I31 – TD 1 – CORRECTION

J. SEINTURIER (j.seinturier@univ-tln.fr)

1. Généralités

Exercice 1.1.
Décrire le processus de création d'un programme en langage C, précisez les notions d'algorithme, de codage de compilation et d'exécution (vous pouvez faire un schéma).

La programmation est le processus visant à créer un programme informatique permettant de résoudre un problème précis (réalisation d'un calcul, automatisation de tâches, …). Ce processus repose sur la communication entre un humain (formalisant le problème, proposant un algorithme) et une machine (exécutant l'implantation de l'algorithme). A partir du problème, plusieurs étapes emmènent vers l'exécution d'un programme produisant des résultats :

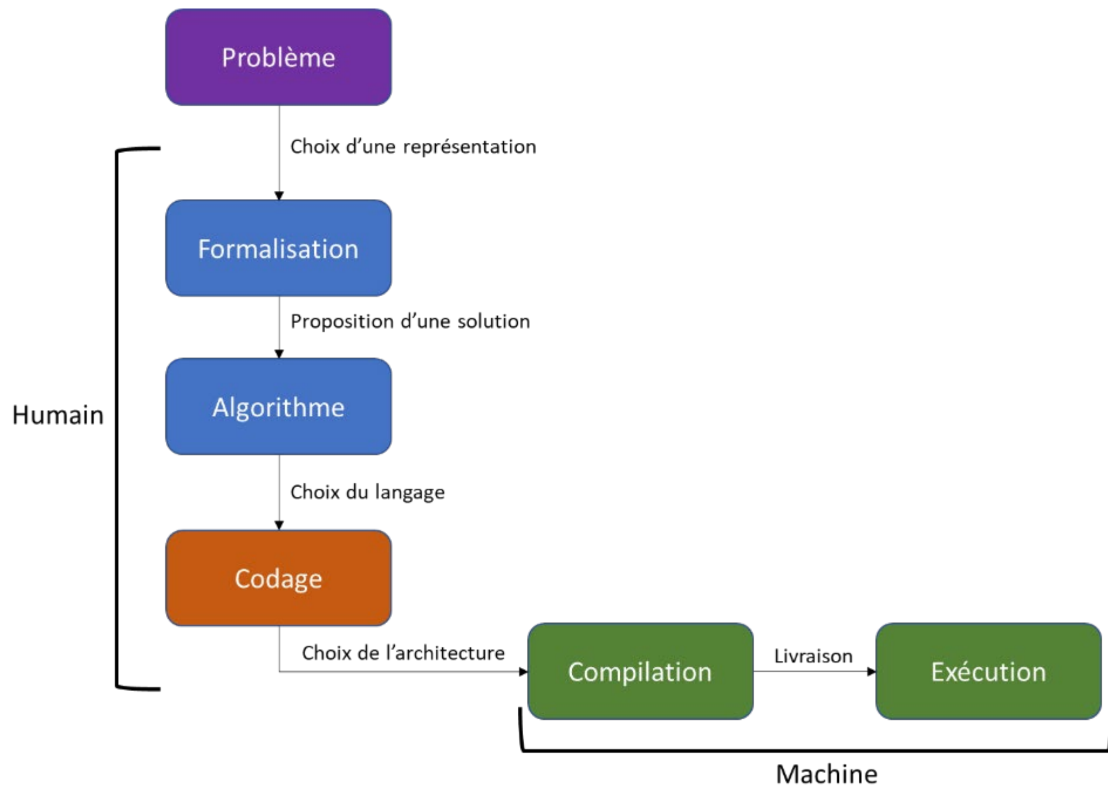

Figure 1 Etapes du processus de programmation

La formalisation du problème consiste à écrire le problème dans un formalisme adapté à sa résolution (par exemple en mathématiques, en logique, …) et à proposer une solution.

L'écriture d'un algorithme permet de traduire la solution proposée dans un formalisme, souvent de haut niveau, en ensemble d'opérations simples (voir https://fr.wikipedia.org/wiki/Algorithme)

Le codage consiste à la traduction de l'algorithme fournit dans un langage de programmation choisi. Le langage doit permettre de représenter efficacement l'algorithme tout en garantissant de produire un programme optimal et adapté.

La compilation est la traduction du programme écrit en langage de programmation vers une suite d'instructions en langage machine. Ce langage dépend de l'architecture (x86_32, x86_64, ARM64, …) et du système (Windows, Linux, MacOS, Android, …) sur lequel le programme devra être exécuté.

L'exécution est le déroulement du programme produit sur les données inhérentes au problème. C'est l'utilisateur final qui est impliqué dans cette étape.

Voici un exemple du processus de programmation pour un problème simple :

Problème : Calculer la composante maximale d'un vecteur de nombres réels.

Formalisation : Soit un vecteur V défini par $\{V = (v_1, \ldots, v_i, \ldots, v_n) \mid n \in \mathbb{N}, \forall i\ 1 \leq i \leq n, v_i \in \mathbb{R}\}$, une valeur maximale v_m de composante de V est définie par $\{v_m \in V \mid \forall v_i \in V, v_i \neq v_m, v_i \leq v_m\}$.

Algorithme :

```
début
        V : tableau de n réels
        i : entier
        vm : réel

        vm ⟵ v[1]
        i ⟵ 2

        tantque i < n faire
                si v[i] > v[m] alors
                        v[m] ⟵v[i]
                finsi
        fintq

        retourner vm
fin
```

Codage :

```c
#include <stdio.h>

int main(){
    const int N = 5;

    int V[5] = {1, 4, 5, 2, 3};

    int vm = V[0];

    for(int i = 1; i < N; i++){
        if (V[i] > vm)
            vm = V[i];
    }

    printf("%d", vm);
}
```

Exercice 1.2.
La commande pour compiler le programme prog.c en un exécutable prog est :

```
gcc prog.c -o prog
```

La commande `gcc` appelle le compilateur GNU C (GNU Compiler Collection). Le paramètre `prog.c` indique à la commande `gcc` que le fichier à compiler est `prog.c`. L'option `-o` indique à la commande `gcc` que l'exécutable produit doit s'appeler `prog`.

Si l'option `-o` est omise, `gcc` compile alors le programme dans un exécutable nommé `a.out`.

Exercice 1.3.
En langage C, qu'est-ce qu'une instruction précédée d'un #. Comment fonctionne ce type d'instruction ?

En langage C, une instruction précédée d'un # (croisillon et non hashtag ou dièse) est une instruction préprocesseur. Le préprocesseur est l'étape préliminaire à la compilation qui a pour but d'exécuter des instructions spécifiques avant le processus de compilation. Les instructions de préprocesseur permettent de modifier le code source avant la compilation, par exemple en activant ou désactivant certaines lignes de code, en définissant des valeurs ou en incluant du code provenant d'une source spécifiée (voir https://zestedesavoir.com/tutoriels/755/le-langage-c-1/1043_aggregats-memoire-et-fichiers/le-preprocesseur/).

Exercice 1.4.
Quel est le fonctionnement de l'instruction `#include` lors de la compilation ? Quel est le fonctionnement de l'instruction `#define` ?

L'instruction `#include` (syntaxe `#include <fichier>` ou `#include "fichier"`) copie le contenu du `fichier` passé en paramètre à l'endroit où se trouve l'instruction. Ce fonctionnement est utile lorsque l'on souhaite réutiliser du code (ex `#include<stdio.h>`) qui inclus les fonctions basiques d'entrées / sorties (`printf`, …). L'instruction `#define` (syntaxe `#define nom valeur`) remplace toutes les occurrences de `nom` par la `valeur` fournie dans le programme avant la compilation.

2. Variables

Exercice 2.1.
En langage C, qu'est-ce qu'une variable ? Qu'est-ce qu'un type ?

Une variable est un objet repéré par un nom, pouvant contenir des données, qui pourront être modifiées lors de l'exécution du programme.

En langage C, les noms de variables peuvent être aussi long que l'on désire, toutefois le compilateur ne tiendra compte que des 32 premiers caractères. De plus, elles doivent répondre à certains critères :

- un nom de variable doit commencer par une lettre (majuscule ou minuscule) ou un « _ » (pas par un chiffre)
- un nom de variable peut comporter des lettres, des chiffres et le caractère « _ » (les espaces ne sont pas autorisés !)
- les noms de variables ne peuvent pas être les noms suivants (qui sont des noms réservés) :
    - auto, char, double, float, int, long, short
    - signed, unsigned
    - if, else
    - switch, case, break, default
    - for, do, while
    - const, continue
    - extern
    - goto
    - register, return
    - sizeof, static
    - typedef, struct, union, enum
    - void, volatile

Le type d'une variable décrit l'ensemble des valeurs que peut prendre celle-ci. Par exemple, une variable de type `char` peut prendre des valeurs entières dans l'intervalle `[-127, 127]`.

Exercice 2.2.
Quels sont les types numériques disponibles en langage C ? Précisez leur plage de valeurs et leur taille d'encodage.

Les types numériques disponibles sont les suivants :

Type	Plage	Encodage
char	[-127, 127]	1 mot mémoire (1 octet sur machine 8bits)
unsigned char	[0, 255]	
short	[-32 767, 32 767]	2 octets
unsigned short	[0, 65 535]	
int	[-2 147 483 648, 2 147 483 647]	4 octets (si >= 32 bits)
unsigned int	[0, 4 294 967 295]	
long	[-2 147 483 648, 2 147 483 647]	4 octets (si >= 32 bits)
unsigned long	[0, 4 294 967 295]	
float	$\pm[3.4\times10^{-38}, 3.4\times10^{38}]$	4 octets
double	$\pm[1.7\times10-308, 1.7\times10^{308}]$	8 octets
long double	$\pm[3.4\times10^{-4932}, 3.4\times10^{4932}]$	16 octets

Source : https://web.maths.unsw.edu.au/~lafaye/CCM/c/ctype.htm

Exercice 2.3.
Donner le type et le nom de la variable destinée à recevoir les informations suivantes :

- votre âge
- la surface d'une parcelle cadastrale en m²
- l'âge moyen des personnes de votre famille
- Le nombre d'étoiles dans l'univers

L'âge d'une personne peut être raisonnablement compris entre 0 et 255 ans. L'utilisation d'une variable d'`unsigned char` est donc adaptée.

La surface d'une parcelle cadastrale peut varier d'une dizaine de m² (maison) à quelque milliers (champ, parc, …). Un type `unsigned short` est donc indiqué pour représenter une telle valeur.

L'âge moyen des membres d'une famille est un nombre réel (à cause de la division). Le type le plus adapté à sa représentation est donc le `float` (le double apportant une précision inutile)

D'après les scientifiques, l'univers contiendrai 200 000 000 000 000 000 000 000 étoiles. Cet ordre de grandeur 2×10^{23} peut être représenté par un `long double`, même si un nombre réel n'est pas nécessaire.

Exercice 2.4.
Expliquer ce que font chacune des lignes suivantes :

```
int n = 10, p = 4;
long q;
float x = 1.75;
```

`int n = 10, p = 4;` Déclare les variables n et p de type int et les initialise respectivement à 10 et 4.
`long q;` Déclare la variables de type long q sans l'initialiser.
`float x = 1.75;` Déclare la variables x de type float et l'initialise à 1.75.

Exercice 2.5.
Déclarer une constante pour pi=3,14159. Déclarer ensuite une variable puis l'initialiser à pi.

La déclaration d'une variable constante se fait par l'instruction :

```
const <type> <nom> = valeur
```

La déclaration de la variable pi est réalisée via l'instruction :

```
const float pi = 3.14159;
```

Voici un programme déclarant pi et l'affichant :

```
#include <stdio.h>

int main(){
    const float pi = 3.14;
    printf("pi = %f", pi);
}
```

ATTENTION : On trouve souvent dans des exemples les déclarations de constantes via l'instruction de préprocesseur #define, par exemple :

```
#define pi 3.14159
```

Cette façon de faire ne déclare pas réellement une variable constante mais remplace toutes les occurrences de pi dans le code par la valeur 3.14159.

Exercice 2.6.
Soient les trois instructions suivantes :

```
int n = 5, p;
int q = 5.502;
float x = q;
```

Quelle sont les valeurs des variables n, p, q et x ?

La valeur de n est 5 car la variable a été initialisée.

La valeur de p est une valeur quelconque car elle a été seulement déclarée. Sa valeur est celle de la zone mémoire qui lui a été affectée par le système car le langage C n'impose pas de valeur par défaut à une variable.

La valeur de q est 5 car étant de type entier (`int`), seule la partie entière de la valeur d'initialisation est conservée. La valeur est tronquée (troncature)

La valeur de x est 5 car même si elle est de type `float`, lors de l'initialisation précédente la valeur conservée par q est 5 et non 5.502.

Exercice 2.7.
Indiquer pourquoi les lignes suivantes sont incorrectes et les corriger :

```
1.  -2 = x;
2.  (x -1) = 2;
3.  x = x+1
4.  y =+ 5;
```

1. L'instruction `-2 = x` est incorrecte. Il n'est pas possible d'affecter une valeur à autre chose qu'une variable. La bonne syntaxe est `x = -2`

2. L'instruction est incorrecte car la partie gauche du = est elle-même une instruction (`x - 1`).

3. Cette instruction est correcte et peut s'écrire de façon plus concise en `x++.`

4. Cette syntaxe est correcte. Lue correctement (y = +5), celle-ci affecte la valeur 5 à la variable y. Si l'idée de départ est d'incrémenter la variable y de 5, il aurait fallu écrire y += 5, avec l'opérateur += qui incrémente la variable présente à gauche par la valeur présente à droite.

Exercice 2.8.
Parmi les déclarations suivantes, lesquelles sont correctes et lesquelles sont fausses :

```
1.  float var, var1, VAR2;
2.  real x,y,z;
3.  int void, main;
4.  char RS-232;
5.  char 1carac, 2carac;
6.  int ma_variable;
7.  double var1; var2; var3;
```

1. L'instruction est correcte. 3 variables de type `float` sont déclarées.
2. L'instruction est incorrecte. Le type real n'existe pas (utiliser à la place `float` ou `double`).
3. `void` est un type (le type vide) et ne peut dont être accolé à un autre type (ici `int`). Il est cependant possible d'appeler une variable `main`, même si cela peut emmener des confusions.
4. L'instruction est incorrecte. Un nom de variable ne doit pas contenir de tiret.
5. L'instruction est incorrecte. Un nom de variable ne doit pas commencer par un chiffre.
6. L'instruction est correcte.
7. L'instruction est incorrecte. Les points-virgules après var1 et var2 doivent être des virgules.

3. Types simples

Exercice 3.1.
Les variables de type char sont codées sur 1 octet. Le nombre entier qui les représente correspond au codage ASCII du caractère correspondant. Les codes ASCII principaux sont :
'0' : 48, '1' : 49 etc,, 'A' : 65, 'B' : 66 etc,, 'a' : 97, 'b' : 98 etc.

Soit la séquence d'instructions suivante :

```
char c1, c2, c3;
c1 = '0'; c2 = 'A'; c3 = 'a';
```

Ecrire les octets représentant le stockage des valeurs : `c1, c2, c3, c1+2, c2+c3`.

Les valeurs et encodages binaires sont les suivants (ATTENTION, le bit de poids faible est à gauche) :

c1 = 48 - 00001100
c2 = 65 - 10000010
c3 = 97 - 10000110
c1+2 = 50 - 01001100
c2+c3 = 162 – 01000101

Les opérations binaires se font de bit à bit, en propageant la retenue du bit de poids faible vers le bit de poids fort. Pour rappel :

0+0 = 0
0+1 = 1
1+0 = 0
1+1 = 0 (avec une retenue de 1)

Un programme en C pour afficher les résultats :

```
#include <stdio.h>

void affichebin(char n)
{
        unsigned bit = 0 ;
        unsigned mask = 1 ;
        for (int i = 0 ; i < sizeof(char)*8 ; ++i)
        {
                bit = (n & mask) >> i ;
                printf("%d", bit) ;
                mask <<= 1 ;
        }
}

int main()
{
    char c1, c2, c3;
    c1 = '0'; c2 = 'A'; c3 = 'a';

    printf("c1    = %d - ", c1);
    affichebin(c1);
    printf("\n");
```

```
    printf("c2    = %d - ", c2);
    affichebin(c2);
    printf("\n");

    printf("c3    = %d - ", c3);
    affichebin(c3);
    printf("\n");

    printf("c1+2  = %d - ", c1+2);
    affichebin(c1+2);
    printf("\n");

    printf("c2+c3 = %d - ", c2+c3);
    affichebin(c2+c3);
    printf("\n");

    return 0;
}
```

Exercice 3.2.

1. Donner la représentation de l'entier 18 dans le type short (2 octets).

2. Pour tout nombre x positif représenté en base 2, la représentation en base 2 de son opposé -x se construit en allant du bit de poids faible vers le bit de poids fort : on laisse les premiers zéros puis le premier 1 inchangés, on intervertit ensuite tous les autres chiffres. (Remarque, pour le zéro (0000 0000) il n'y a pas de premier un et donc on ne fait rien. En déduire la représentation de -18 en base 2.

3. On rappelle que le type char permet en fait de représenter des entiers signés sur un seul octet ; quel est l'intervalle des entiers représentables dans ce type ?

4. Soit un programme en C :

```
int main()
{
  char c = 'a';
  char d = c+c;
}
```

Que valent c et d à la fin du programme ? Pourquoi ?

1. La valeur 18 se représente en base 2 sur 2 octets par : 0100100000000000 (Avec le bit de poids faible à gauche)

2. La valeur -18 se représente en base 2 comme ceci :

- A partir de la représentation de 18 : 0100100000000000
- Laisser le premiers 0 inchangé : **0**100100000000000
- Laisser le premier 1 inchangé : 0**1**00100000000000
- Inverser les autres bits : 01**11011111111111**
- Le résultat est donc : 0111011111111111

3. Sur 1 octet il est possible de représenter 2^8 soit 256 valeurs différentes. Dans le cas de valeur signées, on peut donc représenter les nombres de -127 à +127 en passant par 0.

4. En C les valeurs de variables de type char sont des entiers. Le caractère `‘a’` correspond à la valeur entière 97. De plus, la somme de valeurs binaires ne peut dépasser la capacité d’encodage du type (ici char). L’opération d’addition se traduit en binaire par :

```
   1    11   (retenues)
  10000110
+ 10000110
-----------
= 01000011
```