int argc, char **argv) {
  int i;
  char *string = "Hello World!";

  for (i = 0; *string != '\0'; i++)
    string++;

  printf ("%s has %d characters\n", string - i, i);
  return 0;
}
```

# Miscellaneous and pointers

Some final words about pointers:

- Structures: Let p be a pointer on a structure, then dereferencing one member is $(*p).member = p->member;$
- Recursive types, like linked lists are built this way:

```
struct cell
{
  int value;
  struct cell *next;
};

typedef struct cell *list;
```

# Duff's device

The Duff's device is a scary C program, made of several optimizations and freedom from the language.
The *"device"* came from an optimization problem of a copy loop, basically

```
void copy (char *to, char *from, int count) {
  do {                                /* count > 0 assumed */
    *to++ = *from++;
  } while (--count > 0);
}
```

# Duff's device (cont'd)

**Question:** Why does the following work?

```
void duff_device (char *to, char *from, int count) {
  int n = (count + 7) / 8;
  switch (count % 8){
    case 0:      do{       *to++ = *from++;
    case 7:                *to++ = *from++;
    case 6:                *to++ = *from++;
    case 5:                *to++ = *from++;
    case 4:                *to++ = *from++;
    case 3:                *to++ = *from++;
    case 2:                *to++ = *from++;
    case 1:                *to++ = *from++;
    } while (--n > 0);
  }
}
```

# Outline

# Definition of subprograms

Function implementations follow a regular pattern: function declaration (e.g. a header file), declaration of local variables and then instructions:

```
type_identifier function_identifier (params) {
  /* local variables */
  /* instructions */
}
```

*"type_identifier"* denotes the type returned by the function. If no value is returned, then this identifier must be void.

# A simple subprogram

```
/* Return the minimum value of two integers */
int min (int a, int b) {
  int result;

  if (a <= b)
    result = a;
  else
    result = b;

  return result;
}
```

*Notes:* In this example, we use blocks of only one expression, block delimiters are optional.

# Rules about subprograms

Subprogram (and types) must be defined prior to being used.
The definition of a subprogram (*prototype*) is similar to its implementation:

```
type_identifier function_identifier (parameters);
```

If the implementation comes before its use, the prototype is not mandatory.
**Warning:** some (old) compilers may not warn if the prototype is not defined, and will use default parameter promotion: integers arguments are cast to $\mathrm{int}$, all floating types are cast to $\mathrm{double}$.

# Parameter passing in C

In C, parameters are passed by *by-copy*, they are pushed on the stack by the caller, and popped by the callee.
*By-reference* semantics is achieved using pointers:

```
#include<stdio.h>
void swap(int *a, int *b)
{
  int temp;
  temp = *a; *a = *b; *b = temp;
}
int main(int argc, char **argv)
{
  int a = 42, b = 51;
  swap (&a, &b);
  printf("%d %d\n", a, b);
}
```

# Pointers to functions

Pointers can also point to functions, that is the address of its implementation in memory.
Pointers to functions are useful to defer binding to a particular processing,

```
int heapsort
  (void *base, size_t nel, size_t width,
   int (*compar)(const void *, const void *));
```

compar is a pointer to a comparison function used in the heapsort() function.

# Variadic functions

Functions with multiple parameters (like printf can be defined this way:

```
#include <stdio.h>
#include <stdarg.h>
int add (int nb,...) {
  int res = 0, i;  va_list parameters;
  va_start(parameters, nb);
  for (i = 0; i < nb; i++)
    res += va_arg(parameters, int);
  va_end(parameters);
  return(res);
}
int main (int argc, char **argv) {
  printf("%d \n", add(4,10,2,8,5));
  return 0;
}
```

# About the main function

The main function has a default signature, where

- argc: denotes the number of parameters on the command line, starting from 0;
- argv: arrays of strings, one per argument;
- return value: 0 if the program exits correctly (or use EXIT_SUCCESS defined in stdlib.h

```
/* Print the list of arguments from the command line */
#include<stdio.h>
int main (int argc, char **argv) {
  int i;
  for (i = 0; i < argc; i++)
    printf ("argument #%d: %s\n", i, argv[i]);
  return 0;
}
```

# Part III

# C Library

# Outline

# Outline

# About C libraries

Libraries have been defined as a handy way to

- package a set of functions as a collection of object and header files, ready to be used, e.g. basic C functions;
- hide implementation code while providing particular functions: e.g. Windows DLL

# Getting help

The installation of a C toolchain on a Unix system follows canonical rules:

- /usr/include stores standard header files;
- the man utility returns a formatted help on a function
  - man -k <function> to look for a function
  - man [-s <section>] <function> to look for its description.

# Text I/O

printf(), scanf() are used to print or parse strings, following particular formatting conventions for formatting:

```
#include<stdio.h>

int main (int argc, char **argv) {
  int age;
  printf ("Type your age\n"); /* \n to flush the output */
  scanf("%d", &age);          /* read from the stdin    */
  printf("you typed : %d\n", age);
  return 0;
}
```

See also: puts(), getc(), getchar().

**Note:** scanf() can be unsafe if used for scanning strings.

# File and buffered I/O

Variants of $\mathrm{printf}()$, $\mathrm{scanf}()$ exist for

- Buffers: $\mathrm{sprintf}()$, $\mathrm{sscanf}()$: operate on a allocated buffer.
- Fixed-size buffers: $\mathrm{snprintf}()$, $\mathrm{snscanf}()$: operate on a fixed-size allocated buffer.
- Files: $\mathrm{fprintf}()$, $\mathrm{fscanf}()$ They operate on opened files for reading, printing.

# File management

fopen(), fclose() provide basic file management.

```
#include<stdio.h>

int main (int argc, char **argv) {
  FILE *f;
  f = fopen ("test.out", "w");
  /* "w" indicates mode, here write */
  fprintf (f, "Hello World!\n");
  fclose (f);
  return 0;
}
```

*Note: all OS services like chown, chmod, etc are also available.*

# Checking assertions at run-time

Assertions are expressions evaluated at runtime, in case debug options are activated.

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

int main (int argc, char **argv) {
  int *ptr = malloc(sizeof(int) * 10);
  assert(ptr != NULL); /* assert ptr is correctly set */

  return 0;
}
```

*Note: useful for validating pre/post conditions on source code, based on a detailed specification of the function*

# String Handling

Strings are **not** first class citizen of the C language. Dedicated functions implement basic manipulations:

- strcat() concatenate two strings
- strchr() string scanning operation
- strcmp() compare two strings
- strcpy() copy a string
- strlen() get string length
- strncat() concatenate one string with part of another
- strncmp() compare parts of two strings
- strncpy() copy part of a string
- strrchr() string scanning operation

# Time management

time.h defines a set of functions for manipulating time, performing arithmetics and printing:

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

int main(void)
{
    time_t timer = time(NULL);
    printf("Time of the day %s\n", ctime(&timer));
    return 0;
}
```

# Math library

math.h defines a full list of mathematical operators. One should carefully check precision of arguments, conventions used, ...

E.g. round(), lround(), llround(), sin()

*Note:* in some environments, performing floating point operations require a specific math library and/or hardware support.

# Other libraries

Other libraries may be defined, e.g. to encapsulate specific concerns, some of which being standardised:

- libpthread: POSIX concurrency;
- libncurses: text-based GUI;
- libcrypto: cryptography;
- libraries for GUI, database, network, ...

On a Unix-based systems, libraries are installed in /usr/lib, header files in /usr/include.

# Part IV

# C toolchain

# Outline

# Outline

# Steps in compiling a C program

Compiling a C program means turning a set of text files into an executable. This is a multi-steps process made of:

1. Preprocessing of file: executing statements associates to #include, #ifdef, etc.;
2. Compilation of each file: building an object file for each compilation unit;
3. Linking phase: combining object files + prologue to build the binary.

Following Unix philosophy, each step is supported by a dedicated tool.

# About the GCC toolchain

The *GNU Compiler Collection* (aka. gcc) is a set of front-ends (supporting C, C++, Ada, Java, Fortran, ...) and back-ends (producing code for various processors).
It is part of the open source movement initiated by the Free Software Foundation.
It is the default compiler on all Linux platforms, and used on many others. It is also used for many embedded RTOS like RTEMS or VxWorks.

# Compiling Hello World!

In the following, we suppose the source code of the "Hello World!" example is in file hello.c.
To compile it, we issue the following command:

```
gcc -Wall -Werror -std=c99 -o hello hello.c
```

- -o hello is the name of the program to create;
- -std=c99 forces the use of the C99 rules;
- -Wall -Werror enforces stricter syntactic rules.

```
neraka-2:c hugues$ ./hello
Hello World!
```

# Getting help

The installation of a C toolchain on a Unix system follows canonical rules:

- /usr/include stores standard header files;
- the man utility returns a formatted help on a function
  - man -k <function> to look for a function
  - man [-s <section>] <function> to look for its description.

# Outline

# Programming in the large

*"Programming in the large"* refers to defining and implementing large software-based systems made by a team. It involves splitting a system into modules with well-defined interfaces and boundaries. In the case of C, this implies splitting code base into a set of compilation units made of:

- One implementation file (e.g. $\mathrm{hello.c}$) that implements a set of functions;
- A set of header files (e.g. $\mathrm{stdio.h}$) that defines functions, types, etc.

# Programming in the large (cont'd)

One and only one of them shall define the $\mathrm{main}()$ function.
All these compilation units are compiled as $\mathrm{object}$ files, then $\mathrm{linked}$ together to form the final program
Compilation units allow for $\mathrm{separate}$ compilation. If one unit is changed, re-building the whole program is reduced to recompiling only the unit that changed.
This provided a significant speed-up in the early days of programming.
Dependency tracking is usually done through a $\mathrm{makefile}$.

# About the preprocessor

The role of the preprocessor is to prepare the source files to be compiled.
Directives are used to insert or remove source code.
All directive are prefixed by #, like #include

- #include insert the content of a file, e.g. a header file;
- #define to define a string constant;
- #ifdef/#ifndef, #else, #endif for conditional compilation.

# Preprocessor example

```
/* Define booleans */
#ifndef __BOOL_H_
#define __BOOL_H_
#if __STDC_VERSION__ >= 199901L
/*  We are using a C99 compiler, nothing to do */
# include<stdbool.h>
#else
# define bool    unsigned char;
# define false   (bool)0
# define true    (bool)1
#endif /* __STDC_VERSION__ */
#endif /* __BOOL_H_ */
```

The use of #ifndef is mandatory to ensure that at most one copy of the header file is embedded.

# About #define

The macro #define is used to define a token that will be substituted in the source code; for instance the constant size of an array, like

**#define** SIZE 10

**Note** there is no ';' at the end, as the token (SIZE) will be changed to 10.

Another usage of this macro is for one-liner functions:

**#define** MIN(X,Y) ((X) < (Y) ? : (X) : (Y))

# Outline

# Compiling large C code

The compilation of a program, made of several .c files (or *compilation units*) is a multi-step process:

- Compile each .c file to build an object file:

  ```
  $ gcc -Wall -Werror -std=c99 -c file1.c
  $ gcc -Wall -Werror -std=c99 -c file2.c
  ```

- Link all object files together:

  ```
  $ gcc -o prog file1.o file2.o
  ```

If only one unit changed, one can reduce re-compilation time. Yet, performing these steps is error-prone, and can be automated using a *Makefile*.

# Makefile

A *makefile* defines the set of steps to build files, it can be either a program, a PDF file, ...
Makefiles are processed by the make utility part of Unix, and most compilation chains (e.g. Cygwin, mingw for Windows).
Makefile defines *targets*, and for each target a set of *rules* to build them.
Each rule is a set of calls to program, including shell script, that produce intermediate files.

# Anatomy of a makefile

Each makefile rule follows the same pattern
*target*: *[dependences]*
**TAB***command1*
**TAB***command2*
whenever *dependences* is newer than *target*, then *command1*, *command2* are executed sequentially.
**Important:** the TAB character shall be the first character of each rule.
make builds a tree of dependences, and rebuilds only intermediate targets for which dependences have been updated since last invocation.

# Example of a Makefile

```
CC = gcc
CFLAGS = -Wall
BINARIES = hello

all: $(BINARIES)

$(BINARIES):
	$(CC) $(CFLAGS) -o $@ $@.c

clean:
	-rm -rf *.o
distclean: clean
	-rm -rf $(BINARIES)
```

# Example of a Makefile (cont'd)

hello is the first target, it is the default target when invoking only *"make"*.
clean is used to clean intermediate generated files, executed when invoking *"make clean"*.
distclean is used to clean all files, it executes first the clean target, then perform additional clean up.

# Macros

Macros are useful to define reusable elements of a makefile, like

```
CC = gcc            # name of the compiler
CFLAGS = -Wall -O2 # compilation flags
BINARIES = foo bar # binaries to build
all: $(BINARIES)
$(BINARIES):
        $(CC) $(CFLAGS) -o $@ $@.c
```

Here, the special macro $@ refers to the file to be built.
$< to the input file (first dependence).

# About makefile

Mastering makefiles is an important process to compile large C programs, like e.g. gcc itself.
One difficulty when defining a makefile is to have the correct list of dependencies. This is usually controlled by external tools that will build makefiles, like the IDE in use (such as AVR-Studio), or part of the GCC toolchain like autoconf, automake, ....
This goes well beyond the scope of these lectures notes.

# Outline

# Why debugging?

> *As soon as we started programming, we found to our surprise that it was not as easy to get programs right as we had thought. Debugging had to be discovered. I can remember the exact instant when I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.*

Maurice Wilkes, 1949.

# About the debugger

When executing a program, the operating system allocates some resources to run the program in a process (instance of a running program).

A debugger allocates a particular execution environment to control the execution of a program: set breakpoints, analyze the list of calls, examine function parameters or the memory, etc.

To ease interaction and reporting, the program must be compiled with particular arguments to store position of each instruction. This is often referred to the *debug mode*.

# Using gdb

gdb is the GNU debugger, to be used with gcc.
To debug a C program, you must add the '-g' flag:

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

Then, start debugging session with gdb using

```
$ gdb hello
```

# Using gdb (cont'd)

We use the test from source/c/test_gdb.c

```
neraka-2:c hugues$ gdb ./test_gdb
(gdb) r -div
Starting program: test_gdb -div
Program received signal EXC_ARITHMETIC, Arithmetic exception.
0x0000000100000d66 in test_zerodiv () at test_gdb.c:29
29  printf ("%d\n", 1/x);
(gdb) bt
#0  0x0000000100000d66 in test_zerodiv () at test_gdb.c:29
#1  0x0000000100000dda in main (argc=2, argv=0x7fff5fbff510) at test_gdb.c:41
(gdb) print x
$1 = 0
(gdb) quit
```

# Using gdb (cont'd)

gdb has several functions to ease debugging

- break ('b') to place a breakpoint, the execution stops there
- continue ('c') to continue after a breakpoint
- next ('n') and step ('s') are used for step-by-step execution; step will reveal the execution of subprograms;

Advanced functions are available for a better control of breakpoints management. See http://www.gnu.org/s/gdb/ for more details.

# Other tools

The C toolchain may come with additional tools to

- Evaluate coverage: is every statement executed at lease once? gcov provides coverage analysis.
- Profiling: how often is a piece of code evaluated? gprof builds profile reports that can be used to tune some compilation parameters like inlining.

These tools are usually embedded in IDE, or called through scripts that produce HTML outputs like lcov.

# Part V

# C for embedded sytems

# Outline

# C & embedded systems

C has been defined as a high-level language (compared to assembly language) to implement advanced programs, but also as a low-level one (compared to e.g. Pascal) to be close enough to the bare machine.
C has some distinctive features to support low-level programming.
We review some of the in the following.

# Outline

# Bit-level operator

C has specific operators for bit-level operations:

& AND | OR ^ XOR

~ complement to 1 « left shift » right shift

Operators &, | and ^ follow typical truth tables;

~ change the value of each bit to the opposite value;

Left shift (resp. right) is equivalent to a multiplication (resp. division) by 2.

C defines the corresponding composite operators:

&=, ^=, |=, «=, »=

# Bit-level operator: example

$$
\begin{aligned}
\text{a} &= 01001101 : 77 \\
\text{b} &= 00010111 : 23 \\
\text{a \& b} &= 00000101 : 5 \\
\text{a} \mid \text{b} &= 01011111 : 95 \\
\text{a \^{} b} &= 01011010 : 90 \\
\tilde{}\text{a} &= 10110010 : 178 \\
\text{b} << 2 &= 01011100 : 92 \\
\text{b} << 5 &= 11100000 : 224 \\
\text{b} >> 1 &= 00001011 : 11
\end{aligned}
$$

See source/c/test_bits.c for more details.

# Bit fields

Bit fields are used to map a particular structure onto a memory area. C provides representation clauses to align members to a specified number of bits.

```
typedef struct {
  unsigned int reg_status : 1;  /* field of 1 bit */
  unsigned int data : 4;
  unsigned int padding :3; /* 3 bits of padding */
} _bitfield ;

void f (void) {
   _bitfield field;
   field.reg_status = 1;
   field.data = 0x0b;
}
```

# Enumerations representation clause

Enumerations are types representing particular discrete values.
They are mapped to integer values, computed from the position of the literal.
This value may be forced, for instance:

```
enum status {
   on = 0x01,
   off = 0x00
};
```

Such representation clause may be used for mapping states of an IC to binary values.

# Outline

# register, const and volatile

C defines several storage classes to instruct how to treat variable or parameter definitions:

- register: tells the compiler to store the variable being declared in a CPU register, e.g. **register int** x = 42;
- const: makes variable value or pointer parameter unmodifiable, e.g. **const** x = 42;
- volatile: indicates that a variable can be changed by a background routine, for instance an ISR. e.g. **volatile** i = 10;.

static variables are mapped onto a permanent area of memory (data segment). This area is initialized at zero at program start up.

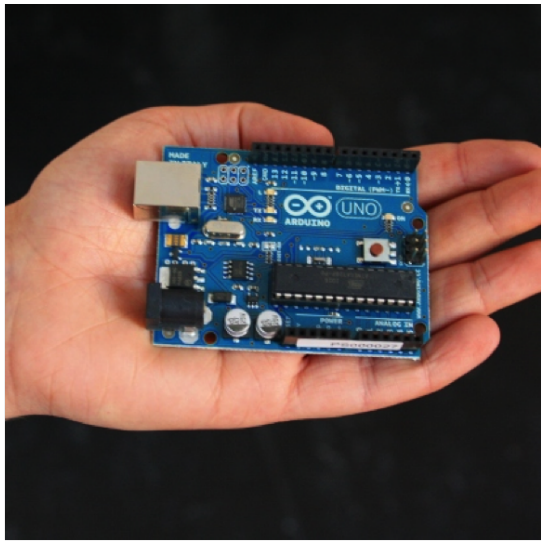
A nice effect of static variable is that they can serve as permanent storage for variable, like in

```
#include<stdio.h>
void f(void) {
  static int storage; /* initialized to 0 */
  printf ("storage = %d\n", storage++); /* varies from 1 to 10 */
}
int main (int argc, char **argv) {
  int i;
  for (i = 0; i < 10; i++) f();
  return 0; }
```

# Outline

# About the Arduino AVR board

The Arduino board is an "Open Hardware" platform based on the AVR 8-bit micro-controller.
It is a small CPU: 16Mhz, 32KB for code, 1KB of SRAM, with many interfaces: UART, I2C, SPI, GPIOs, PWM, ...




The Arduino is used for many projects: monitoring temperature, tank of a car, performing nice effects, and even controlling UAVs

# LED Blinking

```
#include<avr/io.h>
#include<util/delay.h>

int main (void)
{
  /* Initialization part                          */
  DDRB |= (1 << DDB5); /* set PB5 for output*/
  /* Main infinite loop                           */
  while (1) {
    PORTB |= (1 << PORTB5); /* set PB5 high */
    _delay_ms (1000.0);
    PORTB &= ~(1 << PORTB5); /* set PB5 low */
    _delay_ms (1000.0);
  }
  return 1;
}
```

# Part VI

# Conclusion

# Outline

# Outline

# Conclusion I

These lectures notes covered basics of the C language. Many elements are not part of the C language itself, and dedicated to external libraries:

- concurrency: flows of computations (threads), synchronization mechanisms (mutexes, conditional variables, ...) are either standardized (POSIX, ARINC653) or ad hoc (RTEMS, VxWorks);
- math functions: large numbers, advanced mathematics (e.g. matrices) are covered by GPGPU libraries (CUDA), OpenMP (cluster computing);
- distribution: Remote Procedure Calls, Distributed Objects, Shared Memory are covered by dedicated frameworks like CORBA, DDS, ...

# Conclusion II

All these technologies rely heavily on the C language, and associated tools to build large programs.
We will cover (some of) them in other courses ;)