# Programming and Algorithms

## C: overview and basic elements of programming.

Jean-Bernard Hayet

CIMAT, A.C.

# C: overview

# C: Some history


- Conceived in 1972 by D. Richie and K. Thompson (Bell Labs).
- Successor of B.
- Developed to develop Unix!
- Formalized by B. Kernighan/ D. Richie in "The C Programming language" (K&R) in 1978.
- Becomes dominant in the 1980s (replacing Fortran).

# C: Normalization

- Normalized by ANSI (American National Standards Institute) 1983-1989 (C89): **ANSI C**.
- Norma retaken by ISO (International Standards Organization) in 1990 (**C90**).
- In 1999: **C99** (changes in preprocessor, new keywords, declarations nearly anywhere, inline, variable-size arrays).
- In 2011: **C11** Support for multi-threading. gets removed. Anonymous structures/unions.
- **C18** is very close to **C11**.

Check and tune your compiler to C11!

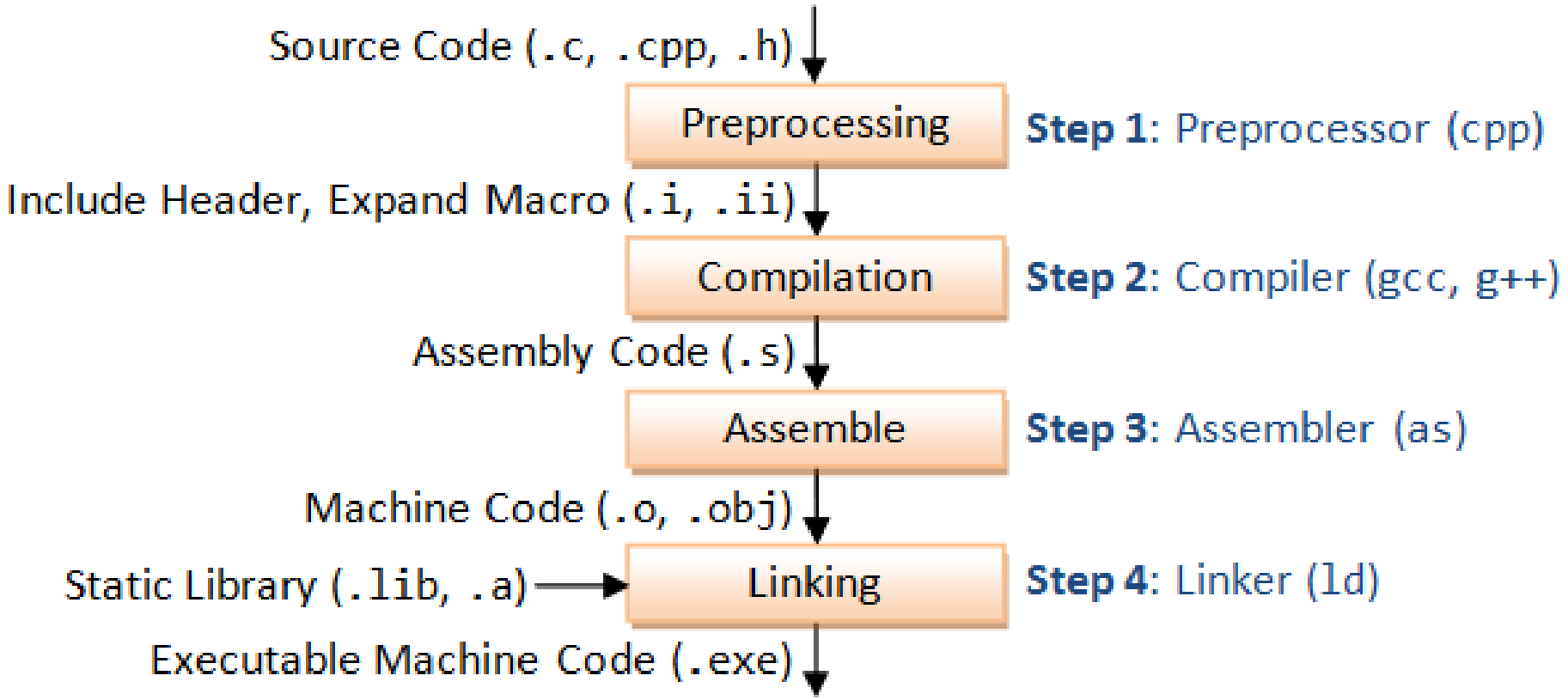
# C: Main features

+ Among the "high level" languages, it is a rather "low level" language.
+ Efficient.
+ Adapted to **system operations**, accesses to memory, control at low level (inherited from the context of its creation).
+ Freedom for the programmer and much control.
+ Huge community, lot of support.

# C: Main features

- Require some efforts and vigilance from the programmer (memory liberation, index checking. . . ).
- The language itself or the standard library do not offer many high level features (e.g., handling strings).
- The standard library is relatively limited.
- Not very adapted to most modern programming paradigms.

# C: The compiler



# C: The compiler

Code is organized into code units/files:

- the **header** files (.h) give function prototypes (declarations).
- the **code** files (.c) give function definitions.

# C: The compiler

Ex: with the files functionCode.c, functionCode.h, main.c

```
gcc -c functionCode.c -o functionCode.o
gcc -c main.c -o main.o
gcc main.o functionCode.o -o test
```

or

```
gcc functionCode.c main.c -o test
```

# C: Language elements

- Identifiers: names of variables, structs, labels, functions.
- Keywords: type names, control structures, type qualifiers, storage qualifier.
- Constants/macros from the preprocessor (#DEFINE).
- Strings.
- Operators: +,*. . .
- Punctuation.
- Comments : /* Comment */ (or // Comment from C99)

# C: Expressions

- **Instructions** are sequences of expressions syntactically correct, that end with a semi-colon.
- Variable declaration:

```
int a ;
double d ;
```

- Definition and use of variables:

```
a = 1;
d = 2.1*a+a*a ;
```

# C: Defining integer constants

**Decimal**

```
int x=100;
```

**Octal** (introduced by 0)

```
int x=0144;
```

**Hexadecimal** (introduced by 0x)

```
int x=0x64 ;
```

One can use suffixes for long (l or L) or unsigned (u or U).

# C: Defining integer constants

```
int x = 09;
```

```
main.c:7:11: error: invalid digit "9" in octal constant
```

# C: Defining floating constants

- Representation mantissa-exponent.
- Default: double.
- Can be modified in long double (L) or float (F).

```
12.34
12.3e-4
12.34F
12.34L
```

(long double: in general, 80 bits)

# C: Defining functions

```
type myFunction(type1 arg1, type2 arg2, ...) {
        declarationofinternalvariables
        instructions
        return somethingoftypetype;
}
```

They are the structuring elements of programs.

# C: Defining functions

Example:

```
int computeSum (int a, int b) {
        int sum = 0;
        sum = a+b;
        return sum;
}
```

# C: Defining functions

- A program in C is mainly a **set of functions codes**.
- During the execution of a program, the structure of the functions calls is the one of a **stack**.
- At the bottom of the stack is the first function that is called wen you execute the program, the **function main()** (which always needs to be defined).

# Control Structures

# If... then... else...

Conditional evaluation

```
if  (expression1)
        instructionblock1
else  if  (expression2)
        instructionblock2
...
else  if  (expresionn)
        instructionblockn
else
        instructionblockdefault
```

When using more than one instruction in a block, needs {}

# switch: handling cases

```
int a=1;
int b;
switch (a) {
        case 1:
                b=5;
        case 2:
                b=9;
        default :
                b=3
                break ;
}
```

Do not forget the break after each case block!

# switch: handling cases

```
switch (expression) {
        case valor1 :
                instruccionbloque1
                break;
        case valor2 :
                instruccionbloque2
                break;
                ...
        case valorn :
                instruccionbloquen
                break;
        default :
                instruccionbloquedefault
                break;
}
```

# while

```
i = 2;
while (i<5) {
        printf ("\n i = %d",i++);
}
```

Does not enter the while block if the condition is not OK.

```
i = 2;
do {
        printf ("\n i = %d", i ++);
} while (i<5);
```

Enters the block and then decides whether to continue or not.

# for

```
for (i = 0;i<5; i++)
        printf ("\n i = %d",i);
```

Initialization/Termination condition/Post-instruction.

# continue/break

- To exit cycles

```
i = 2;
do {
        printf ("\n i = %d",i++);
        if (i==6) break;
} while (i<100);
```

- To continue to the next cycle

```
for (i=0;i<5;i++) {
        if (i%2==1)
                continue;
        printf ("\n i = %d",i);
}
```

# The preprocessor

# The preprocessor

- The preprocessor transforms your code into into another code (but that stays code!).
- Its directives can be used to:
  - Include the content of another file (#include).
  - Define constants (#define).
  - Define macros (#define).
  - Allow conditional compilation (#if, #ifdef).

# The #include directive

- Includes the content of a file. Think in copy/paste.
- Is used mainly for headers: To include function declarations.

```
#include <math.h>
double x = 0.144;
double y = pow(x,3.5) ;
```

- Including with <> will search in directory path predetermined through the compiler ("-I") and then environment variables (standard library...).
- Including with "" will search first in the local directory.

# The #define directive

#define something someotherthing

Its main usage is to **define constants**: The preprocessor replaces any occurrence of 'something' by 'someotherthing' (as a **text**). They can be combined:

```
#define NUM_LINES 480
#define NUM_COLUMNS 640
#define IMAGE_SIZE NUM_LINES * NUM_COLUMNS
```

# The #define directive

Some useful predefined constants:

| Constant | Use |
|---|---|
| __LINE__ | Line number in the code |
| __FILE__ | Compiled file name |
| __DATE__ | Compilation date |
| __TIME__ | Compilation time |

# Macros

Macros: replace text expressions that may depend on other expressions, in a "function-like" way (but the result is NOT a function, this is just text edit).
In general

#define macroname(argexpression) macrobody

Example:

```
#define max(x,y) (x>y ?x:y)
#define min(x,y)  (x<=y?x:y)
```

# Macros

Caution: no space before the (;

```
#define square (x) (x * x)
```

Be aware of side effects (1)

```
#define square(x) (x * x)
int c = square( a+b ) ;
```

Be aware of side effects (2)

```
#define square(x) ((x)*(x))
int b = square(++a) ;
```

# Operators

# C: Arithmetic operators

- Arithmetic operators: +,-,/,*,%
- **Caution to the operator**/ (integer division/division): to be an integer division, both operands should be integers.
- For more mathematical operators, see **math.h**.

# C: lvalue/rvalue

- An **lvalue** refers to an expression to which can be associated a memory address: They can be put at the left of the assignment operator "=".
- An **rvalue** is typically the result of mathematical operations (they "exist" at some point in CPU registers) but they are not kept in memory.
- i is an lvalue, 2*i is not.

# C: composite assignation

`+=` `-=` `*=` `/=` `%=` `&=` `^=` `|=` `<<=` `>>=`

- As an example, $a+ = b$ is equivalent to $a = a + b$.
- The advantage is that the term on the left is **evaluated once**.

# C: increment operators

- Increment/decrement the values of integers.
- Pre/post operation.

```
int a = 3,b,c;
b = ++a ;
/* a and b are 4 */
c = b++;
/* c is 4 and b is 5 */
```

- Caution!

```
for (unsigned char i=10;i>=0;i--) {
        ...
}
```

# C: Comparison operators

- Difference between == (comparison) and = (assignment) .
- Operators >, <, >=, <=, ==, ! =.
- Be careful when using close floating numbers (see Session 1).

# C: Logic operators

In C, originally no Boolean type, only integers (0:false, 1:true).

| | |
|---|---|
| && | AND |
| \|\| | OR |
| ! | NOT |

- The evaluation of a logic operator returns a 1 or 0.
- Evaluation **from left to right** with groups of expressions:

```
if ((i>=0) && (i<= 9) && !(a[i]==0) && (b[i]>3)) {
        ...
```

# C: Logic operators

Choose smartly the order of evaluated expressions: can win computational time:

```
if (( frequentlyViolatedPrecondition ) && ( lessFrequentlyViolayedPreco
        ...
}
```

# C: Bitwise operators

| | |
|---|---|
| & | AND |
| \| | OR |
| ^ | XOR |
| ~ | NOT |

<< and >> : bits shift to left/right/

Do not mismatch with Boolean operators.

# C: Bitwise operators

What does this function do?

```
void mysterious(int *x, int *y) {
        *y = *x ^ *y;
        *x = *x ^ *y;
        *y = *x ^ *y;
}
```

# C: Bitwise operators

Shifts add zeros at the end/beginning of the representation.
Caution!

- with data size,

```
unsigned char c1 =128;
unsigned char c2=c1 <<1;
 printf ("%d\n",c2);
```

- with data sign,

```
char c3 =127;
char c4= c3 <<1;
 printf ("%d\n",4) ;
```

- with shift to the right: it **does not change the sign**

```
char c5 =-127;
char c6=c5 >>1;
 printf  ( "%d\n" , c6 ) ;
```

# C: Bitwise operators

To avoid problems:

- be sure that you use bit shifts on unsigned variables,
- don't abuse of them to keep your code readable.
- typically, for multiplication/divisions by powers of two.

# C: Bitwise operators

Applications: flags.

```
#define FLAG_X 0
#define FLAG_1 1
#define FLAG_2 2
#define FLAG_3 4
void function(unsigned int flags) {
        if (flags!=0) {
                if ((flags & FLAG_1)!=0)
                ...
                if ((flags & FLAG_2)!=0)
                ...
        } else { // FLAG X
        ...
// The calls
function(FLAG_1|FLAG_2);
```

# C: Cast operators

To convert variables from one type to another, one uses the **cast operator**:

```
type1 b = ...;
type2 a = (type2) b;
```

Example:

```
int  i=1, j=2;
double medio = (double)i/j  ;
```

# C: Automatic cast

With operations between heterogeneous data, **implicit cast**!

- Between integers: char/short are converted into int; then the compiler selects the largest present among

  int , long , long long (c99)

- When unsigned and signed integers are mixed, **implicit cast to unsigned**.
- Between integers and floats, **conversion to float**.
- Between floats, smaller operand type converted into the larger:

  float , double , long double

  Caution to overflows...

# C: Automatic cast

Side effects:

```
unsigned long a = 23 ;
signed char b   = -23;
 printf ("a %c b\n", a<b ? '< ' : ( a==b ? '= ' : '> ' ) ) ;
```

(cf. Gandhi bug)

# C: Address operator

One of the most emblematic of the C language!

```
type1   a= ...;
type1  *b=  &a;
type1 **c=  &b;
```

Gives the **physical memory address** of a variable.

# C: Priorities among operators

| Priority | Operadores |
|---|---|
| 1 (strongest) | () |
| 2 | ! ++ – |
| 3 | * / % |
| 4 | + - |
| 5 | < <= > >= |
| 6 | == != |
| 7 | && |
| 8 | \|\| |
| 9 | = += -= *= /= %= |

- With operators at the same level: From left to right.
- Use parenthesis.

# C: Priorities among operators

```
int a =2;
int b =2;
printf ("%d" ,!--a==!b ) ;
```