Chapitre 3 : Les bases du C, partie 2

Notes de cours éditées par Alexandre Blondin Massé
modifié par Rachid Kadouche
Construction et maintenance de logiciels
INF3135

Département d'informatique
Université du Québec à Montréal

28 janvier 2018

Table des matières

1. Opérateurs et conversions
2. Tableaux
3. Pointeurs
4. Chaînes de caractères
5. Fonctions

Opérateurs arithmétiques

Opérateur	Opération	Utilisation
+	addition	`x + y`
−	soustraction	`x - y`
*	multiplication	`x * y`
/	division	`x / y`
%	modulo	`x % y`

Lorsque les deux opérandes de la division sont des types entiers, alors la division est entière également.

Représentation interne

Représentation par le complément à deux :

	signe							
127 =	0	1	1	1	1	1	1	1
2 =	0	0	0	0	0	0	1	0
1 =	0	0	0	0	0	0	0	1
0 =	0	0	0	0	0	0	0	0
-1 =	1	1	1	1	1	1	1	1
-2 =	1	1	1	1	1	1	1	0
-127 =	1	0	0	0	0	0	0	1
-128 =	1	0	0	0	0	0	0	0

S'il y a débordement, il n'y a pas d'erreur :

```
signed char c = 127, c1 = c + 1;
printf("%d %d\n", c, c1);
// Affiche 127 -128
```

Opérateurs de comparaison et logiques

Opérateurs de comparaison

Opérateur	Opération	Utilisation
==	égalité	`x == y`
!=	inégalité	`x != y`
>	stricte supériorité	`x > y`
>=	supériorité	`x >= y`
<	stricte infériorité	`x < y`
<=	infériorité	`x <= y`

Opérateurs logiques

Opérateur	Opération	Utilisation
!	négation	`!x`
&&	et	`x && y`
||	ou	`x || y`

Opérateurs d'affectation et de séquençage

- =, +=, -=, *=, /=, %= ;

```
int x = 1, y, z, t;
t = y = x;      // Equivaut à t = (y = x)
x *= y + x;     // Equivaut à x = x * (y + x)
```

- Incrémentation et décrémentation : ++ et - ;

```
int x = 1, y, z;
y = x++;        // y = 1, x = 2
z = ++x;        // z = 3, x = 3
```

- Opération de séquençage : évalue d'abord les expressions et retourne la dernière.

```
int a = 1, b;
b = (a++, a + 2);
printf("%d\n", b);
```

Opérateur ternaire

```
<condition> ? <instruction si vrai> : <instruction si faux>
```

- Très utile pour alléger le code ;
- Très utilisé.

Quelles sont les valeurs affichées par le programme suivant ?

```c
#include <stdio.h>

int main() {
    int x = 1, y, z;
    y = (x-- == 0 ? 1 : 2);
    z = (++x == 1 ? 1 : 2);

    printf("%d %d\n", y, z);
}
```

Opérations bit à bit

Opérateur	Opération	Utilisation
&	et	`x & y`
|	ou	`x | y`
^	ou exclusif	`x ^ y`

- Très utile pour simuler les opérations ensemblistes de façon très compacte.

Types numériques de base

- Plusieurs conversions (*cast*) se font automatiquement ;
- Si un des opérandes est `long double`, alors le résultat est également `long double`.
- Sinon, si un des opérandes est `double`, alors le résultat est également `double`.
- Sinon, si un des opérandes est `float`, alors le résultat est également `float`.
- Bref, évitez de mélanger les types dans une même opération ou montrez les conversions de façon explicite.

Conversions implicites

Attention aux conversions implicites entre types signés et non signés.

```c
#include <stdio.h>

int main() {
    char x = -1, y = 20, v;
    unsigned char z = 254;
    unsigned short t;
    unsigned short u;

    t = x;
    u = y;
    v = z;
    printf("%d %d %d\n", t, u, v);
    // Affiche 65535 20 -2
}
```

Conversion de types

```c
#include <stdio.h>

int main() {
    unsigned char x = 255;
    printf("%d\n", x);
    // Affiche 255
    printf("%d\n", (signed char)x);
    // Affiche -1
    int y = 3, z = 4;
    printf("%d %f\n", z / y, ((float)z) / y);
    // Affiche 1 1.333333
}
```

Priorité des opérateurs

Arité	Associativité	Par priorité décroissante
2	gauche, droite	`( ), [ ]`
2	gauche, droite	`->, .`
1	droite, gauche	`!, ++, -, +, -, (int), *, &, sizeof`
2	gauche, droite	`*, /, %`
2	gauche, droite	`+, -`
2	gauche, droite	`<, <=, >, >=`
2	gauche, droite	`==, !=`
2	gauche, droite	`&&`
2	gauche, droite	`||`
3	gauche, droite	`? :`
1	droite, gauche	`=, +=, -=, *=, /=, %=`
2	gauche, droite	`,`

Tableaux

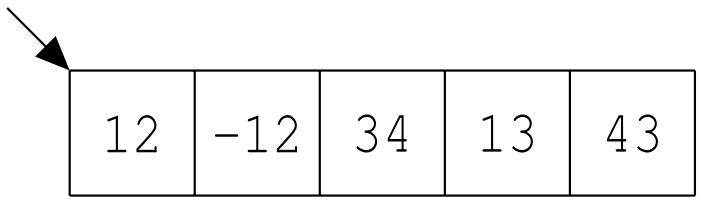
- Collection de données de même type ;
- Déclaration :

```
int donnees[10];
// Réserve 10 "cases" de type "int" en mémoire
int donnees[taille];
// Seulement avec C99 et allocation sur la pile
```

- Définition et initialisation :

```
int toto[] = {12,-12,34,13,43};
```

- Stockées de façon contiguë en mémoire ;



Accès

- À l'aide de l'opérateur [ ] :

```c
#include <stdio.h>

int main() {
    int donnees[] = {12,-12,34,13,43};
    int a, b;
    a = donnees[2];
    b = donnees[5];

    printf("%d %d\n", a, b);
        /* que vaut b ? */
}
```

- Le premier élément est à l'indice 0 ;
- S'il y a dépassement de borne, aucune erreur ou un avertissement (warning).
- Source fréquente de segfault.

Chaînes de caractères

- Les chaînes de caractères sont représentées par des tableaux de caractères ;
- Les chaînes constantes sont délimitées par les symboles de guillemets " ".
- Les deux déclarations suivantes sont équivalentes :

```
char chaine[] = "tomate";
char chaine[] = {'t','o','m','a','t','e','\0'};
```

- Termine par le caractère \0 ;
  - Longueur de la chaîne "tomate" : 6 ;
  - Taille du tableau de la chaîne "tomate" : 7.

Définition

- Un pointeur est l'adresse d'une donnée en mémoire ;
- On déclare un pointeur en utilisant le symbole * ;

```
int *p; // Un pointeur vers un entier
```

- L'opérateur & retourne l'adresse d'une donnée en mémoire :

```
#include <stdio.h>

int main() {
    int x = 210;
    printf("La variable x vaut %d et est stockée
            à l'adresse hexa %p.\n", x, &x);
}
```

Affiche : `La variable x vaut 210 et est stockée à l'adresse 0x7fff5fbff73c.`

Exemple

```c
#include <stdio.h>

int main() {
    int *pi, x = 104;
    pi = &x;
    printf("x vaut %d et se trouve à l'adresse %p\n", x, &x);
    printf("pi vaut %p et pointe sur la valeur %d\n", pi, *pi);

    *pi = 350;
    printf("x vaut %d et se trouve à l'adresse %p\n", x, &x);
    printf("pi vaut %p et pointe sur la valeur %d\n", pi, *pi);
}
```

Affiche :

```
x vaut 104 et se trouve à l'adresse 0x7fff5fbff73c
pi vaut 0x7fff5fbff73c et pointe sur la valeur 104
x vaut 350 et se trouve à l'adresse 0x7fff5fbff73c
pi vaut 0x7fff5fbff73c et pointe sur la valeur 350
```

Affectation

- Impossible d'affecter directement une adresse à un pointeur :

```
int *pi;
pi = 0xdff1;          /* interdit */
```

- Par contre, avec une conversion explicite, c'est possible :

```
int *pi;
pi = (int*)0xdff1; /* permis, mais a eviter */
```

- On peut aussi utiliser une conversion pour associer une même adresse à des pointeurs de types différents :

```
int *pi;
char *pc;
pi = (int*)0xdff1;
pc = (char*)pi;
```

Lien entre tableaux et pointeurs

- Un tableau d'éléments de type *t* peut être vu comme un pointeur constant vers des valeurs de type *t* ;
- Exemple : `int a[3]` définit un pointeur `a` vers des entiers ;
- De plus, `a` pointe vers le premier élément du tableau :

```
#include <stdio.h>

int main() {
    int a[3] = {1,2,3}, *pi;
    pi = a;              /* initialisation de pi */
    printf("%p %p %d %d %d %d\n",
           a, pi, a[0], a[1], a[2], *pi);
}
```

- Affiche : `0x7fff5fbff720 0x7fff5fbff720 1 2 3 1`
- `pi = a` est valide, mais `a = pi` n'est pas valide.

Opération sur les pointeurs

- Considérons un tableau `tab` de `n` éléments. Alors
  - `tab` correspond à l'adresse de `tab[0]` ;
  - `tab + 1` correspond à l'adresse de `tab[1]` ;
  - ...
  - `tab + n - 1` correspond à l'adresse de `tab[n - 1]` ;
- On peut calculer la différence entre deux pointeurs de même type ;
- De la même façon, l'incrémentation et la décrémentation de pointeurs sont possibles ;
- Finalement, deux pointeurs peuvent être comparés.

Exemple

```c
#include <stdio.h>

int main() {
    int a[3] = {1, -1, 2}, *pi, *pi2;
    pi = a;
    pi2 = &a[2];
    printf("%d ", pi2 - pi);
    printf("%d ", *(--pi2));
    printf("%d\n", *(pi + 1));
    if (pi + 1 == pi2)
        printf("pi+1 et pi2 pointent vers la même
                case mémoire.\n");
}
```

Affiche :

```
2 -1 -1 pi+1 et pi2 pointent vers la même case mémoire.
```

Gestion de la mémoire

```
int *pi, tab[10];
```

- La déclaration d'un tableau réserve l'espace mémoire nécessaire pour stocker le tableau ;
- La déclaration de `*pi` ne réserve aucun espace mémoire (sauf l'espace pour stocker l'adresse pointée) ;
- Les expressions

```
*pi = 6;
*(pi + 1) = 5;
```

sont valides, mais ne réservent pas l'espace mémoire correspondant. Autrement dit, le compilateur pourrait éventuellement utiliser cet espace.

Chaînes de caractères

- Une chaîne de caractères est représentée par un tableau de caractères terminant par le caractère `\0` ;

t	o	m	a	t	e	\0

- Des fonctions élémentaires sur les caractères se trouvent dans la bibliothèque `ctype.h` ;
- D'autre part, la bibliothèque standard `string.h` fournit plusieurs fonctions permettant de manipuler les chaînes de caractères.

La bibliothèque <ctype.h>

Fonction	Description
`int isalpha(c)`	Retourne une valeur non nulle si `c` est alphabétique, 0 sinon
`int isupper(c)`	Retourne une valeur non nulle si `c` est majuscule, 0 sinon
`int islower(c)`	Retourne une valeur non nulle si `c` est minuscule, 0 sinon
`int isdigit(c)`	Retourne une valeur non nulle si `c` est un chiffre, 0 sinon
`int isalnum(c)`	Retourne `isalpha(c) || isdigit(c)`
`int isspace(c)`	Retourne une valeur non nulle si `c` est un espace, un saut de ligne, un caractère de tabulation, etc.
`char toupper(c)`	Retourne la lettre majuscule correspondant à `c`
`char tolower(c)`	Retourne la lettre minuscule correspondant à `c`

Attention ! Les fonctions `toupper`, `tolower`, etc. sont définies sur les caractères et non sur les chaînes.

La bibliothèque <string.h>

- La fonction `unsigned int strlen(char *s)` retourne la longueur d'une chaîne de caractères ;
- La fonction `int strcmp(char *s, char *t)` retourne
  - une valeur négative si `s < t` selon l'ordre lexicographique ;
  - une valeur positive si `s > t` ;
  - la valeur 0 si `s == t`.
- Quelle est la différence entre `s == t` et `strcmp(s, t)==0` ?

Exemple

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

int main() {
    char s[] = "bonjour";
    char t[] = "patate";

    printf("Longueur de \"%s\" et \"%s\" : %lu, %lu\n",
           s, t, strlen(s), strlen(t));
    printf("strcmp(\"%s\", \"%s\") : %d\n", s, t,
           strcmp(s, t));
    return 0;
}
```

Sortie :

```
Longueur de "bonjour" et "patate" : 7, 6
strcmp("bonjour", "patate") : -14
```

Concaténation

- Les fonctions

```
char *strcat(char *s, const char *t);
char *strncat(char *s, const char *t, int n);
```

  permettent de concaténer deux chaînes de caractères ;
- Plus précisément, la chaîne `t` est ajoutée à la fin de la chaîne `s` ainsi qu'un caractère `\0` ;
- La chaîne `s` doit avoir une capacité suffisante pour contenir le résultat de la concaténation ;
- Le paramètre `n` donne une limite maximale du nombre de caractères à concaténer.

Exercices

Quel résultat donne le code suivant ?

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

int main() {
    char s[10] = "Salut ";
    char t[] = "toi!";
    strcat(s, t);
    printf("%s", s);
    return 0;
}
```

Copie

- Les fonctions

```
char *strcpy(char *s, const char *t);
char *strncpy(char *s, const char *t, int n);
```

  permettent de copier une chaîne de caractère dans une autre ;
- Dans ce cas, la chaîne `t` est copiée dans la chaîne `s` et un caractère `\0` est ajouté à la fin ;
- Comme pour `strcat`, la chaîne `s` doit avoir une capacité suffisante pour contenir la copie ;
- Le paramètre `n` donne une limite maximale du nombre de caractères à copier ;
- Quelle est la différence entre `s = t` et `strcpy(s, t)` ?

Segmentation d'une chaîne

- La fonction

```
char *strchr(const char *s, int c);
```

retourne un pointeur vers la première occurrence du caractère `c` dans `s` si elle existe, sinon retourne `NULL`.

- La fonction

```
char *strtok(char *s, const char *delim);
```

permet de décomposer une chaîne de caractères en plus petites chaînes délimitées par des caractèreres donnés ;

- Le paramètre `s` correspond à la chaîne qu'on souhaite segmenter, alors que le paramètre `delim` donne la liste des caractères considérés comme délimiteurs ;
- Très utile lorsqu'on souhaite extraire des données d'un fichier texte.

Décomposition avec champs vides

- La fonction strtok ne gère pas les cas où certains champs sont vides ;
- Par exemple, si les données sont

```
"124:41:3::23:10"
```

il ne sera pas détecté qu'il y a une donnée manquante entre `3` et `23` ;

- La fonction

```
char *strsep(char **s, const char *delims);
```

résoud ce problème.

- Attention ! Les fonctions strtok et strsep modifient la chaîne `s`.

Exemple (1/2)

```c
#include <stdio.h>
#include <string.h>
#define DELIMS ":"

int main() {
    char s[80];
    char *pc, *ps;

    strcpy(s, "124:41:3::23:10");
    printf("Avec strstok:\n");
    pc = strtok(s, DELIMS);
    while (pc != NULL) {
        printf("/%s/\n", pc);
        pc = strtok(NULL, DELIMS);
    }

    strcpy(s, "124:41:3::23:10");
    printf("Avec strsep:\n");
    ps = s;
    while ((pc = strsep(&ps, DELIMS)) != NULL) {
        printf("/%s/\n", pc);
    }
```

Exemple (2/2)

Résultat :

```
Avec strstok:
/124/
/41/
/3/
/23/
/10/
Avec strsep:
/124/
/41/
/3/
//
/23/
/10/
```

Utilité des fonctions

- Elles sont l'unité de base de programmation ;
- Chaque fonction doit effectuer une tâche bien précise ;
- Elles permettent d'appliquer la stratégie diviser-pour-régner ;
- Elles sont à la base de la réutilisation ;
- Elles favorisent la maintenance du code ;
- Lorsqu'elles sont appelées, l'exécution du bloc appelant est suspendue jusqu'à ce que l'instruction `return` ou la fin de la fonction soit atteinte.

Types de fonctions

On distingue deux catégories :

- Les fonctions pures :
  - Le résultat ne dépend que des arguments ;
  - Pas d'effet de bord ;
  - Par exemple, les fonctions mathématiques.
- Les fonctions non pures :
  - Le résultat dépend de l'environnement ou modifie l'environnement ou a des effets de bord ;
  - Par exemple, les fonctions d'allocation dynamique, les fonctions utilisant des variables globales.

Arguments et paramètres

```
int max(int x, int y) {
    if (x >= y) return x;
    else return y;
}

printf(max(3, 4));
```

- Un paramètre d'une fonction est une variable formelle utilisée dans cette fonction (ex : `x` et `y`) ;
- Un argument de fonction est une valeur passée à une fonction lors de son appel (ex : `3` et `4`) ;
- Les fonctions ont un, aucun ou plusieurs paramètres d'entrée ;
- Elles renvoient au plus un résultat en sortie.

Documentation d'une fonction

- Bien qu'il n'y ait pas de standard de documentation en C, on utilise souvent le standard Javadoc :
- Aussi, si la déclaration (du prototype) et l'implémentation sont séparées, on documente plutôt la première.

```
/**
 * Calcule la n-ième puissance de x.
 *
 * La n-ième puissance d'un nombre réel x, n étant un entier
 * positif, est le produit de ce nombre avec lui-même répété
 * n fois. Par convention, si n = 0, alors on obtient 1.0.
 *
 * @param x  Le nombre dont on souhaite calculer la puissance
 * @param n  L'exposant de la puissance
 * @return   Le nombre x élevé à la puissance n
 */
float puissance(float x, unsigned int n);
```

Passage par valeur

- Les types de base sont passés par valeur ;
- Une copie de la valeur est transmise à la fonction ;
- La modification de cette valeur à l'intérieur de la fonction n'affecte pas celle du bloc appelant.

```c
#include <stdio.h>

int carre(int x) {
    x *= x;
    return x;
}

int main() {
    int x = 2;
    printf("%d %d\n", carre(x), x);
}
```

Exercice

Quelles sont les valeurs affichées par ce programme ?

```c
#include <stdio.h>

void echanger(int a, int b) {
    int z = a;
    a = b;
    b = z;
}

int main() {
    int a = 5, b = 6;
    echanger(a, b);
    printf("%d %d\n", a, b);
}
```

Passage par adresse

Correction du programme :

```c
#include <stdio.h>

void echanger(int *a, int *b) {
    int z = *a;
    *a = *b;
    *b = z;
}

int main() {
    int a = 5, b = 6;
    echanger(&a, &b);
    printf("%d %d\n", a, b);
}
```

Passage d'un tableau

- Les tableaux peuvent être arguments d'une fonction :

```
float produit_scalaire(float a[], float b[], int d);
```

- Un tableau est représenté par un pointeur constant ;
- Il est donc passé par adresse lors de l'appel d'une fonction ;
- Si la fonction n'est pas supposée modifier le tableau qu'elle reçoit en paramètre, il est convenable d'utiliser le mot réservé `const`.

```
float produit_scalaire(const float a[], const float b[],
                       int d);
```

Exemple

```c
#include <stdio.h>

float produit_scalaire(const float a[], const float b[],
                       unsigned taille) {
    int i;
    float p = 0.0;
    for (i = 0; i < taille; i++) {
        p += a[i] * b[i];
    }
    return p;
}

int main() {
    float u[] = {1.0, -2.0, 0.0};
    float v[] = {-1.0, 1.0, 3.0};
    printf("%f\n", produit_scalaire(u, v, 3));
}
```

Affiche : `-3.000000`

Fonction retournant un tableau 1/2

- Une fonction ne peut pas retourner un pointeur créé dans la fonction, sauf s'il y a eu allocation dynamique ;
- En particulier, on ne peut pas retourner un tableau comme résultat. Il faut plutôt que le tableau soit un des arguments de la fonction.

```
#include <stdio.h>

int* initialise_tableau(unsigned taille) {
    int tableau[taille];
    int i;
    for (i = 0; i < taille; ++i)
        tableau[i] = 0;
    return tableau;
}

int main() {
    int *tableau;
    tableau = initialise_tableau(4);
    printf("%d\n", tableau[0]);
}
```

Affiche :

```
exemple1.c: In function 'initialise_tableau': exemple1.c:7:9:
warning: function returns address of local variable
[-Wreturn-local-addr] return tableau; Erreur de segmentation (core
dumped)
```

Fonction retournant un tableau 2/2

```c
#include <stdio.h>

void  initialise_tableau(int tableau [], unsigned int taille) {
    for (unsigned int i = 0; i < taille; ++i)
        tableau[i] = 0;
}

int main() {
    int tableau[4];
    initialise_tableau(tableau,4);
    printf("%d\n", tableau[0]);
}
```

Affiche :

0

Déclaration et définition des fonctions

- C'est une bonne pratique de déclarer les prototypes des fonctions au début du fichier où elles sont définies et/ou utilisées ;
- Il n'est alors pas nécessaire, mais tout de même encouragé de donner un nom aux variables ;
- Lors de la définition, le nom des variables est évidemment obligatoire.
- Contrairement à C++ et Java, la surcharge de fonctions est interdite :

```
int max(int x, int y);
int max(int x);
```

```
test.c:2: error: conflicting types for 'max'
test.c:1: error: previous declaration of 'max' was here
```

Visibilité des variables et des fonctions

- La visibilité des variables en C est plus complexe lorsqu'on manipule plusieurs fichiers ;
- D'abord, il y a les variables locales (ou automatiques), dont la portée est limitée à un bloc donné ;
- Ensuite, il y a les variables globales qui sont visibles dès leur déclaration et ce, jusqu'à la fin du fichier ;
- Il est également possible de définir des variables globales à plusieurs fichiers, par l'intermédiaire du mot réservé `extern` ;
- Par opposition aux variables externes, les variables statiques, déclarées à l'aide du mot réservé `static`, ont une portée limitée au fichier dans lequel elles sont déclarées.

Variables locales (ou automatiques)

- Visibles dès leur déclaration jusqu'à la fin du bloc où elles sont définies ;
- Accessibles uniquement dans leur bloc ;
- Supprimées lorsqu'on sort du bloc ;
- Aucune garantie sur leur valeur lorsqu'elles ne sont pas initialisées.

Quelles valeurs sont affichées par le programme suivant ?

```
int main() {
    int i = 0, j = 0;
    if (i == 0) {
        int i = 5;
        printf("%d ", i);
    }
    printf("%d\n", i);
}
```

Variables et fonctions globales

- Visibles de leur déclaration jusqu'à la fin du fichier où elles sont définies ;
- Utilisables jusqu'à la fin du programme ;
- Initialisées à 0 par défaut ;
- Les fonctions ont la même visibilité, accessibilité et durée de vie que les variables globales.

Fichier `main.c`

```c
#include <stdio.h>
#include "math.c"

int main() {
    printf("PI = %f\n", PI);
    printf("Le carre de %d est %d\n",
           4, carre(4));
}
```

Affiche :

```
PI = 3.141593
Le carre de 4 est 16
```

Fichier `math.c`

```c
const float PI = 3.141592654;

int carre(int x) {
    return x * x;
}
```

La fonction `main`

- La fonction principale de tout programme C. C'est cette fonction que le compilateur recherche pour exécuter le programme ;
- La fonctin `main` d'un programme n'acceptant aucun argument est

```
int main();
```

- Par convention, la valeur de retour de la fonction `main` est `0` si tout s'est bien déroulé et un entier correspondant à un code d'erreur différent de `0` autrement.

Les arguments de la fonction `main`

- Lorsque la fonction `main` accepte des paramètres, elle est de la forme :

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

- `argc` correspond au nombre d'arguments (incluant le nom du programme) ;
- `argv` est un tableau de chaînes de caractères, vues comme des pointeurs.
- `argv[0]` est une chaîne de caractères représentant le nom du programme ;
- `argv[1]` est le premier argument, etc.

Récupération des arguments de la fonction `main`

- Fonctions provenant de la bibliothèque `stdlib.h` ;

```
double strtod(const char *chaine, char **fin);
unsigned long strtoul(const char *chaine, char **fin,
                      int base);
long strtol(const char *chaine, char **fin,
            int base);
...
```

- `chaine` : chaîne qu'on veut traiter ;
- `fin` : ce qui reste de la chaîne après traitement ;
- `base` : base dans laquelle le nombre est exprimé dans la chaîne ;
- Les fonctions `atof`, `atoi`, `atol`, etc. sont déconseillées, car elles ne permettent pas de valider si la conversion s'est bien déroulée.