Informatique de base-I

9 septembre 2014

Note Dans chaque exercice nous proposons un programme toujours structuré de la même manière (cependant certains éléments ne sont pas toujours présents) : #include, #define, prototypes de fonctions, fonction principale, et enfin définitions de fonctions. Cet ordre doit être considéré comme obligatoire. De plus le programme contient des commentaires (/* ... */ (ou simplement //)) et à la fin un exemple d'exécution sous la forme également d'un commentaire.

1 Algorithmique

L'**algorithmique** est la science qui décrit comment résoudre un problème donné par application d'un nombre fini d'**opérations** sur un certain nombre de données. Il est composé d'un nombre fini d'étapes, chaque étape étant constituée d'un nombre fini d'opérations élémentaires bien définies. Un algorithme est donc la description détaillée d'une méthode pour aboutir à un résultat. Une **recette de cuisine** est un algorithme.

Dans le domaine de l'informatique, l'algorithmique est la science qui permet de décrire une méthode pour calculer une certaine fonction de un ou plusieurs paramètres, de produire des documents ou des fichiers en transformant des informations d'entrée... Ainsi, on applique un algorithme sur des **informations en entrée** et cela produit des **résultats**. Un ordinateur est la machine qui effectue ces *transformations* :

$$\text{Entrées} \rightarrow \text{Ordinateur(Algorithme)} \rightarrow \text{Sortie}$$

Pour qu'un ordinateur puisse comprendre un algorithme, on utilise un **langage de programmation** (dans ce cours, le langage C). On doit donc traduire les algorithmes en programmes informatiques. Ce programme peut alors être exécuté par l'ordinateur. Ce dernier possède différentes parties :

- Une **unité de calcul** qui peut effectuer un certain nombre d'opérations élémentaires.
- Une **mémoire** pour contenir et manipuler les **instructions** qui composent les programmes et les *données* manipulées par ces programmes.
- Une unité qui décode les instructions des programmes et qui possède un **index** sur l'instruction courante et un autre sur l'appel courant de la pile des appels et des variables locales.

– Des **périphériques** (avec un ensemble de sous-programmes appelé système d'exploitation) pour stocker/récupérer des informations (sous forme de fichiers) : disque dur, disquette, accès réseau ; entrer des informations au clavier, afficher des résultats sur l'écran, gérer la souris, les fenêtres, utiliser une carte son...

2 UNIX et C

Nous présentons ici les éléments du système d'exploitation UNIX permettant d'écrire et exécuter des programmes.

2.1 Historique

En 1969 K. THompson développe le système UNIX, premier système multi-utilisateur et multi-tâche générique, c'est-à-dire conçu indépendamment d'une machine particulière.

En 1970 D. RItchie introduit le langage C, un langage permettant d'écrire des programmes qui sont ensuite compilés en programmes constitués d'instructions exécutables par l'ordinateur.

En 1990 la première norme ANSI définit strictement le langage : les programmes écrits en respectant la norme doivent pouvoir être compilés et exécutés, en donnant le même résultat quel que soit le compilateur et la machine. Un compilateur particulier peut avoir, en dehors de la norme, des comportements particuliers, mais il est déconseillé de les utiliser.

L'utilisateur interagit avec l'ordinateur via un programme particulier, un interpréteur : l'utilsateur écrit une ligne de commande, celle-ci est interprétée et exécutée par la machine. Le principal langage de commande sous unix est le bash.

3 Langage

3.1 Valeurs et types

Toutes les valeurs que nous manipulerons appartiennent à un type. Ci-dessous les types élémentaires courants et deux tableaux (un tableau est un cas de type *composé*) :

Algorithme	Programme C
Entier	`int` (et `long`) 1, -333, ...
Réel (flottant)	`double` (et `float`) `1.0 -3e20`
Caractère (`'a'`)	`char 'a' '\n'`
Booléen (Vrai/Faux)	`int` (Faux = 0, Vrai sinon)
Tableau [10] de Réels	`double [10]`
Chaîne (= Tableau)	`char[] "ceci est une chaine"`

3.2 Variables et expressions

Une **variable est une zone de la mémoire** dans laquelle on peut stocker des données. La taille de cette zone dépend du type de la donnée (et pas de la valeur qui est stockée). Par exemple, un *int* se range dans une zone de 4 octets, un *double* dans une zone de 8 octets, un tableau à une dimension d'*int* dont l'indice varie de 0 à 9 comporte $4 * 10 = 40$ octets. On les désigne par un nom appelé **identificateur** (de variable).

Les variables et les constantes peuvent alors être composées à l'aide d'opérateurs pour former des expressions. Ces expressions sont calculées en remplaçant les variables par la valeur qui est stockée dans la zone mémoire associée à ces variables et en effectuant les opérations élémentaires :

- 4 → 4
- x → 10 (si 10 est stocké dans x)
- 4 + X * X ou 4 + (X × X) → 104
- (4+X) * X → 400
- r * cos(t) → -2.0 (si r = 2 et t = π)
- x < 10 ou x > 20 → Faux

Opérations et fonctions disponibles :

+ - * × / %

t[i]

f() sin() cos() ...

== != >=

...

3.3 Actions élémentaires

	Algorithme	Programme C
L'affectation d'une variable	V ← E	V = E ;
Lecture d'une variable depuis le clavier	Lire(adressede V)	`scanf("%d", &E);`
Écriture d'une valeur sur l'écran	Ecrire(E)	`printf("La valeur de E est %d", E);`

3.4 Structures de contrôle

	Algorithme	Programme C
Séquence d'instructions	A1 A2 ... An	A1 A2 ... An

Instructions conditionnelles	Si (E) alors
	  A1
	  ...
	Sinon
	  B1
	  ...
	FinSi
		`if (expr)`
		`{`
		  A1
		  ...
		`}`
		`else`
		`{`
		  B1
		  ...
		`}`

Itérations	Pour V de E1 à E2 faire
	  A1
	  ...
	Fin Pour
		`for(V=E1; V <= E2;  V = V+1)`
		`{`
		  A1
		  ...
		`}`

Boucle Tantque	Tant que (E) faire
	  ...
	FinTantque
		`while (E)`
		`{`
		  ...
		`}`

Boucle Répéter tantque	Répéter
	  ...
	Tantque (E)
		`do`
		`{`
		  ...
		`}`
		`while (E)`

3.5 Algorithme

Un algorithme est composé de plusieurs parties :

Description	Algorithmique	C
En tête :	Algorithme Exemple	`#include<stdio.h>`
		`#include<stdlib.h>`
		`#include<math.h>`
Constantes :	Constante Pi = 3.1416	`#define Pi 3.1416`
Fonctions f :	Fonction f(x :Réel) :Réel	`double f(double x)`
		`{`
Corps de f :	Retourner 1/x	`return 1/x;`
Fin de f :	FinFonction	`}`
Fonction principale	Fonction principale(Rien) :Entier	`int main(void)`
		`{`
Variables :	Variable c, s : Réel	`double c,s;`
	Variable n,i : Entier	`int n,i;`
Corps :	Écrire("Calcul cos et f...")	`printf("...");`
	Écrire("Valeur de N ?")	`printf("...");`
	Lire(adressede n)	`scanf("%d", &n);`
	i ← 1	`i = 1;`
	Tantque (i ≤ n) faire	`while (i<=n)`
		`{`
	c ← cos(Pi/i)	`c = cos(Pi/i);`
	s ← f(Pi/i)	`s = f(Pi/i);`
	Écrire("Cos =", c, "f = ", s)	`printf("Cos = %g f = %g\n", c, s);`
	i ← i+1	`i=i+1;`
	FinTantque	`}`
	retourner un entier (succès)	`return EXIT_SUCCESS;`
Fin	FinFonction	`}`

La constante EXIT_SUCCESS est définie dans la bibliothèque *stdlib* et a la valeur 0. Par la suite nous utiliserons indifféremment 0 ou EXIT_SUCCESS.

3.6 Compilation et exécution

Le texte d'un programme en C va être compilé en un fichier *exécutable* par la machine. Ci-dessous un programme en C qui demande les valeurs initiales de deux variables a et b , échange les valeurs de ces variables, puis affiche les valeurs de ces variables.

```
/*
 *  ex3_1.c
 */
# include <stdio.h>
# include <stdlib.h>
int main (){
        int a,b,aux;
        printf("a et b?");
        scanf("%d",&a);
        scanf("%d",&b);
```

```
        aux=a;
        a=b;
        b=aux;
        printf("a =%d, b=%d\n", a,b);
        return EXIT_SUCCESS;
}
```

Il est dans un fichier texte dont le nom est `ex3_1.c` qui se trouve à une certaine adresse dans la hiérarchie, ici `/Users/soldano/dufee/ex3_1.c`. Le répertoire dans lequel il se trouve est `/Users/soldano/dufee`.
`~/dufee` est le même répertoire : il se trouve dans mon répertoire personnel `/Users/soldano` qui est aussi connu par le système comme `~`. Dans ce qui suit la commande `cd ~/dufee` permet de se positionner dans le répertoire `/Users/soldano/dufee`, c'est-à-dire de faire de ce répertoire le répertoire *courant*. La commande `pwd` a comme résultat l'adresse *absolue* du répertoire courant. La commande `gcc ex3_1.c -o ex3_1` a comme premier argument l'adresse *relative* au répertoire courant du fichier `ex3_1.c` , et comme *option* `-o ex3_1`. Le résultat de la commande est la création d'un fichier `ex3_1`, qui est *exécutable*, dans le répertoire courant. Autrement dit `gcc` compile le fichier texte `ex3_1.c` et construit un fichier exécutable. `./ex3_1` est l'adresse absolue de ce fichier (rappelons que `./` est une manière de désigner `/Users/soldano/dufee`). Comme ce fichier est exécutable, il peut être utilisé comme une commande. Le résultat est une interaction entre la machine et l'utilisateur. Ici la machine affiche `a et b?` et l'utilisateur entre `2 3`. La machine affiche alors `a =3, b=2`. Remarquons qu'une fois l'exécutable `ex3_1` obtenu celui-ci peut être utilisé plusieurs fois sans recréer par `gcc` un autre exécutable. Ci-dessous `ex3_1` est utilisé aussi avec les valeurs `a =4, b=3`.

```
bash−3.2 cd ~/dufee
bash−3.2 pwd
/Users/soldano/dufee
bash−3.2 gcc ex3_1.c −o ex3_1
bash−3.2 ./ex3_1
a et b? 2 3
a =3, b=2
bash−3.2 ./ex3_1
a et b? 4 3
a =3, b=4
bash−3.2
```

L'édition du texte se fait soit avec une commande (comme `nano`) soit avec un éditeur ayant une interface graphique (comme l'éditeur de l'environnement `KdevelopC/C++`).

Lorsque le texte `ex3_1.c` est écrit, la commande `gcc ex3_1.c -o ex3_1` permet de créer le fichier exécutable `ex3_1` mais parfois la commande échoue et `ex3_1` n'est pas créé. En effet le texte `ex3_1.c` est *compilé* (traduit en langage machine), et des erreurs peuvent apparaître à ce niveau. On corrige alors les erreurs dans le texte `ex3_1.c`, on enregistre ensuite `ex3_1.c`, puis on exécute de nouveau la commande `gcc` pour obtenir l'exécutable.

3.7 Simulation

Une simulation permet de **jouer le rôle de l'ordinateur**. L'ordinateur possède un algorithme (un programme) qu'il exécute. Pour représenter les étapes successives exécutées par l'ordinateur, on utilise une notation dans laquelle le résultat de chaque action et surtout son effet sur l'état de la mémoire est mentionné, et en particulier, **la modification** des variables (par la suite, nous verrons qu'on peut également créer et détruire des variables). Tous ces effets sont notés entre accolades. On indente ce qui suit l'appel d'une fonction, les valeurs des arguments et des variables locales sont données ensuite, et la valeur de retour est donnée après être revenu à l'indentation précédente.

{a=?, b=?, aux=? /∗ *les valeurs des variables a,b et c est inconnue* ∗/}
{affichage de "a et b ?"}
{lecture a=2}
{lecture b=3}
{aux=2 /∗ *la valeur de a est rangée dans aux* ∗/}
{a=3 /∗ *la valeur de b est rangée dans a* ∗/}
{b=2 /∗ *la valeur de aux est rangée dans b* ∗/}
{affichage de "a=3, b=2" ∗/}

3.8 Affectation

Elle est notée `=` en langage C, et ← en langage algorithmique.
nom ← *expression* signifie :

1. *expression* est évaluée, c'est-à-dire renvoie une *valeur* v
2. La nouvelle valeur de la variable *nom* est la valeur v

Ainsi, les instructions suivantes :

```
a=3;
b=5;
a = a+2*a+b;
```

ont pour effet :

1. de ranger dans `a` la valeur `3`
2. de ranger dans `b` la valeur `5`
3. d'évaluer l'expression `a+2*a+b`, ce qui donne ici 3 +2*6 + 5 =14 et de ranger dans `a` la valeur `14`

Remarque : l'affectation est une opération donc les parties gauche et droite sont les arguments, et renvoie donc une valeur que nous n'utiliserons pas.

3.9 Conditions composées

Il y a deux opérateurs logiques binaires (ET, OU) et un opérateur logique uniare (NON) permettant de construire des conditions composées dans les structures conditionnelles. Les opérateurs ont les signatures suivantes :

– ET (`&&` en *C*) : $Bool \times Bool \rightarrow Bool$

– OU (`||` en C) : $Bool \times Bool \to Bool$
– NON (`!` en C) : $Bool \to Bool$

Ces opérateurs ont les propriétés habituelles :
– $NON\ A$ est Vrai si et seulement si A est Faux
– $NON\ (A_1\ \ OU\ A_2)$ est équivalent à $(\ NON\ A_1)\ ET\ (\ NON\ A_2)$

On a également les opérateurs relationnels $\geq$ (`>=`), $>$ (`>`), $\leq$ (`<=`), $<$ (`<`) et $=$ (`==`). Attention, en C, `=` est l'opérateur d'affectation, et `==` renvoie vrai si ses deux arguments sont égaux.

Considérons la suite d'instructions suivante :

```
a=3;
b=5;
a = a+2*a+b;
if (a==a+2*a+b || b>=a+2*a+b) {
        a=0;
}
else{
        a=1;
}
printf("%d\n", a);
```

La simulation se déroule ainsi :

{a=3, b=5} /*
{*a=3 +2*6 + 5 =14*}
{*a=14 égal a+2*a+b=14+2*14+5=47\ est Faux*},
{*b=5 supérieur ou égal à a+2*a+b=14+2*14+5=47\ est Faux*}
{*donc la condition (a==a+2*a+b || b>=a+2*a+b) est Fausse*}
{*a=1*}
{ *affichage de 1*}

3.10 Exemple d'utilisation de la structure `for`

La structure `for` met en jeu une instruction `inst1` exécutée avant la première itération, une condition `cond` pour poursuivre l'itération, et une instruction `inst2` à chaque fin d'exécution du `corps` de la structure. Sa syntaxe est :

```
for (inst1; cond; inst2) {
        corps
}
```

et elle correspond exactement à la structure `while` suivante :

```
inst1;
while (cond) {
        corps;
        inst2;
}
```

Elle est particulièrement utile pour faire des sommes ou des produits comme le montre le programme suivant qui calcule le nombre $C(n,p)$ de combinaisons

de p éléments parmi n en utilisant $C(n,p) = \frac{n!}{(p!)(n-p)!}$ et $n! = 1 * \cdots * n$. Pour écrire chacune des boucles il suffit de remarquer que $0! = 1$ et pour tout j, $j! = 1 * \cdots * j = (j-1)! * j$. A chaque itération on utilise donc le produit partiel courant $(j-1)!$ pour calculer le prochain produit partiel $j!$:

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

int main(void){
        /* Affiche C(n,p)  (nombre de combinaisons de p parmi n) */
        int i,c,n,p,nf,nmpf,pf;
        printf("Calcul du nombre de combinaisons de p éléments parmi n\n");
        printf("nombre d'éléments n?");
        scanf("%d",&n);
        printf("nombre d'éléments p?");
        scanf("%d",&p);
                /* n !*/
        nf=1;
        for(i=1;i<=n;i=i+1){
                nf=nf*i;
        }
                /* p !*/
        pf=1;
        for(i=1;i<=p;i=i+1){
                pf=pf*i;
        }
                /* (n-p) !*/
        nmpf=1;
        for(i=1;i<=n-p;i=i+1){
                nmpf=nmpf*i;
        }
        c=nf/(pf*nmpf);
        printf("C(%d,%d)=%d\n",n,p,c );
        return EXIT_SUCCESS;
}
/*
G215-5:- gcc ex2.c -o ex2
G215-5:- ./ex2
Calcul du nombre de combinaisons de n éléments parmi p
nombre d'éléments n?3
nombre d'éléments p?2
C(3,2)=3
G215-5:-
```

On remarquera qu'une valeur initiale du produit est calculée avant la boucle, que `inst1` initialise un *compteur*, que `inst2` augmente la valeur du compteur pour préparer l'itération suivante.

3.11 Boucles imbriquées

Dans une boucle l'une des actions peut elle-même nécessiter une boucle.

Afficher les entiers entre 0 et $n^2 - 1$ en n lignes de n éléments

Ici afficher une ligne de n éléments nécessite une boucle. Voici une solution :

```
/*
 *  Duf_forfor_1.c
 */
# include <stdio.h>
# include <stdlib.h>
int main (){
        /* Affichage des entiers de 0 à n^2 -1 par ligne de n éléments */
        int s,i,j,n;
        /*lecture */
        printf("n?");
        scanf("%d",&n);
        /* Calcul et affichage */
        s=0;
        for(i=0;i<n;i=i+1){
                for(j=0;j<n;j=j+1){
                        printf("%3d ",s);
                        s=s+1;
                }
                printf("\n");
        }
        return EXIT_SUCCESS;
}
//dyn246:- ./duf_forfor_1
//n?4
//0   1   2   3
//4   5   6   7
//8   9  10  11
//12  13  14  15
//dyn246:-
```

On remarque que pour l'affichage on utilise `%kd`, où `k` est le nombre le nombre fixe de caractères pour afficher l'entier, ce qui permet d'avoir des colonnes alignées.

Afficher la moitié inférieure (diagonale incluse) de la matrice carrée $M(i,j) = i + j$, pour i et j variant entre 0 et $n - 1$

Chaque ligne nécessite une boucle mais ici le nombre d'itérations dépend de l'indice de la ligne : pour les indices de colonne plus grands que l'indice de ligne on n'affiche rien.

```
/*
```

```
 *   duf_forfor_2.c
 */
# include <stdio.h>
# include <stdlib.h>
int main (){
        /* Afficher la moitié inférieure (diagonale incluse) de la matrice carrée
         M(i,j)=i+j pour i et j variant entre 0 et n-1                          */
        int s,i,j,n;
        /* lecture */
        printf("n?");
        scanf("%d",&n);
        /* Calcul et affichage */
        for(i=0;i<n;i=i+1){
                for(j=0;j<=i;j=j+1){
                        printf("%3d ",i+j);
                }
                printf("\n");
        }
        return EXIT_SUCCESS;
}
//dyn246:- gcc duf_forfor_2.c -o duf_forfor_2
//dyn246:- ./duf_forfor_2
//n?5
//0
//1   2
//2   3   4
//3   4   5   6
//4   5   6   7   8
//dyn246:-
```

4 Tableaux

Un tableau est un ensemble structuré de variables. Les éléments sont tous de même type et accessibles par un (ou plusieurs indices). Les éléments d'un tableau sont rangés consécutivement (dans l'ordre des adresses) en mémoire. Lorsqu'on déclare un tableau, on doit préciser, par une (ou plusieurs) constantes la taille du tableau.

4.1 Tableau à une dimension

Une constante (`NMAX`) suffit à déterminer la place réservée pour le tableau. Celle-ci correspond à une taille maximale, déterminée dans le texte en C. Lors de l'exécution seule une partie *utile* représentée par une variable (`n` ici) est utilisée, et appelée la *taille* du tableau. Les indices du tableau vont de 0 à $taille - 1$.

Lecture d'un tableau d'entiers, représentant une note, et affichage de la note moyenne et de sa variance

```
/*  duf_tab_1.c */
# include <stdio.h>
# include <stdlib.h>
# define NMAX 100
int main (){
        /* Lecture d'un tableau d'entiers, représentant une note,
          et affichage de la note moyenne et de sa variance              */
        int s,i,j,n;
        float m, v;
        int t[NMAX];
        /*lecture */
        printf(" taille du tableau ?");
        scanf("%d",&n);
        /* Lecture */
        printf("%d éléments du tableau ?", n);
        for(i=0;i<n;i=i+1){
                scanf("%d",&t[i]);
        }
        /* Moyenne */
        s=0;
        for(i=0;i<n;i=i+1){
                s=s+t[i];
        }
        m=((float) s)/n;
        /* Variance */
        v=0;
        for(i=0;i<n;i=i+1){
                v=v+(t[i]-m)*(t[i]-m);
        }
        v=v/n;
        /* Affichage*/
        printf("moyenne=%f, variance= %f \n", m,v);
        return EXIT_SUCCESS;
}
//dyn246:- ./duf_tab_1
// taille du tableau ?6
//6 éléments du tableau ?2 2 2 3 3 3
//moyenne=2.500000, variance= 0.250000
//dyn246:-
```

On remarquera l'utilisation de `(float) s`, pour transformer le résultat de l'évaluation de s, un entier, en un réel. En effet si `s`=3 et `n`=2, `s/n` est la division d'un entier par un entier et donne un entier, c'est-à-dire 1. Cela vient de la signature de `/` : $int \times int \rightarrow int$.

L'opérateur unaire de *conversion* `(type1) val` transforme la valeur *val*, de type *type1* en une valeur de type *type2*. Ici `(float) s`, lorsque `s` a la valeur *int* 3, renvoie la valeur *float* 3.0. De ce fait `((float) s) / n` devient 3.0/2, ce qui donne le résultat correct 1.5.

4.2 Tableaux à plusieurs dimensions

Une matrice se représente par un tableau à deux dimensions : les lignes et les colonnes. En langage C, un tableau de nl lignes et nc colonnes d'entiers est décrit comme un tableau de nl éléments, chacun de ces éléments étant un tableau de nc éléments de type int. Là encore, l'allocation statique de la mémoire impose de surdimensionner les tableaux en bornant par des constantes $NLMAX$ et $NCMAX$ le nombre de lignes et de colonnes du tableau. Là encore il faudra spécifier les nombres de lignes et colonnes nl et nc réellement nécessaire à chaque tableau. On pourra ainsi représenter des matrices dont le nombre de lignes et de colonnes est variable et sera déterminé à l'exécution.

La déclaration d'un tableau à deux dimensions se fait de la manière suivante :

```
int tab[NLMAX][NCMAX];
```

Pour placer une valeur v à l'élément en i^{eme} ligne et j^{eme} colonne on écrira simplement :

```
tab[i][j]=v;
```

Cela signifie précisément que l'on considère le i^{eme} élément du tableau tab, c'est à dire $tab[i]$ et que celui-ci étant lui-même un tableau, on peut accéder à son j^{eme} élément, un entier.

Dans la suite nous représenterons ainsi des matrices. Nous utiliserons aussi cet exemple pour parler d'allocation dynamique, c'est à dire à l'exécution, de la mémoire d'un tableau.

5 Fonctions

Une *fonction*
- a comme effet de calculer une valeur *spécifiée* résultant d'un ensemble d'instructions
- dépend de la valeur de certains arguments,

La programmation est naturellement *modulaire*, et les fonctions permettent de structurer et d'éclaircir le sens du programme. Nous allons voir ci-dessous l'exemple d'une fonction calculant $n!$ et celui d'une fonction calculant le nombre C_n^p de combinaisons de p éléments parmi n.

Définition et utilisation d'une fonction factorielle `fact`
Voici un programme qui calcule $(p+1)! + (n-1)!$ *pour* $p+1 \geq 0$ *et* $n-1 \geq 0$:

```
/*  duf_fonc_1.c */
#include <stdio.h>
#include <stdlib.h>

/* prototypes */
int fact(int n);

/* fonction principale */
```

```
int main(){
/*      (p+1)!) * (n-1)! pour p >= -1 et n >=1 */
        int n,p, r;
        /* lecture des paramètres */
        printf("p et n? ");
        scanf("%d",&p);
        scanf("%d",&n);
        /* calcul */
        r= fact(p+1)*fact(n-1);
        /* Affichage */
        printf("somme= %d \n",r);
        return EXIT_SUCCESS;
}
/* définitions des autres fonctions */
int fact(int n){
        /* renvoie n! */
        int f,i;
        f=1;
        for (i=1;i<=n;i=i+1){
                f=f*i;
        }
        return f;
}
//dyn246:- gcc duf_fonc_1.c -o duf_fonc_1
//dyn246:- ./duf_fonc_1
//p et n? 3 4
//somme= 144
//dyn246:-
```

Plusieurs remarques sur cet exemple :

- Le *prototype* de la fonction `fact` permet de prévenir le compilateur (gcc) des caractéristiques de la fonction. Il est placé avant la définition de la fonction principale `main`. Le prototype est constitué de l'*en-tête* de la fonction et d'une fin d'instruction `";"`. L'en-tête est constitué du *type* du résultat de la fonction, du *nom* de la fonction et d'une suite de couples *(type, nom)* chacun associé à un argument de la fonction.
- Le *corps* de la fonction main contient deux *appels* à `fact` avec des arguments différents : `fact(p+1)` et `fact(n-1)`.
- La définition de la fonction `fact` est placée après la fin de la définition de la fonction `main`.
- Dans le corps de la fonction `main`, `fact(p+1)` * `fact(n-1)` est considéré comme une expression à évaluer. L'évaluation commence par l'évaluation de `fact(p+1)`. Pour cela l'argument de la fonction `p+1` est évalué, puis l'appel est exécuté et `fact(p+1)` est évalué. L'évaluation de `fact(n-1)` commence par l'évaluation de `n-1` puis par l'appel de `fact(n-1)`. Lorsque les deux valeurs ont été obtenues elles sont multipliées et le résultat est rangé dans `r`.
- on aurait pu écrire directement

printf("somme= %d \n",fact(p+1)*fact(n-1;

- Lors de l'appel de `fact(p+1)`, de l'espace mémoire est alloué aux *variables locales* à cet appel de la fonction : `n, f, i, j`. Celles-ci sont disponibles pendant l'exécution de cet appel et sont perdues par la suite. La variable `n` locale à l'appel `fact(p+1)` est différente, c'est-à-dire associée à une autre *adresse mémoire*, que la variable `n` locale à la fonction `main`.
- Au moment de l'appel `fact(p+1)`, la valeur v de l'expression `p+1` est transmise à la variable locale `n` associée à cet appel.

Ci dessous, une simulation de l'exécution en commentaire ci-dessus :

```
{p=3, n=4}
{fact(p+1) ?}
        {n=3+1=4}
        {f=1}
        ...
        {f=6*4}
{=6}
{fact(n-1) ?}
        {n=4-1=3}
        {f=1}
        ...
        {f=6*4}
{=6}
{r=6*6=36}
{affichage de "somme=36"}
```

Définition et utilisation d'une fonction `comb` qui utilise la fonction `fact`

Nous voulons écrire un programme quii vérifie l'identité $\mathcal{C}_n^p = \mathcal{C}_{n-1}^p + \mathcal{C}_{n-1}^{p-1}$ sur un exemple. Pour cela nous allons écrire une fonction `comb` telle que l'appel `comb(n,p)` renvoie $\mathcal{C}_n^p$, le nombre de combinaisons de p éléments parmi n. Nous savons que $\mathcal{C}_n^p = \frac{n!}{p!(n-p)!}$, nous allons donc utiliser la fonction `fact` définie ci-dessus :

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

/* prototypes */
int fact(int n);
int comb(int n, int p);

/* fonction principale
int main(){
        /* affiche Vrai si l'égalité C_n^p = C_{n-1}^p + C_{n-1}^{p-1} */
        /* est vérifiée et Faux sinon                                  */
        int n,p,i,j;
        /* lecture */
        printf("n? ");
```

```
        scanf("%d",&n);
        printf("p? ");
        scanf("%d",&p);

        /* calcul */
        if (comb(n,p) == comb(n -1,p)+comb(n -1,p -1)){
                printf("Vrai\n");
        }
        else{
                printf("Faux\n");
        }
        return EXIT_SUCCESS;
}
/* définitions des autres fonctions */
int fact(int n){
        /* renvoie n! */
        int f,i;
        f=1;
        for (i=1;i<=n;i=i+1){
                f=f*i;
        }
        return f;
}
int comb(int n, int p){
        /* renvoie C_n^p) */
        return fact(n)/(fact(p)*fact(n-p));
}
//bash-3.2 ./ex6
//n? 4
//p? 3
//Vrai
//bash-3.2
```

6 Fonctions et adresses

L'opérateur adresse `&` Nous appelons *objet* une zone de mémoire, associée à un type, qui est accessible pendant l'exécution d'un programme. Si `o` est un objet, alors `&o` est l'adresse de cet objet dans la mémoire. Je noterai par la suite les adresses sous la forme $axxx$ où le préfixe a nous rappelle qu'il s'agit dune adresse, et xxx est un nombre entier. Les seuls objets que nous avons vu pour l'instant sont les *variables*, et on accède en général à une variable par son *identificateur*, par exemple `v`. Ainsi, lors de l'exécution, `v` a une certaine adresse, par exemple $a131$. Dans ce cas `&v` a comme valeur $a131$ [1].

Passage de l'adresse d'un tableau Ce que nous avons vu à la section précédente est le mode de *passage par valeur* des arguments à l'appel d'une

1. En réalité une adresse est simplement un entier.

fonction : on évalue un argument lors de l'appel, la *valeur* obtenue est passée à une variable représentant cet argument pendant l'appel de la fonction. Cependant, en programmation il est souvent utile de passer une *référence* à un objet, de manière à éviter de le recopier (ce qui peut-être coûteux) et surtout de manière à pouvoir modifier cet objet.

Un cas particulier en *C* est le passage d'un argument de type *tableau*. En *C* le nom d'un tableau est équivalent à l'adresse de son premier élément, autrement dit, si `t` est un tableau, on a :

`t` ≡ `&t[0]`

Lors de l'appel d'une fonction dont un argument est un tableau, c'est en réalité l'adresse de son premier élément qui est passé. Ceci est très clair dans le cas d'une fonction `lireTab`, dont l'appel `lireTab(t,n)` a pour effet de lire les valeurs du tableau `t`. Ci-dessous, un programme qui :

- lit la taille `taille` d'un tableau `tab`
- lit le tableau `tab`.
- affiche le tableau `tab`.

Dans le programme, on définit :

- une fonction `lireTab`, telle que après `lireTab(t,n)` le tableau `t` de taille `n` est rempli.
- une fonction `AfficheTab`, telle que l'éxécution de `AfficheTab(t,n)` affiche les `n` éléments de `t`.

```
#include <stdio.h>
#include <stdlib.h>
#define N 5
/* Prototypes */
void lireTab(int t [],  int n);
void afficheTab(int t [],  int n);
/* Fonction principale*/
int main(){
        int  taille ;
        int tab[N];
        printf(" taille ?  ");
        scanf("%d",&taille);

        // lecture
        lireTab(tab,  taille );
        afficheTab(tab, taille );
        return EXIT_SUCCESS;
}
/* Fonctions*/
void lireTab(int t [],  int n){
        /* lecture  des n éléments du tableau t*/
        int i;
        for(i=0;i<n;i=i+1){
                printf("t[%d]?  ",i);
                scanf("%d",&t[i]);
        }
```

```
        return;
}

void afficheTab(int t [], int n){
        /* Affichage des n éléments du tableau t*/
        int i;
        for(i=0;i<n;i=i+1){
                printf("%d ",t[i]);
        }
        printf("\n");
        return;
}

//bash-3.2 gcc ex8.c -o ex8
//bash-3.2 ./ex8
// taille ?  4
//t[0]?  1
//t[1]?  4
//t[2]?  5
//t[3]?  3
//1 4 5 3
```

Quelques remarques sur cet exemple et son exécution en commentaires à la fin du programme :

- Dans l'en-tête des fonctions `lireTab` et `afficheTab`, le premier argument est le tableau `t`, l'écriture `int t[]` indique précisément que `t` est un tableau d'entiers, sans préciser le nombre d'éléments du tableau. Ceci est possible parce que ce qui est transmis au moment de l'appel est l'adresse du premier élément du tableau (le nombre d'éléments n'intervient pas).
- Dans l'exemple d'exécution, lors de l'appel `lireTab(tab, taille);`
  - `taille` prend la valeur 4 qui est transmise à la variable `n` accessible à l'appel de la fonction.
  - `tab` est un tableau et a la valeur de l'adresse de son premier élément, par exemple $a120$, et cette adresse devient celle du tableau `t` de la fonction. On a alors `t` $\equiv$ `tab`, c'est-à-dire qu'ils désignent le même objet dont l'adresse de début est $a120$.
  - Lors de l'exécution du corps de `lireTab`, à la première itération où `i`=0, la valeur 1 lue par `scanf("%d",&t[i]);` est rangée dans le premier élément du tableau `t`, donc dans le premier élément du tableau `tab` de la fonction principale, qui est à l'adresse $a120$.
- A la fin de l'appel `lireTab(tab, taille);`, l'exécution continue dans la fonction principale, et `tab` contient les valeurs 1 4 5 3.
- Lors de l'appel `afficheTab(tab, taille);` `taille` a la valeur 4 transmise à `n` et l'adresse $a120$ est associée au premier élément de `t`. Ce sont donc les éléments du tableau `tab` de la fonction principale qui sont affichés.

L'expression `*p` désigne l'objet qui est à l'adresse contenue dans le *pointeur* `p` Une adresse, comme $a120$ peut être rangée dans une variable

de type *pointeur*. Il faut alors spécifier le type d'objet que l'on trouvera à cette adresse. Par exemple une variable est un *pointeur sur un int* si elle peut contenir l'adresse d'un objet de type `int`.

Pour déclarer un pointeur p dont le contenu est une adresse d'un objet de type `int`, on écrit :

```
int *p; /* l'objet d'adresse p est un entier */
```

Échange de deux variables Ci-dessous un programme dans lequel les valeurs de deux variables sont échangées à l'aide d'une fonction `echange` :

```
# include <stdio.h>
# include <stdlib.h>
/* Prototypes */
void echange (int *pa, int *pb);
/*  Fonction principale */
int main (){
        int a,b;
        printf("a et b?");
        scanf("%d",&a);
        scanf("%d",&b);
        echange(&a,&b);
        printf("a =%d, b=%d\n", a,b);
        return EXIT_SUCCESS;
}
/* Fonctions */
void echange (int *pa, int *pb){
        int aux;
        aux=(*pa);
        (*pa)=(*pb);
        (*pb)=aux;
        return;
}
//bash-3.2 gcc ex11.c -o ex11
//bash-3.2 ./ex11
//a et b?2 3
//a =3, b=2
//bash-3.2
```

Le but étant d'échanger les valeurs des variables `a` et `b` de la fonction principale, on transmet les adresses de ces variables à la fonction `echange`. En effet si on ne transmettait que leurs valeurs, les objets `a` et `b` ne seraient pas modifiés à la fin de l'exécution. Nous allons supposer ci-dessous que la variable `a` est à l'adresse $a9$, la variable `b` a l'adresse $a21$ et que `a`=2 et `b`=3.

- l'appel `echange(&a,&b);` transmet les adresses des variables de type `int` `a` et `b`. Ici les adresses $a9$ et $a21$ sont transmises.
- Dans l'en-tête de la fonction les arguments `pa` et `pb` sont de type *pointeur sur* `int`, ce qui s'écrit `int *pa, int *pb`. Lors de l'appel on a donc `pa`=$a9$ et `pb`=$a21$.
- Dans le corps de la fonction, l'échange est effectué de la manière suivante :

  - On range la valeur de l'objet d'adresse `pa` dans la variable `aux` par `aux=(*pa);` : `aux` a alors la valeur 2 prise dans l'objet d'adresse `pa`=$a9$.
  - On range la valeur de l'objet d'adresse `pb` dans l'objet d'adresse `pa` par `(*pa)=(*pb);` : `(*pa)` a alors la valeur 3 prise dans l'objet d'adresse `pb`.
  - On range la valeur de la variable `aux` dans l'objet d'adresse `pb` par `(*pb)=aux;` : `(*pb)` a alors la valeur 2 prise dans `aux`.
  - On a donc maintenant la valeur 2 à l'adresse `pb`=$a21$ et la valeur 3 à l'adresse `pa`=$a9$
- Au retour dans la fonction principale, `a`, qui est à l'adresse $a9$, contient 3, et `b` qui est à l'adresse $a21$ contient 2 : les variables ont échangé leurs valeurs.

Pourquoi écrit-on `scanf(...,&a)` mais `printf(..., a)` ? Il faut passer à `scanf` l'adresse de `a`, c'est-à-dire `&a`. La valeur sera alors rangée dans l'objet d'adresse `&a` pendant l'exécution de `scanf`, et au retour `a` aura la valeur lue. En revanche `printf` ne doit pas modifier la variable `a`, on peut donc simplement passer la valeur de `a` à la fonction `printf`.

Plus généralement, si un objet doit être modifié par l'appel d'une fonction, on doit passer son adresse, sinon on peut passer sa valeur, ce qui donne une écriture plus simple des fonctions, et diminue le risque d'erreur.

Exemple : Une fonction qui inverse l'ordre des éléments d'un tableau
On veut écrire un programme inversant l'ordre des éléments d'un tableau, sans utiliser de tableau supplémentaire. On utilisera `lireTab` et `afficheTab` pour la lecture et l'affichage du tableau, et on écrira une fonction `inverseTab`, telle que `inverseTab(t, ideb,ifin)` inverse l'ordre des éléments entre les indices `ideb` et `ifin` du tableau `t`. Cette fonction utilisera la fonction `echange` permettant l'échange des valeurs de deux objets de type `int` :

```
/*  inverseTab.c */
#include <stdio.h>
#include <stdlib.h>
#define N 5
/* Prototypes */
void inverseTab(int t [],  int ideb,  int  ifin );
void echange (int *pa, int *pb);
void lireTab(int t [],  int n);
void afficheTab(int t [],  int n);


/* Fonction principale*/
int main(){
        /* lit  les éléments d'un tableau, range les éléments en ordre inverse
                sans  utiliser  de tableau intermédiaire                    */
        int  taille ;
        int tab[N];
```

```
        // lecture
        printf(" taille ? (<= %d) ",N);
        scanf("%d",&taille);
        printf("élements? ",N);
        lireTab(tab, taille );
        // inversion entre les éléments d'indice 0 et taille -1
        inverseTab(tab, 0, taille -1);
        //affichage
        afficheTab(tab, taille );
        return EXIT_SUCCESS;
}
/* Fonctions*/
void inverseTab(int t [], int ideb, int ifin ){
        /* inversion de l'ordre des éléments du tableau t
          entre l'indice ideb et l'indice ifin */
        int i,me; // me est le milieu par défaut de l'intervalle entier [ideb, ifin ]
        me=(ifin+ideb) /2;
        // on échange les éléments en (ideb, ifin ), (ideb+1/ifin-1) ....
        for(i=ideb;i<=me;i=i+1){
                echange(&t[i],&t[ideb + ifin -i]);
        }
        return;
}
void echange (int *pa, int *pb){
        /* echange des valeurs des objets (de type int) aux adresses pa et pb */
  .......
}

void lireTab(int t [], int n){
        /* lecture des n éléments du tableau t*/
  .......
}

void afficheTab(int t [], int n){
        /* Affichage des n éléments du tableau t*/
  .......
}
//dyn246:- gcc inverseTab.c -o inverseTab
//dyn246:- ./inverseTab
// taille ? (<= 5) 4
//élements? 1 3 5 9
//9 5 3 1
//dyn246:-
```

Quelques remarques sur ce programme :

- La fonction `inverseTab` est appelée ici sur toute la partie utile du tableau, entre les indices 0 et `taille`-1
- Dans la fonction `inverseTab` on progresse dans le tableau à partir de `ideb` jusqu'au *milieu par défaut* `me` du segment [`ideb,ifin`].
- Le milieu par défaut est en (`ideb+ifin)/2`, alors que le milieu par excès

est en `(ideb+ifin+1)/2`.
- Lors de l'appel de `echange`, ce sont les adresses des éléments à échanger qui sont passées en argument.

En pratique, lorsque dans un corps de boucle on a deux indices dépendant l'un de l'autre, ici ceux des éléments à échanger, alors on calcule le deuxième indice en fonction du premier. Ici on sait qu'on veut échanger :
- $i = ideb$ et $f(i) = f(ideb) = ifin$
- $i = ideb + 1$ et $f(i) = f(ideb + 1) = ifin - 1$
- ...

On en déduit la fonction f définie par $f(i) = ideb + ifin - i$:
- $f(ideb) = ifin = ideb + ifin - ideb$
- $f(ideb + 1) = ifin - 1 = ideb + ifin - (ideb + 1)$
- ...
- $f(i) = ideb + ifin - i$
- ...

7 Représentations de matrices par des tableaux à deux dimensions

Nous donnons ci-dessous un programme qui lit une matrice, calcule sa transposée, effectue le produit de la matrice par sa transposée et affiche le résultat. Le programme permet alors à l'utilisateur de faire un calcul sans mémorisation (on ne peut réutiliser le résultat courant que comme premier argument de l'opérateur suivant). Les opérateurs sont l'addition, la multiplication, la transposition.

Pour cela nous allons représenter une matrice par un tableau, le nombre de lignes et le nombre de colonnes de la matrice. Lorsqu'un traitement doit être fait sur une matrice, ces trois informations sont nécessaires.

```
/*
 *  calcul matriciel
 */
#include <stdio.h>
#include <stdlib.h>
#define NL 100
#define NC 100

// primitives de calcul matriciel avec représentations par tableaux et
/*  types  */

/* prototypes */

void lireMatrice(int m[][NC],int nl,int nc);
void afficheMatrice(int m[][NC],int nl,int nc);
void produitMatrices(int m1[][NC],int m2[][NC],int nl1,int n,int nc2,int mr[][NC]);
void transpose(int m[][NC],int nl,int nc,int mr[][NC]);
void copieMatrice(int m[][NC],int nl,int nc,int mr[][NC]);
```

```
void calculette ();

/* fonction principale */

int main () {
    //    primitives de calcul matriciel avec tableaux (sans struct)
    int nl1,nc1;
    int nl2,nc2;
    int nlr, ncr;
    int m1[NL][NC],m2[NL][NC],mr[NL][NC];

    //définit les dimensions de m1, lit m1 et affiche m1
    printf("nb lignes, nb de colonnes et elements ?\n");
    scanf("%d",&nl1);
    scanf("%d",&nc1);
    lireMatrice(m1,nl1, nc1);
    afficheMatrice(m1,nl1, nc1);
    // transposition de m1 dans mr
    // definitions des dimensions de mr et affichage
    transpose(m1,nl1,nc1,mr);
    nlr=nc1;
    ncr=nl1;
    afficheMatrice(mr,nlr, ncr);
    // produit de m1 et de sa transposée mr dans m2
    // definitions des dimensions de m2 et affichage
    produitMatrices(m1,mr,nl1,nc1,ncr,m2);
    nl2=nl1;
    nc2=ncr;
    afficheMatrice(m2,nl2, nc2);
    // lancement de la calculette
    calculette ();
    return 0;
}

/* fonctions */

void lireMatrice(int m[][NC],int nl,int nc){
    /* lit une matrice m de dimensions nl nc */
    int i,j;
    for (i=0;i<nl;i++){
        for (j=0;j<nc;j++){
            scanf("%d",&m[i][j]);
        }
    }
    return;
}

void afficheMatrice(int m[][NC],int nl,int nc){
    /* affiche la matrice m de dimensions nl nc*/
    int i,j;
```

```
    for (i=0;i<nl;i++){
        for (j=0;j<nc;j++){
            printf("%d ",m[i][j]);
        }
        printf("\n");
    }
    return;

}
void produitMatrices(int m1[][NC],int m2[][NC],int nl1,int n,int nc2,int mr[][NC]){
    /* calcule le produit de deux matrices m1 et m2                    */
    /* de dimensions nl1 n et n nc2                                    */
    /* range le résultat dans mr de dimensions nl1 nc2          */
    int i,j,k;
    for (i=0;i<nl1;i++){
        for (j=0;j<nc2;j++){
            mr[i][j]=0;
            for(k=0;k<n;k=k+1){
                mr[i][j]= mr[i][j]+ m1[i][k]*m2[k][j];
            }
        }
    }
    return;


}

void transpose(int m[][NC],int nl,int nc,int mr[][NC]){
    /*  calcule la transposee d'une matrice m                       */
    /* de dimensions nl nc                                          */
    /* range le résultat dans m de dimensions nc nl                 */
    int i,j;
    for (i=0;i<nl;i++){
        for (j=0;j<nc;j++){
            mr[j][i]=m[i][j];
        }
    }
    return;
}

void sommeMatrices(int m1[][NC],int m2[][NC],int nl,int nc,int mr[][NC]){
    /* calcule le produit de deux matrices m1 et m2                    */
    /* de dimensions nl1 n et n nc2                                    */
    /* range le résultat dans mr de dimensions nl1 nc2          */
    int i,j;
    for (i=0;i<nl;i++){
        for (j=0;j<nc;j++){
            mr[i][j]= m1[i][j]+m2[i][j];
        }
    }
```

```
    return;
}

void copieMatrice(int m[][NC],int nl,int nc,int mr[][NC]){
    /* recopie m de nl lignes et nc colonnes dans mr */
    int i,j;
    for (i=0;i<nl;i++){
        for (j=0;j<nc;j++){
            mr[i][j]= m[i][j];
        }
    }
    return;
}

void calculette(){
/* lit la première matrice m1 puis un opérateur parmi '+' addition, '*' multiplication et
   't' transposition ou le caractère de fin '#', puis si l'opérateur est binaire
   une seconde matrice m2, effectue le calcul, range le résultat dans m1 et
   demande de nouveau un operateur.
  On calcule ainsi une expression repre'sente'e par un peigne gauche
 */
    int m1[NC][NC];int nl1,nc1;
    int mr[NC][NC];int nlr, ncr;
    int m2[NC][NC];int nl2,nc2;
    char ch[2]; // on lira l'operateur dans une chaine de longueur 1
    char op;

    //
    printf("nb lignes, nb de colonnes et elements ?\n");
    scanf("%d",&nl1);
    scanf("%d",&nc1);
    lireMatrice(m1,nl1, nc1);
    printf("operateur ? + * t ou # pour finir\n");
    scanf("%s",ch);op=ch[0];
    while(op != '#'){
        if (op == '+'){
            nl2=nl1;
            nc2=nc1;
            printf("matrice %d lignes %d colonnes ?\n",nl1,nc1);
            lireMatrice(m2,nl2, nc2);
            sommeMatrices(m1,m2,nl1,nc1,mr);
            nlr=nl1;
            ncr=nc1;
        }
        else if (op == '*'){
            printf("nb de colonnes de la matrice ? \n");
            scanf("%d",&nc2);
            nl2=nc1;
            printf("matrice %d lignes %d colonnes ?\n",nl2,nc2);
            lireMatrice(m2,nl2, nc2);
```

```
            produitMatrices(m1,m2,nl1,nc1,nc2,mr);
            nlr=nl1;
            ncr=nc2;
        }
        else if (op =='t'){
            transpose(m1, nl1, nc1, mr);
            nlr=nc1;
            ncr=nl1;
        }
        copieMatrice(mr, nlr, ncr, m1);
        nl1=nlr;
        nc1=ncr;
        printf("matrice %d lignes %d colonnes \n",nl1,nc1);
        afficheMatrice(m1, nl1, nc1);
        printf("operateur ? + * t ou # pour finir\n");
        scanf("%s",ch);op=ch[0];
    }
//     afficheMatrice(m1, nl1, nc1);
    return;
}


/*
 nb lignes, nb de colonnes  et elements ?
 2 3
 2 3 1
 3 5 7
 2 3 1
 3 5 7
 2 3
 3 5
 1 7
 14 28
 28 83
 nb lignes, nb de colonnes  et elements ?
 2 3
 2 3 1
 3 5 7
 operateur ? + * t ou # pour finir
 t
 matrice 3 lignes  2 colonnes
 2 3
 3 5
 1 7
 operateur ? + * t ou # pour finir
 +
 matrice 3 lignes  2 colonnes ?
 1 1
 1 1
 1 1
```

```
 matrice 3 lignes  2 colonnes
 3 4
 4 6
 2 8
 operateur ? + * t ou # pour finir
 #
*/
```

On notera qu'il est nécessaire de donner la deuxième dimension `NC` du tableau dans l'en-tête, (par exemple `m1` dans `produitMatrices`) car le compilateur ne peut pas deviner comment il doit interpréter `m[1][0]` s'il ne sait pas découper le bloc de mémoire associé à `m1` en lignes de `NC` éléments.

Une remarque sur ce programme est qu'on est obligé de maintenir en permanence le lien entre le tableau (par exemple `m1`) et ses dimensions (ici `nl1` et `nc1`) et de transmettre ces informations aux fonctions.

Nous allons voir, un peu plus loin une autre implémentation de ce programme de calcul matriciel dans laquelle on enfermera le tableau et ses deux dimensions dans un seul objet dont le type sera un enregistrement. Nous proposerons d'ailleurs plusieurs variantes de cette implémentation à base d'enregistrement.