Le langage C

Les origines de C

Développé par Dennis Ritchie entre 1969 et 1973, le langage C est la conséquence de l'évolution de deux langages BCPL et B.

- La page Web de Dennis Ritchie
- Le Lysator

Objectif d'un nouveau langage :

Développer Unix "**confortablement**" (au lieu des langages assembleurs) !

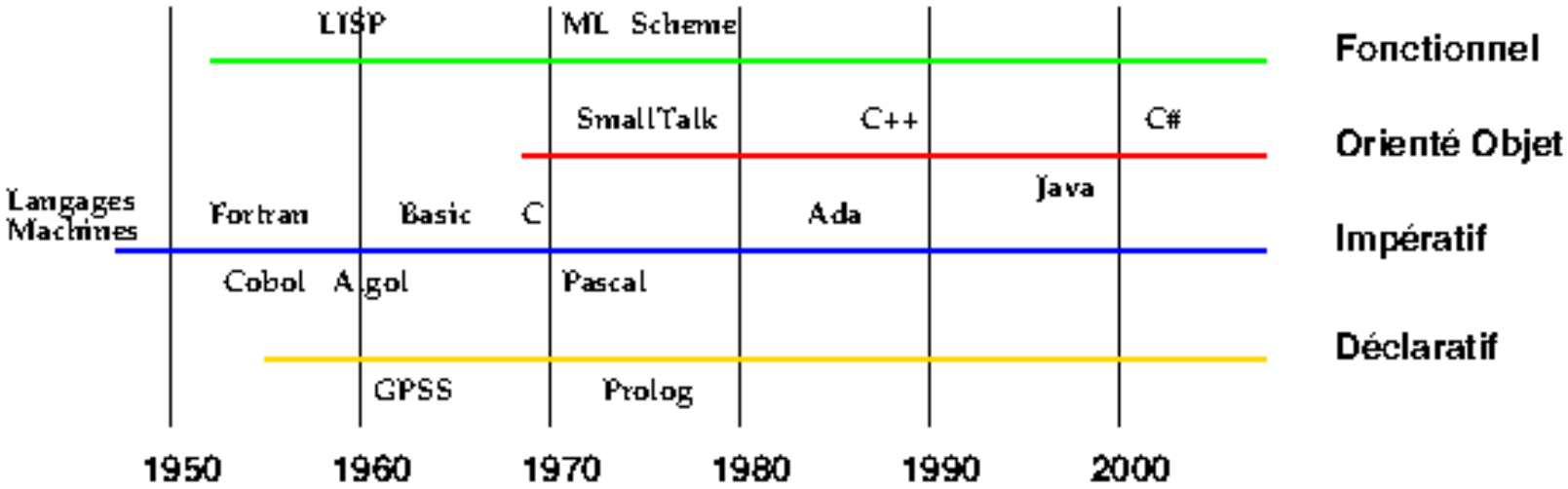
- Cet aspect "confort" est obtenu en augmentant le "niveau" du langage de programmation.
- Ce "niveau" correspond à un certains de concepts disponibles dans la langage :
  - ✔ types,
  - ✔ structures de contrôles, . . .

Ecriture Simplifiée : Assembleur PDP8 / Langage C

```
*200                    // set assembly origin (load address)
hello,  cla cll
        tls             // tls to set printer flag.
        tad charac      // set up index register
        dca ir1         // for getting characters.
        tad m6          // set up counter for
        dca count       // typing characters.
next,   tad i ir1       // get a character.
        jms type        // type it.
        isz count       // done yet ?
        jmp next        // no : type another.
        hlt
type,   0               // type subroutine
        tsf
        jmp .-1
        tls
        cla
        jmp i type
charac, .               // used as initial value of ir1
        310  // H
        305  // E
        314  // L
        314  // L
        317  // O
        241  // !
m6,     -15
count,  0
ir1 = 10
```

```
#include <stdio.h>

int main(void) {
  printf("Hello!\n");
  return 0;
}
```

Le langage C n'est pas, et n'a jamais été, le "seul" langage de programmation.



Par exemple, avant C, il y a :

- ✔ Fortran, Cobol,
- ✔ Lisp, . . .

Paradigme

Le chemin intellectuel entre le problème du monde réel que l'on souhaite "informatiser" et la machine physique qui sera le support de l'exécution peut faire l'objet de différents point de vues !

What do you see?
By shifting perspective you might see an old woman or a young woman.

Dépendant du contexte :

- ✔ domaines d'applications,
- ✔ puissance et architecture de la machine, . . .
- ✔ . . .

Programmation

Le terme informatique est un néologisme (i.e. mot nouveau) construit à partir des mots `information` et `automatique` par P. Dreyfus en 1962.

La définition acceptée par l'Académie Française est la suivante :

- « science du traitement rationnel, notamment par machines automatiques, de l'information . . . »

La notion de **programmation** aborde la problématique de l'expression de ce traitement automatique de l'information :

La solution à ce problème n'est pas unique !

Chaque paradigme développe une approche différente / complémentaire dans le cadre de l'élaboration d'une solution.

Paradigme Impératif

Les abstractions proposées/manipulables par le langage C restent très proches de celles disponibles dans la machine physique :

de la mémoire et des instructions !

➠ Il n'y a pas d'"objets", d'"itérateurs", de "généricité" et autres concepts désormais courant dans les langages modernes.

En ce sens, on dit souvent que le **C n'est qu'« un » assembleur de haut niveau**.

Paradigme Impératif

Le langage C permet l'utilisation des paradigmes structurés, procéduraux et impératifs !

Faire un programme en langage C consiste :

① à définir l'ensemble des variables mémoire formant **l'état** du programme.

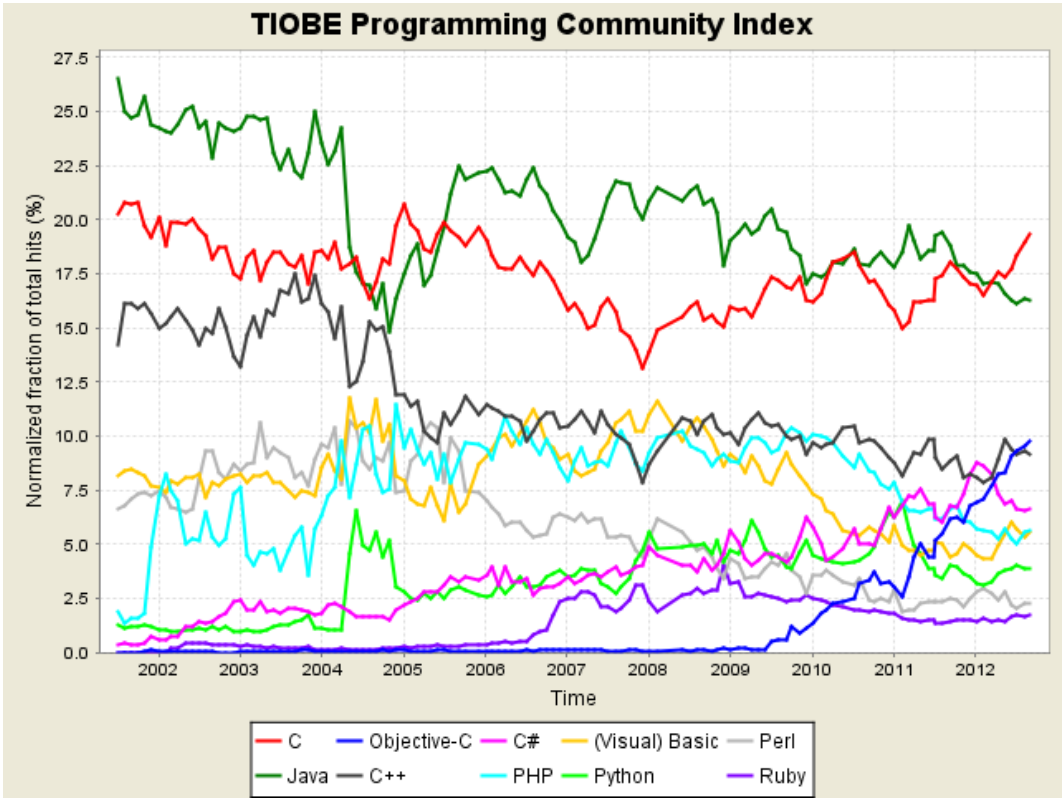
② et la suite/succession d'instructions produisant **les transitions** entre les états par lesquels devra passer la mémoire pour que :

- ➠ en partant d'un **état initial** permettant l'initialisation du programme,
- ➠ elle arrive dans un **état final** fournissant les résultats recherchés.

➢ On dit aussi que les instructions réalisent des « effets de bords ».

Popularité et Intéret du Langage C

Un langage « ancien » **MAIS ENCORE très largement utilisé :**



TIOBE : « Popularité » des langages de programmation

Les motivations et pourquoi on utilise C ?

Trois objectifs prioritaires sont fixés dans la définition de ce langage :

① Gagner en facilité d'écriture et en clarté,

② Conserver les possibilités de manipulation « fine » de la machine que propose l'assembleur tout en étant portable ($\rightarrow$ introduction des types),

③ Faire un compilateur « **léger** »/**simple et efficace**.

En 1970, la mémoire RAM d'un PDP11 c'est 24 KOctets soit 10^5 fois moins qu'un ordinateur familial 2012 (2 GOctets) !

➢ A l'échelle d'un terrain de foot ($100m * 70m$ représentant la mémoire d'un PC en 2012),

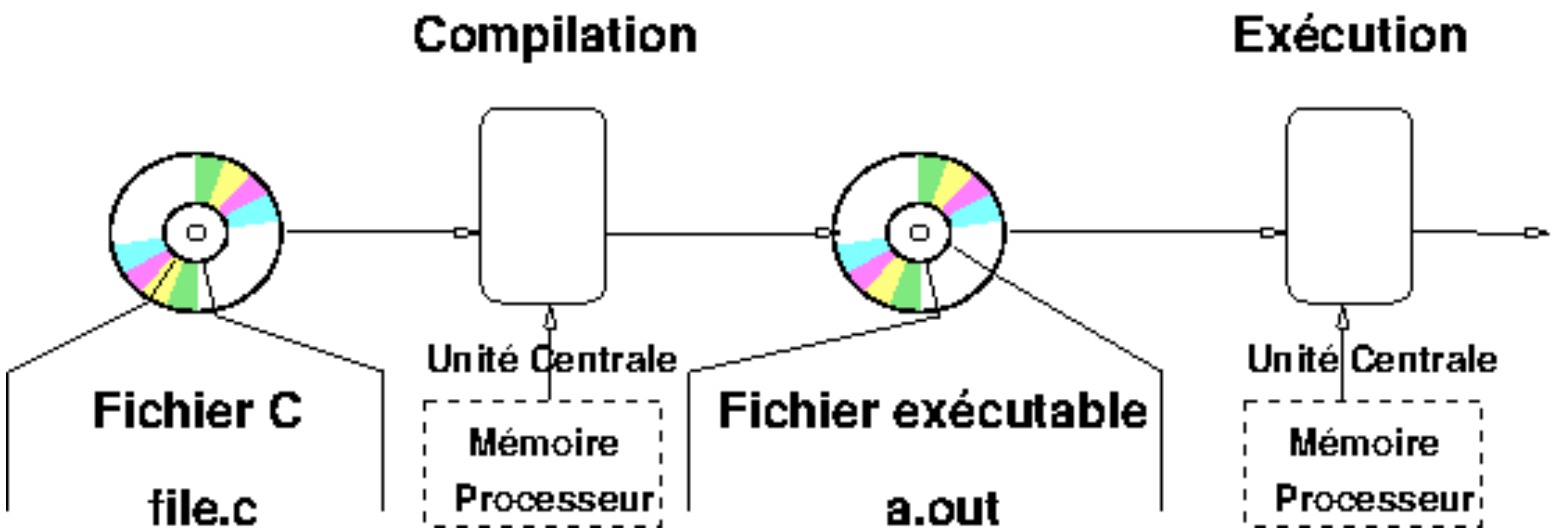
➢ c'est un tout petit peu plus qu'une feuille A4 !

Visiblement ces critères restent d'actualité !

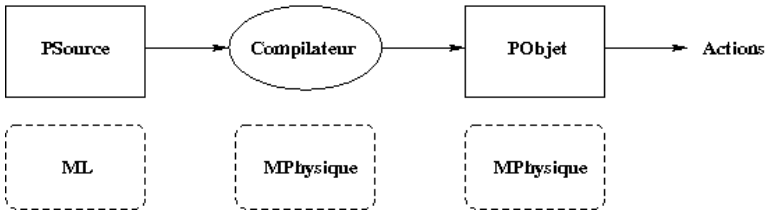
Efficacité

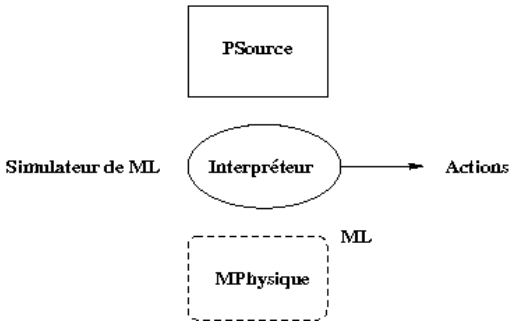
Le C est un **langage compilé** :

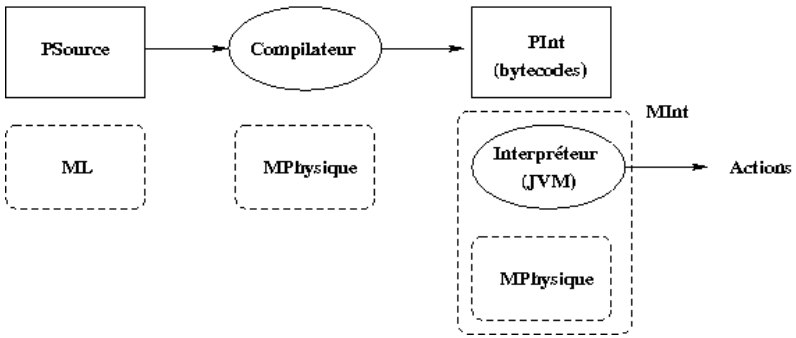
- Le compilateur produit un fichier contenant des instructions (i.e. codes) directement destinées au microprocesseur (la machine cible **matérielle**.



Schémas d'exécution : situer la compilation !







① Compilation : Langage C, C++

➢ Traduire le programme P_{source} dans un programme P_{objet} et faire exécuter P_{objet} sur $M_{physique}$.

② Interprétation : Langage Lisp

➢ Construire (par programme) sur $M_{physique}$ un simulateur de la machine M_L, et faire exécuter le programme original P_{source} sur ce simulateur (l'interprète).

③ Mixte : Langages Java ou Python

➢ Construire un compilateur traduisant le programme P_{source} vers un programme P_{Int} destiné à s'exécuter sur une machine logicielle virtuelle (par exemple JVM).

Premier programme en C

Le langage de programmation va permettre de formaliser l'algorithme et de permettre son exécution par une machine.

- Un programme en langage C dans un fichier,
- Compilation,
- Exécution.

Un exemple . . . tout simple !

➢ Faire afficher (par pas de 20) pour chaque température Fahrenheit entre 0 et 300, le degré celsius correspondant

$$celsius = \frac{5.0}{9.0} * (fahr - 32.0)$$

```
> a.out
  0 <=>  -17.8
 20 <=>   -6.7
 40 <=>    4.4
 60 <=>   15.6
 80 <=>   26.7
100 <=>   37.8
120 <=>   48.9
140 <=>   60.0
160 <=>   71.1
180 <=>   82.2
200 <=>   93.3
220 <=>  104.4
240 <=>  115.6
260 <=>  126.7
280 <=>  137.8
300 <=>  148.9
```

Table de correspondance : Fahrenheit-Celsius

```c
/*-------------------------------------------
 * Conversion Fahrenheit-Celsius
 * Auteurs      : K et R ------------------*/
#include <stdio.h>
#include <stdlib.h>

/* MacroCste = Pas de l'évolution de fahr */
#define INTERVALLE 20

/*-------------------------------------------
 * Fonction main : point d'entrée du pgm  */
int main(int argc, char *argv[]){
  /* Définition des variables   */
  float fahr, celsius;
  int mini, maxi; /* En Fahrenheit */
  /* Définition des traitements */
  mini = 0;
  maxi = 300;
  fahr = mini;
  while (fahr <= maxi) {
    celsius = (5.0/9.0) * (fahr-32.0);
    printf("%3.0f <=> %6.1f \n", fahr, celsius);
    fahr = fahr + INTERVALLE;
  }
  return EXIT_SUCCESS;
}
/*-------------------------------------------*/
```

```
> a.out
  0 <=>  -17.8
 20 <=>   -6.7
 40 <=>    4.4
 60 <=>   15.6
 80 <=>   26.7
100 <=>   37.8
120 <=>   48.9
140 <=>   60.0
160 <=>   71.1
180 <=>   82.2
200 <=>   93.3
220 <=>  104.4
240 <=>  115.6
260 <=>  126.7
280 <=>  137.8
300 <=>  148.9
```

Concepts/notions présents dans ce "petit" programme

```
1   /*-------------------------------------------
2    * Conversion Fahrenheit-Celsius
3    * Auteurs      : K et R ------------------*/
4   #include <stdio.h>
5   #include <stdlib.h>
6
7   /* MacroCste = Pas de l'évolution de fahr */
8   #define INTERVALLE 20
9
10  /*-------------------------------------------
11   * Fonction main : point d'entrée du pgm  */
12  int main(int argc, char *argv[]){
13    /* Définition des variables   */
14    float fahr, celsius;
15    int mini, maxi; /* En Fahrenheit */
16    /* Définition des traitements */
17    mini = 0;
18    maxi = 300;
19    fahr = mini;
20    while (fahr <= maxi) {
21      celsius = (5.0/9.0) * (fahr-32.0);
22      printf("%3.0f <=> %6.1f \n", fahr, celsius);
23      fahr = fahr + INTERVALLE;
24    }
25    return EXIT_SUCCESS;
26  }
27  /*-------------------------------------------*/
```

: Unité de compilation
: Commentaires
: Directives préprocesseur
: Fichiers d'entêtes
: Macro Constante
: Littéral entier
: Définition de fonction
: Type retourné
: Paramètres
: Variable Scalaire
: Agrégat tableau
: Pointeur sur char
: Variables automatiques
: Type flottant
: Type entier
: Opérateur d'affectation
: Instructions
: Enoncé itératif
: Opérateur de comparaison
: Bloc d'instructions
: Littéral flottant
: Opérateurs flottants : $/, *, -$
: Appel de fonction
: Bibliothèque d'E/S de haut niveau
: Conversion implicite : $+$
: Instruction return
: Valeur retournée
: Code de retour (communication avec le Shell)

Finalement, ce n'est pas un "si petit" programme . . .

Un paradigme : Une syntaxe de fichier source.

La structure du programme est conforme au paradigme impératif.

```
1  /*-----------------------------------------
2   * Conversion Fahrenheit-Celsius
3   * Auteurs       : K et R ----------------*/
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  /* MacroCste = Pas de l'évolution de fahr */
8  #define INTERVALLE 20
9
10 /*-----------------------------------------
11  * Fonction main : point d'entrée du pgm  */
12 int main(int argc, char *argv[]){
13   /* Définition des variables   */
14   float fahr, celsius;
15   int mini, maxi; /* En Fahrenheit */
16   /* Définition des traitements */
17   mini = 0;
18   maxi = 300;
19   fahr = mini;
20   while (fahr <= maxi) {
21     celsius = (5.0/9.0) * (fahr-32.0);
22     printf("%3.0f <=> %6.1f \n", fahr, celsius);
23     fahr = fahr + INTERVALLE;
24   }
25   return EXIT_SUCCESS;
26 }
27 /*-----------------------------------------*/
```

Elle est composée de :

① **Déclarations de variables** qui contribuent à déterminer l'état du programme.

➢ Il y a plusieurs formes de variables : statiques, dynamiques, ...

② **Déclarations de traitements** sur ces variables

➢ C'est eux qui provoquent les changements d'états du programme.

➢ Les traitements prennent la forme d'une séquence d'instructions nécessairement (en C) incluse dans une **fonction**.

Variables et Fonctions

```
1   /*-------------------------------------------
2    * Conversion Fahrenheit-Celsius
3    * Auteurs       : K et R ------------------*/
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include "celsius.h"
7
8   /* MacroCste = Pas de l'évolution de fahr */
9   #define INTERVALLE 10*2
10
11  /*-------------------------------------------
12   * Fonction main : point d'entrée du pgm  */
13  int main(int argc, char *argv[]){
14    /* Définition des variables   */
15    float fahr, celsius;
16    int mini, maxi; /* En Fahrenheit */
17    /* Définition des traitements */
18    mini = 0;
19    maxi = 300;
20    fahr = mini;
21    while (fahr <= maxi) {
22      celsius = getcelsius(fahr); /* FONCTION !! */
23      printf("%3.2f <=======> %6.1f \n", fahr, celsius);
24      fahr = fahr + INTERVALLE;
25    }
26    return EXIT_SUCCESS;
27  }
28  /*-------------------------------------------*/
```

Nommer une **variable**, c'est donner un nom à une zone mémoire et donc la faculté de pouvoir l'utiliser pour y lire ou y écrire une valeur.

Le concept de **fonction** permet de nommer un ensemble d'instructions.

- ➡ On peut invoquer ces instructions par l'appel à la fonction.
- ➡ L'appel de la fonction peut permettre une initialisation spécifique de variables : les paramètres de la fonction.
- ➡ En C, il n'y a pas de fonction locale à une autre fonction : **un seul plan de fonctions**.

C : un langage compilé

Le processeur cible est celui d'une machine matérielle.

- ➢ Les codes opérations sont ceux d'un processeur "en silicium".

C'est l'invocation du compilateur qui va permettre de les générer !

```
gcc -Wall -Werror fahr.c
```

On trouve sur quasiment toute machine un compilateur C :

- ➢ Ces compilateurs lorsqu'ils sont invoquables textuellement s'appellent `cc` (pour C Compiler) ou `gcc` (g pour Gnu C Compiler).

Remarquons que **le suffixe du nom du fichier source** doit être `.c`

- ➢ L'option **-Wall** demande au compilateur de signaler tous les risques (oubli de déclaration préalable à l'utilisation, variable non utilisée . . .)
- ➢ L'option **-Werror** arrête la compilation lorsqu'un avertissement (warning) est généré.

Le fichier exécutable

Si la phase de compilation s'est déroulée correctement . . .

➢ Lire les informations (même en anglais !) fournies par la compilation.

Elle crée (ou écrase) un **fichier exécutable** : `a.out` ou `a.exe`

➢ Le nom peut être choisi dans la commande de compilation.

La commande de dé-assemblage permet de voir code machine :

```
objdump -d a.out
```

```
1   as.out:     file format elf32-i386
2
3   Disassembly of section .text:
4
5   08048228 <main>:
6    ...
7    804823c:       fstpl  (%esp)
8    804823f:       call   8048250 <_sin>
9    8048244:       fstps  0xffffffff8(%ebp)
10   ...
11   804824b:       pop    %ebp
12   804824c:       lea    0xfffffffc(%ecx),%esp
13   804824f:       ret
14
15  08048250 <_sin>:
16   ...
17   8048271:       fstp   %st(1)
18   8048273:       fsin
19   8048275:       ret
20   8048276:       nop
```

Exécution du programme

Ce que l'on résume par "exécution du programme" correspond à deux phases :

① Le **chargement** du programme en mémoire (depuis le disque où il est stocké),

② et le **lancement** de son exécution.

Dès lors, le système d'exploitation lui attribut du temps CPU.

L'exécution est provoquée, en version textuelle, par l'appel du nom du fichier exécutable.

`a.out` ou `a.exe` .

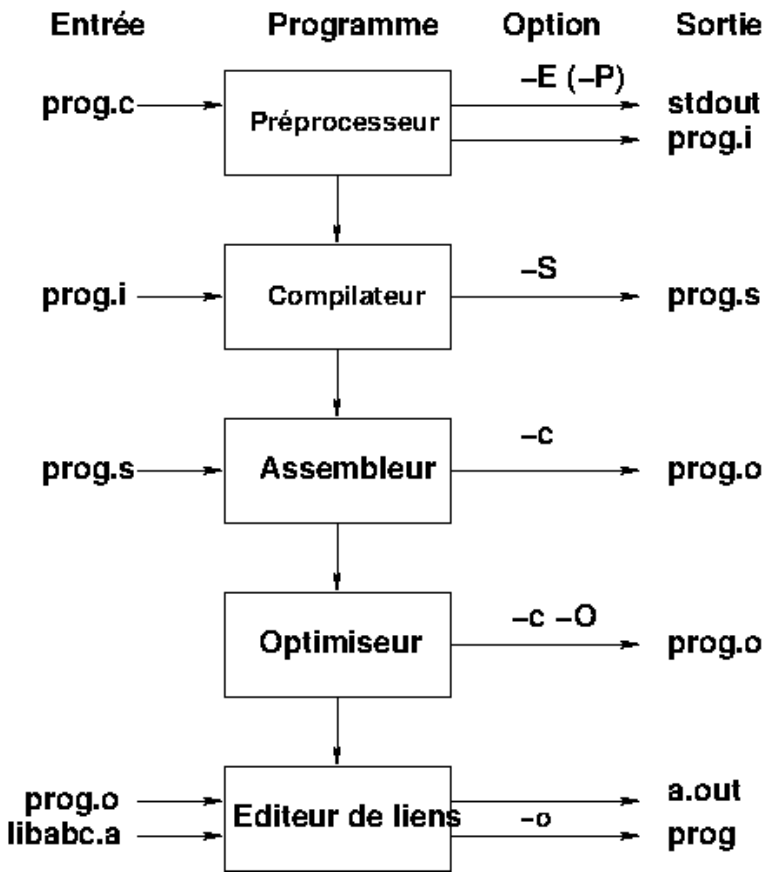
```
> a.out

  0 <=>   -17.8
 20 <=>    -6.7
 40 <=>     4.4
 60 <=>    15.6
 80 <=>    26.7
100 <=>    37.8
120 <=>    48.9
140 <=>    60.0
160 <=>    71.1
180 <=>    82.2
200 <=>    93.3
220 <=>   104.4
240 <=>   115.6
260 <=>   126.7
280 <=>   137.8
300 <=>   148.9
```

Le fichier exécutable est une commande SHELL à part entière.

➢ Au même titre, que les commandes : `ls` , `cp` , `xclock`

Chaîne de compilation

Derrière l'appel à `gcc` se cache une grande complexité de traitements :



On y reviendra . . .

Anatomie d'un programme C

- Les différentes parties d'un programme C

Unité de compilation

Définition :

Un texte accepté par un compilateur constitue une **unité de compilation** .

- Une unité de compilation se situe toujours dans un fichier.
- Certains langages permettent de sélectionner dans un fichier, une classe (Java) ou un module (Modula2).

En C :

Le langage C ne permet rien de cela :

- Le fichier **est** l'unité de compilation.

Dans le cadre de règles de « bonne lisibilité », rappellons que :

- La taille, en nombre de lignes, de ces morceaux doit rester raisonnable. Il est difficile de définir un standard, mais $<$1000 lignes.
- Une largeur de ligne $<$ 80 colonnes permet une présentation correcte et plus particulièrement lors d'une éventuelle sortie sur imprimante.

Anatomie d'une Unité de Compilation C

```c
/*
 * Prologue : Fichier, Auteurs, Date, Version
 */

/*------- Header files includes ----------*/
#include ....

/*-------  Définitions de constantes -------*/
#define .....

/*-------  Définitions de type ----------*/
typedef ..... ;

/*-------  Déf et Decl de variables globales */
static int .... ;
float .... ;

/*-------  Définitions de fonctions. ------*/
float fonction1(/* Def des param. formels */){
  /* Déf et Décl de variables locales */
  ....
  /* Instructions                      */
  ....
}

/*------  Fonction point d'entree du pgm ---*/
int main(int argc, char *argv[]){
  /* Déf et Décl de variables locales */
  .....
  /* Appel de fonctions               */
  fonction1(/* parametres effectifs   */)
  .....
}
```

Le **prologue** est un commentaire en tête de fichier pour expliquer :

- le contenu du fichier ;
- le rôle des objets définis par le fichier et qui auraient une visibilité externe ;
- le nom des auteurs ;
- la date de révision ;
- ...

Ces informations sont particulièrement importantes lorsque l'on travaille en équipe.

- On écrit du code ... pour que d'autres l'utilisent et le lisent !

Anatomie d'une Unité de Compilation C

```c
/*
 * Prologue : Fichier, Auteurs, Date, Version
 */

/*------- Header files includes ----------*/
#include ....

/*-------  Définitions de constantes ---------*/
#define .....

/*-------  Définitions de type ----------*/
typedef ..... ;

/*-------  Déf et Decl de variables globales */
static int .... ;
float .... ;

/*-------  Définitions de fonctions. -------*/
float fonction1(/* Def des param. formels */){
  /* Déf et Décl de variables locales */
  ....
  /* Instructions                    */
  ....
}

/*-------  Fonction point d'entree du pgm ---*/
int main(int argc, char *argv[]){
  /* Déf et Décl de variables locales */
  .....
  /* Appel de fonctions              */
  fonction1(/* parametres effectifs  */)
  .....
}
```

Les **commentaires doivent** décrire les points-clés du programme et notamment le rôle

- ➢ d'une fonction,
- ➢ d'un paramètre,
- ➢ d'une variable, ...

Tout langage a une syntaxe pour les commentaires :

- ➢ Commentaires valides en C :

```c
/*
 ca
 */

 /*  ou ca                    */
```

- ➢ Commentaires non-valides en C :

```c
 /*
      /* non valide car emboîtés !! */
 */
```

- ➢ Commentaires lignes en C ?

```c
 // Non Valide en C90 mais valide en C99
```

Anatomie d'une Unité de Compilation C

```
/*
 * Prologue : Fichier, Auteurs, Date, Version
 */

/*------- Header files includes ----------*/
#include ....

/*-------  Définitions de constantes ------*/
#define .....

/*-------  Définitions de type ------------*/
typedef ..... ;

/*-------  Déf et Decl de variables globales */
static int .... ;
float .... ;

/*-------  Définitions de fonctions. ------*/
float fonction1(/* Def des param. formels */){
  /* Déf et Décl de variables locales */
  ....
  /* Instructions                        */
  ....
}

/*-------  Fonction point d'entree du pgm --*/
int main(int argc, char *argv[]){
  /* Déf et Décl de variables locales */
  .....
  /* Appel de fonctions                  */
  fonction1(/* parametres effectifs     */)
  .....
}
```

On commence par l'analyse par **la définition des fonctions** .

- ➢ En C, c'est le seul moyen de faire figurer des instructions dans le programme.
- ➢ **On ne peut pas avoir d'instruction hors d'une fonction !**

Une fonction informatique s'inspire de l'objet bien connu en mathématiques !

a) Prototype : ensemble de départ et ensemble d'arrivée,

b) Paramètres formels,

c) Définition,

d) Un objet dynamique : Principe de l'appel . . . pour récuperer la(les) valeur(s) calculée(s) par la fonction.

Anatomie d'une Unité de Compilation C

```
/*
 * Prologue : Fichier, Auteurs, Date, Version
 */

/*------- Header files includes ----------*/
#include ....

/*-------  Définitions de constantes -------*/
#define .....

/*-------  Définitions de type ----------*/
typedef ..... ;

/*-------  Déf et Decl de variables globales */
static int .... ;
float .... ;

/*-------  Définitions de fonctions. -------*/
float fonction1(/* Def des param. formels */){
  /* Déf et Décl de variables locales */
  ....
  /* Instructions                      */
  ....
}

/*-------  Fonction point d'entree du pgm ---*/
int main(int argc, char *argv[]){
  /* Déf et Décl de variables locales */
  .....
  /* Appel de fonctions                */
  fonction1(/* parametres effectifs    */)
  .....
}
```

Il peut n'y avoir aucune déclaration/définition de variable, mais il doit y avoir au moins la définition d'une fonction dont le nom est `main` .

```
/* Prototypes C99 */
int main ( void )
int main ( int argc, char *argv[] )
```

- C'est le **point de départ/point d'entrée** de l'exécution du programme !
  C'est « la » première fonction que le SHELL va chercher à appeler.
- Même vide il doit exister une fonction `main` pour que le compilateur puisse générer un exécutable.
- Par contre, **il ne peut y en avoir qu'une par programme !**

Anatomie d'une Unité de Compilation C

```c
/*
 * Prologue : Fichier, Auteurs, Date, Version
 */

/*------- Header files includes ----------*/
#include ....

/*-------  Définitions de constantes ---------*/
#define .....

/*-------  Définitions de type ----------------*/
typedef ..... ;

/*-------  Déf et Decl de variables globales */
static int .... ;
float .... ;

/*-------  Définitions de fonctions. -------*/
float fonction1(/* Def des param. formels */){
  /* Déf et Décl de variables locales */
  ....
  /* Instructions                     */
  ....
}

/*-------  Fonction point d'entree du pgm ---*/
int main(int argc, char *argv[]){
  /* Déf et Décl de variables locales */
  .....
  /* Appel de fonctions               */
  fonction1(/* parametres effectifs   */)
  .....
}
```

La fonction `main` permet de communiquer avec le système d'exploitation.

- On verra plus loin comment utiliser `main` pour donner des arguments en entrée à un programme.

```
a.out martin 1234 3.14
```

- L'entier retourné communique, à l'interprète de commandes (sous Unix, le shell)qui a lancé le programme, le statut d'exécution de ce programme ;

  Par convention, cet entier est 0 (ou plutôt `EXIT_SUCCESS`) si le programme s'est exécuté sans erreur et autre si ce n'est pas le cas.

Anatomie d'une Unité de Compilation C

```
/*
 * Prologue : Fichier, Auteurs, Date, Version
 */

/*------- Header files includes ----------*/
#include ....

/*-------  Définitions de constantes -------*/
#define .....

/*-------  Définitions de type ----------*/
typedef ..... ;

/*-------  Déf et Decl de variables globales */
static int .... ;
float .... ;

/*-------  Définitions de fonctions. -------*/
float fonction1(/* Def des param. formels */){
  /* Déf et Décl de variables locales */
  ....
  /* Instructions                      */
  ....
}

/*-------  Fonction point d'entree du pgm ---*/
int main(int argc, char *argv[]){
  /* Déf et Décl de variables locales */
  .....
  /* Appel de fonctions                */
  fonction1(/* parametres effectifs    */)
  .....
}
```

Les lignes de code qui commencent par # sont des **directives** (i.e. instructions) à destination du préprocesseur.

- Le **préprocesseur** est la première phase du processus de compilation : **elle réalise une ré-écriture du programme**.

`include` et `define` sont des directives incontournables :

- **include** : La **directive d'inclusion** provoque l'inclusion d'un fichier en lieu et place de la directive.

- **define** : La **directive de définition** définit une équivalence d'écriture entre deux chaînes de caractères.
  La deuxième sera substituée à la première durant la ré-écriture.

Anatomie d'une Unité de Compilation C

```
/*
 * Prologue : Fichier, Auteurs, Date, Version
 */

/*------- Header files includes ----------*/
#include ....

/*-------  Définitions de constantes ------*/
#define .....

/*-------  Définitions de type ------------*/
typedef ..... ;

/*-------  Déf et Decl de variables globales */
static int .... ;
float .... ;

/*-------  Définitions de fonctions. ------*/
float fonction1(/* Def des param. formels */){
  /* Déf et Décl de variables locales */
  ....
  /* Instructions                        */
  ....
}

/*-------  Fonction point d'entree du pgm ---*/
int main(int argc, char *argv[]){
  /* Déf et Décl de variables locales */
  .....
  /* Appel de fonctions                  */
  fonction1(/* parametres effectifs     */)
  .....
}
```

Déclarer avant d'utiliser !

La directive d'inclusion permet d'introduire dans le source du programme des « déclarations » de fonctions et de types que d'autres ont écrits.

```
#include <stdio.h>              /* Déclaration de printf */
....
printf("Hello World !\n");      /* Utilisation de printf */
```

Elle se trouve classiquement en tête de fichier car :

➢ **les déclarations doivent toujours précéder l'utilisation !**

Ainsi, stdio.h est le nom d'un fichier (/usr/include/stdio.h) qui contient la déclaration de la fonction printf :

```
int printf(const char *FORMAT [, ARG, ...]);
```

stdio.h n'est pas une bibliothèque . . . c'est une partie de son **interface** !

Si l'utilisation est conforme, la compilation de stdio.h n'engendre pas de code et n'alloue pas de mémoire.

➢ Elle sert au **contrôle de type**.

Anatomie d'une Unité de Compilation C

```
/*
 * Prologue : Fichier, Auteurs, Date, Version
 */

/*------- Header files includes ----------*/
#include ....

/*-------  Définitions de constantes ------*/
#define .....

/*-------  Définitions de type ------------*/
typedef ..... ;

/*-------  Déf et Decl de variables globales */
static int .... ;
float .... ;

/*-------  Définitions de fonctions. ------*/
float fonction1(/* Def des param. formels */){
  /* Déf et Décl de variables locales */
  ....
  /* Instructions                      */
  ....
}

/*-------  Fonction point d'entree du pgm ---*/
int main(int argc, char *argv[]){
  /* Déf et Décl de variables locales */
  .....
  /* Appel de fonctions                */
  fonction1(/* parametres effectifs    */)
  .....
}
```

Les macro-constantes permettent de définir des constantes nommées.

```
#define MAX_INDEX 100
...
while (i< MAX_INDEX)
...
if (k == MAX_INDEX)
...
```

Ainsi si l'index maximum (MAX_INDEX) venait à changer, il suffirait de modifier une seule ligne de code !

Une **erreur** classique :

- ➢ Mettre un « ; » en fin de directive . . . en se disant que c'est comme les instructions !

Au niveau du style de codage :

- ➢ Les macro-constantes utilisent des identificateurs en lettres capitales.

Anatomie d'une Unité de Compilation C

```
/*
 * Prologue : Fichier, Auteurs, Date, Version
 */

/*------- Header files includes ----------*/
#include ....

/*-------  Définitions de constantes -------*/
#define .....

/*-------  Définitions de type ----------*/
typedef ..... ;

/*-------  Déf et Decl de variables globales */
static int .... ;
float .... ;

/*-------  Définitions de fonctions. -------*/
float fonction1(/* Def des param. formels */){
  /* Déf et Décl de variables locales */
  ....
  /* Instructions                    */
  ....
}

/*-------  Fonction point d'entree du pgm ---*/
int main(int argc, char *argv[]){
  /* Déf et Décl de variables locales */
  .....
  /* Appel de fonctions              */
  fonction1(/* parametres effectifs  */)
  .....
}
```

Le langage C propose des types de base :

- ➢ int,
- ➢ float,
- ➢ char, ...

et des mécanismes de constructions de types agrégés (tableaux, structures, ...).

L'instruction **typedef** permet d'associer un nom de type à une de ces constructions.

Exemple :

```
/*  Création d'un type "entier"  */
typedef int entier;


/*  Création d'un type "matrix"  */
#define MAX 256
/* Creation du nom matrix */
typedef char matrix[MAX][MAX];
/* Utilisation pour définir une variable */
matrix m;
```

Anatomie d'une Unité de Compilation C

```
/*
 * Prologue : Fichier, Auteurs, Date, Version
 */

/*------- Header files includes ----------*/
#include ....

/*-------  Définitions de constantes -------*/
#define .....

/*-------  Définitions de type ----------*/
typedef ..... ;

/*-------  Déf et Decl de variables globales */
static int .... ;
float .... ;

/*-------  Définitions de fonctions. -------*/
float fonction1(/* Def des param. formels */){
  /* Déf et Décl de variables locales */
  ....
  /* Instructions                    */
  ....
}

/*-------  Fonction point d'entree du pgm ---*/
int main(int argc, char *argv[]){
  /* Déf et Décl de variables locales */
  .....
  /* Appel de fonctions              */
  fonction1(/* parametres effectifs  */)
  .....
}
```

La **définition de variable** hors de toute fonction crée des variables que l'on a coutume de désigner de façon simpliste comme « globales ».

- ➢ Il s'agit d'un espace mémoire que toutes les composantes du programme (autres fonctions et autres fichiers) pourront utiliser (ou altérer suivant le point de vue que l'on a) pour mémoriser de l'information durant la durée entière du programme.

Je n'ai pas d'exemple où cela ne soit pas une **erreur de programmation**.

- ➢ On peut toujours faire sans !
- ➢ C'est un frein au bon fonctionnement, à l'évolutivité et à la réutilisation du code.

Anatomie d'une Unité de Compilation C

```c
/*
 * Prologue : Fichier, Auteurs, Date, Version
 */

/*------- Header files includes ----------*/
#include ....

/*-------  Définitions de constantes ------*/
#define .....

/*-------  Définitions de type ------------*/
typedef ..... ;

/*-------  Déf et Decl de variables globales */
static int .... ;
float .... ;

/*-------  Définitions de fonctions. ------*/
float fonction1(/* Def des param. formels */){
  /* Déf et Décl de variables locales */
  ....
  /* Instructions                        */
  ....
}

/*-------  Fonction point d'entree du pgm ---*/
int main(int argc, char *argv[]){
  /* Déf et Décl de variables locales */
  .....
  /* Appel de fonctions                  */
  fonction1(/* parametres effectifs     */)
  .....
}
```

Le langage C n'impose pas de structure particulière à une ligne de programme.

➢ On peut écrire plusieurs instructions sur la même ligne.

Mais . . . l'abus est mauvais pour la santé !

```c
#include <stdio.h>
int a = 1000,b,c=8400,d,e,f[8401],g;
int main(int argc, char* argv[])
{
  for(;b-c;)
    f[b++] = a/5;
  for(;d=0,g=c*2;b=c-=14,printf("%.4d",e+d/a),e=d%a)
    for(;d+=f[b]*a,f[b]=d%--g,d/=g---,--b;d*=b);
}
```

Votre écriture doit être constamment conforme à des critères de lisibilité et de compréhension.

➢ Les standards de codage du projet Gnu[2] :
www.gnu.org/prep/standards.html

Un exemple : le tri par insertion

```c
/*-------------------------------------------------
 Fichier : trinsert.c ; Auteur : Gilles Menez
 Date : 9/10/1995 - Version : Fausse !
 Tri par insertion dans l'ordre croissant.
 -------------------------------------------------*/
#include<stdio.h>
#include<math.h>

#define MAX_SIZE 101

/*------------ Pt d'entrée ----------------*/
int main(int argc, char *argv[])
{
  int i,N;
  int list[MAX_SIZE];

  printf("Dim de la liste[1,%d] ?",MAX_SIZE);
  scanf("%d",&N);
  if(N<1||N>MAX_SIZE){/* test de conformite */
    fprintf(stderr,"valeur non conforme!\n");
    exit(1);
  }

  for(i=0;i<N;i++){ /* Initialisation */
    list[i]=rand()%1000;
    printf("%d\t",list[i]);
  }

  tri_par_insertion(list,N);   /* Tri */
  printf("\nListe triee:\n"); /* Affichage */
  for(i=0;i<N;i++)
    printf("%d\t",list[i]);
  printf("\n");
}
/*-------------------------------------------------*/
/* tri_par_insertion()                             */
/* In :                                            */
/* Out :                                           */
/* Role :                                          */
void tri_par_insertion(int list[],int N){
  int i,j;
  for(i=0;i<N;i++){
    for(j=i;j>0;j--){
      if (list[j]<list[j-1])
        echanger(list,j);
    }
  }
}
/*-------------------------------------------------*/
/* echanger()                                      */
/* In :                                            */
/* Out :                                           */
/* Role : echanger list[j] et son precedent        */

void echanger(int list[],int j){
  list[j]=list[j-1];
  list[j-1]=list[j]
}
/*-------------------------------------------------*/
```

La « vie » sans Notion de Type

Au début du développement d'Unix, que ce soit en langage B ou en langage BCPL, la notion de type n'existait pas :

- La programmation se faisait en « cellule » ou « mot » mémoire.
- L'évolution rapide du matériel a très vite démontré que **cette approche n'était pas viable** !

Un flottant, d'une machine à l'autre, n'occupait pas forcément le même nombre de cellules.

- Il fallait ré-écrire le code pour chaque machine.

La solution :

Abstraire la notion de flottant au niveau du langage de programmation

- **On écrirait que la variable est flottante** . . .
- C'est **le compilateur qui gèrerait la mise en correspondance avec le nombre de cellules mémoire** nécessaires !