Université Paris 13
Institut Galilée
Licence 1ère année
2006-2007

Programmation Impérative
Polycopié de cours n° 2

C. Recanati

L.I.P.N.

http : //www-lipn.univ-paris13.fr/~recanati

Table des matières

5 FONCTIONS 4

5.1 EXEMPLE 5
5.1.1 Déclaration et définition de fonctions 5
5.1.2 Appel de fonction et passage des paramètres par valeur 8
5.2 PASSAGE DE PARAMETRES 11
5.2.1 Passage par valeur 11
5.2.2 Passage de pointeurs : altération possible des valeurs 12
5.2.3 Passage par référence : le cas des tableaux 12
5.3 VARIABLES LOCALES ET VARIABLES GLOBALES 14

6 TABLEAUX 16

6.1 TABLEAUX A 1 DIMENSION 16
6.1.1 Déclaration 16
6.1.2 Initialisation 17
6.1.3 Les tableaux de caractères 18
6.1.4 Exemples de fonctions manipulant des tableaux de caractères 22
6.2 TABLEAUX MULTI-DIMENSIONNELS 24
6.2.1 Tableaux à deux dimensions 24
6.2.2 Tableaux à plus de deux dimensions 28
6.3 FONCTIONS ET TABLEAUX 28

7 STRUCTURES ET UNIONS 31

7.1 STRUCTURES 31
7.2 OPERATIONS SUR LES STRUCTURES 32
7.3 FONCTIONS A PARAMETRE DE TYPE STRUCTURE 33
7.4 UNIONS 34

Les types que nous avons utilisés jusqu'à présent sont des types de base simples. Or, il arrive souvent en programmation que l'on ait à traiter des données nombreuses, mais formant des groupes relativement homogènes. Mémoriser ces données dans des variables ayant des noms différents, rend le traitement difficile, voire impossible. Par exemple, s'il s'agit de lire une suite de 1000 réels pour les afficher en ordre inverse, déclarer 1000 variables de type `float` dans le programme n'est guère faisable, et nous avons vu qu'il existait dans ce cas la possibilité d'utiliser le type tableau. Dans d'autres cas, on ne connaîtra pas par avance le nombre de données à lire, et l'on ne pourra pas se contenter de tableaux. Pour résoudre ce type de question, les langages impératifs offrent diverses possibilités de définition de **types structurés** de données. Un type structuré est un assemblage de types simples, définis par le programmeur ou prédéfinis par le langage.

Dans les sections qui suivent, nous présentons les types structurés du langage C : fonctions, tableaux, structures, unions et pointeurs. Ces types ont en commun de permettre l'accès à des ensembles de plusieurs valeurs.

Sous cet aspect, une fonction est en effet une variable permettant de lancer le calcul de valeurs paramétrées, au moyen d'une expression particulière : l'appel de fonction. Les variables de types tableaux, structures ou unions, offrent de leur côté différents moyens de regrouper plusieurs valeurs au sein d'une même entité. Les tableaux sont constitués de valeurs d'un même type rangées successivement en mémoire (dans des emplacements de même taille). Les structures et les unions, sont constituées de valeurs également contiguës, mais de types et de tailles possiblement différentes.

Les variables de types pointeurs enfin, permettent d'accéder à des valeurs d'un type donné quelconque à partir d'une adresse qui sera leur valeur initiale. Les pointeurs (sur des int, des float, etc.) permettent ainsi de parcourir les adresses des différentes cellules d'un tableau d'éléments d'un même type qui sera généralement un type de base, mais on manipule aussi des pointeurs sur des types structurés, comme des pointeurs sur des tableaux, sur des structures ou même des pointeurs sur des fonctions.

5 Fonctions

Une fonction est un moyen d'effectuer un calcul à partir d'arguments. La notion mathématique ne s'intéresse qu'au rapport entre les valeurs arguments et la valeur résultat (supposée unique) du calcul, mais les langages de programmation ont étendu la notion, car un calcul dans un programme est aussi susceptible d'avoir d'autres effets (appelés **effets de bords**). On pense toujours, concernant les effets de bords, à la modification des valeurs de certaines variables (on écrit en quelque sorte dans des variables), mais il s'agit a priori d'opérations d'écriture sur des supports variés (écran, imprimantes, fichiers...).

Dans certains langages de programmation, on a cherché à sauver la notion mathématique en introduisant la distinction entre une fonction - qui rapporte un résultat, et une **procédure**, qui ne retourne aucune valeur, et qui a, justement, des effets de bords. Ainsi, les procédures avaient par définition des effets, et les fonctions étaient censées se conformer au modèle mathématique, c'est-à-dire, dans le contexte de la programmation, ne faire rien d'autre que de rapporter le résultat d'un calcul[1]. La volonté sous-jacente des concepteurs de ces langages était de se libérer de la notion de variable en tant qu'emplacement mémoire, pour ne garder que la partie abstraite de cette notion : une valeur typée.

Mais la distinction procédure/fonction proposée alors était illusoire, car le problème soulevé ne provenait pas de la présence ou de l'absence d'une valeur de retour, et il restait en réalité possible, malgré cette distinction superficielle, de définir dans ces langages des « fonctions » ayant des effets de bords[2]. Le langage C n'effectue pas cette distinction, et ses « procédures » (les fonctions dont la valeur de retour est de type `void`) ne sont pas spécialement distinguées des autres fonctions[3].

Conceptuellement, une fonction C n'a donc simplement rien à voir avec une fonction mathématique. Elle fournit juste un moyen pratique de définition pour l'encapsulation d'un module de calcul. Quand des fonctions sont proprement conçues, on peut ignorer comment elles effectuent leur tâche et se contenter de savoir ce qu'elles font (qu'il y ait effets de bords ou non) – ce qui permet, bien entendu, de coller au modèle mathématique dans les cas qui s'y prêtent.

---

[1] Mais il est extrêmement difficile, voire impossible, de se passer complètement d'effets de bords – ceux-ci étant en effet les opérations d'écriture permettant de communiquer des résultats (intermédiaires ou terminaux) vers l'extérieur. Un programme qui n'effectuerait aucune écriture d'aucune sorte (aucune sortie) ne ferait tout simplement rien !

[2] Et aussi, inversement, de rapporter le résultat d'un calcul via une procédure, en plaçant le résultat de ce calcul dans une variable. Ce procédé d'altération volontaire des valeurs de variables argument présente d'ailleurs l'avantage (sur la notion de fonction a valeur de retour unique), de permettre l'implantation de fonctions mathématiques rapportant plusieurs valeurs.

[3] A l'inverse, on peut parfois en C ignorer la valeur de retour d'une fonction.

5.1 Exemple

Voici l'exemple classique de la fonction `puissance`[4]. La fonction principale de ce programme teste simplement cette fonction sur quelques valeurs :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* declaration de la fonction puissance */
5  /* i.e. des types de depart et d'arrivee */
6  /* puissance :  Int x Int -------> Int */
7  int puissance(int  m, int n) ;
8
9  int main ()
10 {
11       int i ;
12
13       printf("%d\n", puissance (2,i) ) ;
14       return EXIT_SUCCESS ;
15 }
16
17 /* definition de la fonction puissance */
18 /* puissance(x,n)=  x a la puissance n avec n >=0 */
19 int puissance (int x, int n)
20 {
21       int i, p ;
22
23       p = 1 ;
24       for (i = 1 ; i <= n; ++i)
25            p = p * x;
26       return p ;
27 }
```

5.1.1 Déclaration et définition de fonctions

La ligne 7 est une déclaration de la fonction `puissance`, qui déclare à la fois le type de la valeur retournée par cette fonction (c'est le premier `int`), ainsi que le nombre et le type des paramètres pris par cette fonction (c'est la liste qui figure entre parenthèses après le nom de la fonction, ici : `int m, int n`).

Cette déclaration n'est pas toujours nécessaire, car elle figure aussi dans la définition de la fonction (ligne 19 et suivantes). Mais dans le cas où, comme ici, la définition de la fonction viendrait après son utilisation dans le programme, il est nécessaire d'en faire la déclaration par avance pour faciliter le travail du compilateur. En effet, la fonction `main`, qui est définie la première, utilise la fonction `puissance` dans son appel à `printf` lignes 13.

Les définitions de fonctions peuvent apparaître dans n'importe quel ordre, et même figurer dans des fichiers sources différents, mais le compilateur demande que toute fonction soit

---

[4] La librairie standard contient une fonction `pow(x,y)` qui calcule x^y, avec un algorithme de calcul plus efficace.

déclarée (ou définie) avant d'être utilisée par une autre[5]. On pouvait donc ici, définir la fonction `puissance` avant de définir la fonction `main`, et dans ce cas, sa déclaration ligne 7 aurait été inutile.

Une **déclaration de fonction** a la forme :

type-retour identificateur **(** *type1 param1, type2 param2, ... , typeN paramN* **) ;**

La liste de paramètres peut être vide, mais s'il y en a plusieurs, leurs déclarations seront séparées par des virgules. Les noms utilisés pour les paramètres dans une déclaration de fonction sont sans importance (seuls les types importent). En fait, les noms de paramètres sont optionnels dans une déclaration, mais, bien choisis, ils seront une source d'information pour le lecteur. Ainsi, la fonction puissance aurait pu être déclarée ligne 7 avec :

```
int puissance (int, int) ;
```

Une **définition de fonction** commence de la même manière, mais cette fois, les noms des paramètres sont obligatoires. De plus, une définition de fonction comporte le bloc d'instructions définissant le calcul effectué par la fonction. Une définition de fonction a donc la forme :

type-retour identificateur **(** *type1 param1, type2 param2, ... , typeN paramN* **)**

{

declarations

instructions

}

La première ligne déclare, en sus du nom de la fonction et du type de sa valeur de retour, le type *et le nom* de chacun des paramètres. Les noms utilisés sont cette fois nécessaires, car ils introduisent des variables qui seront connues dans le bloc qui suit, ou **corps de définition** de la fonction. Dans ce bloc, ces variables auront pour valeurs initiales celles avec lesquelles on aura sollicité un calcul particulier de la fonction. On appelle ces variables les **paramètres formels** de la fonction, car ils permettent de donner une forme à la définition de la fonction (on parle aussi de « définition formelle »).

Il existe plusieurs qualificatifs du mot paramètre qui en modifient le sens : les paramètres formels servent à définir le corps de la fonction, et les **paramètres effectifs**, (appelés parfois aussi paramètres d'appels), désignent les expressions particulières pour lesquelles le calcul de la fonction est (effectivement) demandé dans un contexte de calcul particulier.

Dans l'exemple de la fonction puissance, les paramètres formels déclarés dans la définition de la fonction (ligne 17) s'appellent `x` et `n`.

```
int puissance (int x, int n)
{
      /* bloc de définition de la fonction
       * où x et n sont de nouvelles variables
       *  (dont la déclaration est effectuée
```

[5] Sinon, il considèrera que la valeur retournée par la fonction est un entier de type `int`.

```
         *    de cette maniere)
         */
}
```

A l'intérieur du corps de définition de la fonction, ces noms cachent (ou « masquent ») ceux des variables de mêmes noms qui pouvaient figurer dans le bloc d'où provient « l'appel » de fonction (ici le bloc de la fonction main). Par exemple, avec le programme :

```
int main ()
{
      int x, y ;

      x = 2 ;
      y = puissance (x,x) ;
      printf("x = %d , y = %d\n", x, y) ;
}
```

on obtiendra lors d'une exécution l'impression de la ligne :

```
x = 2 , y = 4
```

car la variable `y` recevra bien la valeur de `puissance(2,2)`, c'est-à-dire, $2^2 = 4$.

Le fait que les paramètres d'appels de la fonction `puissance` soient `x` et `x` ne perturbera pas le calcul effectué dans le bloc de définition de la fonction, et ne perturbera pas non plus la valeur de la variable `x` de la fonction main, même si le calcul de la fonction modifie le paramètre formel `x`. De nouvelles variables `x` et `n` seront utilisées comme paramètres formels (on dit qu'il s'agit de variables « *fraîches* », i.e. dont les adresses sont nouvelles, c'est-à-dire différentes de celles de toutes les autres variables). L'exécution du corps de la fonction se fera alors dans ce nouveau contexte, où le paramètre formel `x` de la fonction `puissance` aura été initialisé avec la valeur `2` de la variable `x` figurant dans le `main`, de même que le paramètre formel `n` de la fonction aura été initialisé avec cette même valeur.

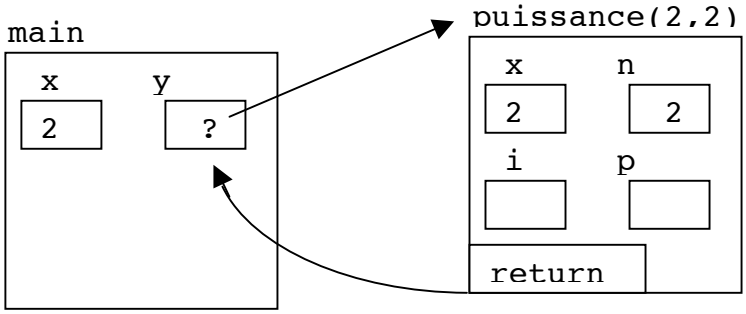


Figure 1. Les paramètres formels sont de nouvelles variables

Chaque fonction dispose ainsi de son propre espace de travail (ou **environnement)** dans lequel sont définies les variables qui lui sont propres (= privées). Ces variables sont qualifiées de locales à la fonction, précisément parce qu'elles ne sont définies que dans le bloc de définition de la fonction. Nous avions déjà introduit cette notion lorsque nous avons introduit les instructions composées (i.e. les blocs d'instructions). Ici, la fonction `main` dispose de deux variables locales à son bloc de définition : `x` et `y` ; et la fonction `puissance` des deux variables locales à son bloc de définition : `i` et `p` ; mais la fonction `puissance` dispose en réalité aussi de deux variables locales supplémentaires : ses paramètres formels `x` et `n`.

5.1.2 Appel de fonction et passage des paramètres par valeur

On nomme **appel de fonction**, une expression de la forme :

identificateur `(` *expression1* `,` ... `,` *expression2* `)`

dans laquelle *identificateur* est un nom de fonction. Les expressions figurant entre parenthèses sont les paramètres effectifs de l'appel de la fonction. Ils doivent être en nombre égal au nombre de paramètres formels de la définition de la fonction[6], et cette liste peut donc être vide si la fonction a été définie avec une liste de paramètres formels vide.

Rappelons qu'un appel de fonction peut constituer une instruction s'il est terminé par un point-virgule. Mais un appel de fonction est une expression qui peut figurer dans une expression quelconque. On le rencontrera souvent comme membre droit d'une affectation, mais aussi bien comme expression d'un paramètre effectif figurant dans l'appel à une autre fonction, comme on l'a vu dans l'exemple précédent ligne 11. (Les appels à la fonction `puissance` sont inclus ici dans un appel à la fonction `printf`).

Mais quoi qu'il en soit, un appel de fonction figure dans une instruction, et cette instruction (englobante) se situe à l'intérieur d'un bloc, qu'on appellera le **bloc d'appel**, **bloc englobant** ou **bloc appelant**.

Lorsqu'une instruction contient un appel de fonction, son exécution nécessite la connaissance de la valeur calculée par cette fonction (pour les valeurs des paramètres effectifs de l'appel). Le bloc appelant interrompt donc l'exécution de l'instruction englobante, et se met en attente de la valeur retournée par la fonction. Le programme lance l'exécution du calcul de la fonction pour les valeurs indiquées, et ce calcul se termine par une instruction de retour rapportant cette valeur. Le bloc appelant peut alors reprendre l'exécution là où il l'avait interrompue, en utilisant la valeur retournée par la fonction pour la poursuite de son calcul.

Dans notre exemple, le corps de la fonction `main` **appelle** la fonction `puissance` lors de l'exécution d'une instruction `printf` figurant dans la boucle `for`. La fonction `main` doit calculer la fonction `puissance` une première fois, pour le calcul de `puissance (2, i)` avec `i` valant zéro. On dit que la fonction **est appelée** avec les paramètres effectifs `2` et `i`. On nomme aussi souvent les valeurs calculées des paramètres effectifs les **arguments** de la fonction, et dans ce cas, les arguments de la fonction sont les **valeurs** `2` et `0`.

Lorsqu'un appel de fonction est évalué, les expressions correspondant aux paramètres effectifs de cet appel sont évaluées, et leurs valeurs permettent d'initialiser les paramètres formels correspondants. C'est ce qu'on appelle le **passage par valeur** des paramètres. C'est la *valeur* des paramètres effectifs qui est attribuée initialement aux paramètres formels. Le lien entre les deux types de paramètres est donc uniquement constitué par le fait que leurs valeurs initiales (avant calcul de la fonction) sont les mêmes. Mais ce qui est important, c'est que les valeurs des variables figurant dans l'appel de fonction restent inchangées, avant et après le calcul de la fonction. La vie des paramètres formels, de son côté, est éphémère : ces variables n'existent que le temps du calcul de la fonction, et leurs valeurs à la fin du calcul est sans incidence pour le bloc appelant. L'exemple qui suit illustre cette propriété :

```
void par_valeur (int i) {
      printf("i = %d au debut de par_valeur\n",i) ;
```

[6] A l'exception des fonctions variadiques.

```
        i = 999999 ;
        printf("i = %d dans par_valeur\n",i) ;
}

int main () {
        int i ;

        i = 1 ;
        printf("i = %d avant\n",i) ;
        par_valeur(i) ;
        printf("i = %d apres\n",i) ;
        return 0 ;
}
```

Exécution :

```
%  a.out
i = 1 avant
i = 1 au debut de par_valeur
i = 999999 dans par_valeur
i = 1 apres
```

L'entier `i` déclaré dans la fonction `main`, et initialisé à `1` n'a aucun rapport avec le paramètre formel `i` de la fonction `par_valeur`, si ce n'est que sa valeur, ici, `1`, aura été transmise à la fonction pour initialiser son paramètre formel. Mais ensuite, si le paramètre formel de la fonction est modifié, cela reste sans incidence pour la variable `i` du programme principal.

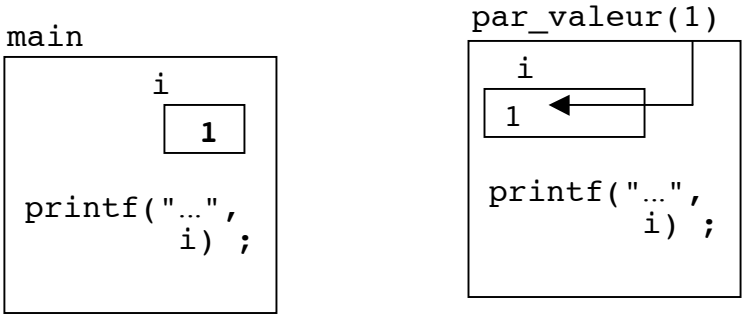


Figure 2. Le passage de paramètre par valeur

Le langage C n'utilise que ce mode de transmission[7] (à l'exception du cas des tableaux, que nous verrons plus loin). Les paramètres formels tiennent lieu de variables qui ne sont connues que dans le bloc de définition de la fonction. Les paramètres formels sont juste initialisés par les valeurs des expressions figurant dans l'appel, et cette initialisation impose que le type des paramètres effectifs (celui des expressions figurant dans l'appel) coïncide avec celui des paramètres formels, qui a été déclaré lors de la définition de la fonction.

Après l'initialisation des paramètres formels, le corps de la fonction est exécuté, et cette exécution doit conduire, s'il s'agit d'une fonction rapportant une valeur non vide, au calcul

[7] Il existe d'autres mode de transmission de paramètres dans d'autres langages. En particulier, le passage « par référence » des paramètres déclarés `var` en Pascal. Nous verrons plus loin que C utilise ce mode pour les variables de type tableau.

d'une valeur de retour, rapportée par une instruction `return`. Cette valeur est alors utilisée dans le bloc appelant, pour poursuivre l'exécution du calcul de l'instruction englobante.

Reprenons donc l'exemple de la fonction `puissance` avec un programme un peu plus complexe :

```
1  int main ()
2  {
3        int i ;
4
5        for (i=0 ; i <
6             printf("%d %d %d\n", i,
7                       puissance (2,i),puissance (-3,i) );
8        return EXIT_SUCCESS ;
9  }
```

Lors du premier cycle d'exécution de la boucle `for` de la fonction `main`, le programme doit exécuter l'instruction

```
 printf("%d %d %d\n", i, puissance (2,i),puissance (-3,i)
);
```

à un moment où `i` vaut `0`. Pour calculer les paramètres effectifs de la fonction `printf` le programme est amené à calculer successivement les valeurs de `puissance (2,i)` et de `puissance (-3,i)` avec `i` valant zéro.

Pour le calcul de `puissance (2,i)`, les paramètres formels `x` et `n` de `puissance` sont initialisés avec `2` et `0`, et le corps de la fonction `puissance` exécuté avec ces valeurs : la variable `p` du bloc de définition de la fonction `puissance` est initialisée à `1`, puis le `for` (de la fonction puissance) est exécuté. La condition `i <= n` est fausse avant même d'entrer dans la boucle, puisque `i` vaut `1` et que `n` vaut `0`. L'instruction de la boucle n'est donc pas exécutée, et la valeur de `p`, c'est-à-dire `1`, est retournée comme valeur (pour le troisième argument de `printf`).

De manière analogue, le calcul de `puissance (-3, i)` conduit à initialiser `x` et `n` dans le corps de définition de la fonction `puissance` avec les valeurs `-3` et `0`, et la valeur de retour est cette fois encore `1`. L'exécution de l'instruction figurant dans le `main` lignes 6-7 peut donc se poursuivre, avec la valeur `1` pour dernier argument de `printf.`

On exécute donc finalement un `printf("%d %d %d\n", 0, 1, 1)` et la première ligne imprimée par ce programme est :

```
0 1 1
```

Remarque

Si une fonction a plusieurs paramètres, ils seront évalués dans un ordre quelconque et pas nécessairement de gauche à droite. Cet ordre n'est en effet pas spécifié par le langage C, et il varie selon les compilateurs (un ordre relativement fréquent est justement de droite à gauche !). Les conséquences peuvent être importantes, si l'évaluation des paramètres effectifs a des effets de bord. Cette situation peut se produire si on utilise des affectations, ou des appels de fonctions ayant elles-mêmes des effets de bord, comme paramètres effectifs.

Si on écrit des fonctions qui ont des effets de bord (c'est-à-dire qui imprime des choses ou qui modifient des variables), on prendra garde à ne pas les faire figurer comme paramètres effectifs dans un appel de fonction.

5.2 Passage de paramètres

5.2.1 Passage par valeur

Comme nous l'avons souligné, le passage des paramètres par valeur fait qu'il ne peut pas y avoir d'altération des valeurs des variables figurant dans l'expression d'appel. Mais cela n'est vrai que si les paramètres ont un des types simples que nous avons jusqu'à présent manipulés, c'est-à-dire pour l'instant un type de base. En effet, nous avons vu que les paramètres formels des fonctions sont des variables temporaires dont les adresses sont différentes, et l'on peut donc les utiliser librement. Quoi qu'il leur advienne dans la fonction (et en particulier si leurs valeurs sont modifiées), les valeurs des variables extérieures qui figurent dans l'appel, ne seront pas, elles, altérées par l'exécution de la fonction.

Cette propriété, propre au passage par valeur, fait qu'on peut se servir des variables introduites par les paramètres formels pour économiser le nombre de variables locales déclarées dans le corps de définition d'une fonction. Ici, par exemple, la variable `i` déclarée dans le corps de la fonction `puissance`, peut être économisée, car son rôle de compteur peut être joué par le paramètre formel `n` :

```
17 int puissance (int x, int n)
18 {
19      int p ;
20
21      for (p = 1 ; n > 0; --n)
22           p = p * x;
23      return p ;
24 }
```

La modification du paramètre `n` à l'intérieur du corps de la fonction `puissance` est en effet sans incidence pour la valeur de la variable `i` figurant dans les appels de cette fonction dans le corps du `main` :

```
 printf("%d %d %d\n", i, puissance (2,i),puissance (-3,i)
);
```

En effet, lors de l'appel `puissance (2,i)`, c'est la *valeur* de l'expression `i` (réduite ici à la variable `i`) qui est calculée pour initialiser le paramètre formel `n`. *Ce n'est pas la variable* `i`, *en tant que variable qui est fournie comme paramètre formel pour la fonction.* Un paramètre effectif ne se réduit d'ailleurs pas nécessairement à une variable, comme le montre le fait qu'on aurait pu aussi bien avoir à calculer `puissance (2,i + 3)` ou `puissance (2,i*i)`.

Les paramètres effectifs sont en effet, non pas des variables, mais des *expressions* dont la valeur est transmise à la fonction pour initialiser d'autres variables que sont ses paramètres formels. Les valeurs des variables qui figurent dans les expressions des paramètres effectifs ne peuvent donc pas *a priori* être altérées par l'exécution du corps de la fonction[8].

[8] Sous réserve, qu'il s'agisse d'expressions simples ne désignant pas d'adresses, comme on l'explique dans le paragraphe suivant.

5.2.2 Passage de pointeurs : altération possible des valeurs

Il est néanmoins faux de penser que le passage par valeur empêche toute variable figurant dans l'expression d'un paramètre effectif d'être modifiée : il y a justement un cas où ce mode de transmission permet de modifier la valeur d'une variable, c'est quand la valeur qui est passée en argument est précisément celle d'une *adresse* de variable (techniquement, il s'agit alors d'un argument de type **pointeur**[9]).

Nous reviendrons plus loin sur cette question du passage de paramètres de type pointeurs, et nous verrons alors que, dans ce cas particulier, les valeurs des variables impliquées peuvent être modifiées. Certains auteurs, pour indiquer que dans ce cas c'est l'adresse même de la variable, et non pas sa valeur, qui est passée en argument, parlent alors de **passage par référence** (on transmet son adresse – c'est-à-dire sa référence, en paramètre à la fonction).

5.2.3 Passage par référence : le cas des tableaux

L'autre cas qui vient contredire l'affirmation selon laquelle les valeurs des variables ne sauraient être modifiées par l'exécution d'un appel de fonction est celui des tableaux. Mais le cas des tableaux est un peu différent de celui des pointeurs. Cette fois, c'est purement et simplement parce que **le passage des arguments de type tableau ne s'effectue pas par valeur**. Le passage par valeur d'argument de type tableau pose en effet le problème de la transmission de nombreuses valeurs, et le langage C a opté pour ne pas suivre ce schéma de transmission dans ce cas particulier. La copie d'un grand nombre de valeurs est en effet très coûteuse, non seulement en place mémoire, mais aussi en temps d'exécution.

En C, si un argument d'appel de fonction est une expression réduite à une variable de type tableau*, il n'y a pas passage par valeur*, et ce n'est pas un nouveau tableau, dont on aurait recopié toutes les valeurs, dont dispose la fonction. Au contraire, la fonction dispose directement de la variable tableau passée en argument, de sorte que toutes les modifications faites sur les cellules du paramètre formel figurant dans la définition de la fonction sont faites directement sur celles du tableau figurant en paramètre effectif dans l'appel.

Plus précisément, quand le nom d'un tableau figure en argument d'une fonction, la valeur passée à la fonction pour initialiser le paramètre formel (de type pointeur ou tableau) est alors celle de *l'adresse* de début du tableau argument (l'adresse de la première cellule). Il s'agit alors d'un **appel par référence**. En effet, c'est l'adresse de la variable tableau qui est transmise à la fonction, donc sa « référence » directe, et non pas la (ou les) valeur(s) rangée(s)

---

[9] Un pointeur est une variable dont la valeur n'est pas celle d'un type de base (ou structuré), mais d'un autre type, très différent des types de base : un type « adresse » (plus précisément, un type adresse de variable d'un type donné, par exemple, adresse d'une variable de type int). On peut alors avoir affaire à des appels un peu particulier comme :

```
procedure  ( ... ,  &i, ...)
```

dans lequel un nom de variable est préfixé de l'opérateur d'adresse. Mais il ne s'agit en réalité pas d'un mode spécial de passage d'arguments : on passe bien toujours les arguments de type pointeurs par valeur. Dans ce cas, leurs valeurs de pointeurs (i.e. les adresses des variables transmises) ne sont effectivement pas modifiées par l'exécution de la fonction, mais il se trouve que, par contre, du fait qu'il s'agit d'adresses de variables, ces variables, elles, peuvent être modifiées. Le langage C fournit en effet le moyen de réaliser une affectation à partir d'une adresse.

à partir de cette adresse en mémoire. On peut donc initialiser les cellules d'un tableau en utilisant une fonction prenant en paramètre un tableau. Ce dernier sera modifié par la fonction :

```
#include <stdio.h>
#include <stdlib.h>
#define TAILLE 10

int initTab(int tab[TAILLE], int n)
{
      int i ;

      if ( n >= TAILLE)
      {
           printf("erreur : %d trop grand\n", n) ;
           return 0 ;
      }

      printf("Entrez %d entiers separes par des blancs\n",
                 n) ;
      for(i=0; i < n; i++)
           scanf("%d\t",&tab[i]);
}

int main() {

      int tableau[TAILLE];
      int i ;

      printf("Combien d'entiers voulez-vous entrer \? \n");
      scanf("%d", &i) ;

      i = initTab(tableau,i) ;

      if (i == 0)
           return EXIT_ERROR ;
      else
           return EXIT_SUCCESS ;
}
```

Exercices

1.Simuler l'exécution de la boucle for du dernier programme utilisant la fonction puissance jusqu'à i = 2 à l'aide d'un tableau fournissant les valeurs des variables et des paramètres formels des fonctions, en supposant que les arguments d'appel des fonctions sont évalués de gauche à droite.

2. Ecrire une fonction `puiss` qui permette de calculer la puissance entière d'un réel de type `float` (positif ou négatif).

3. Ecrire un programme qui demande à l'utilisateur d'entrer 3 valeurs réelles a, b, et c, et qui résout l'équation du second degré $a\,x^2 + b\,x + c = 0$ dans R. Le programme imprimera le nombre de solutions réelles (zéro, une ou deux), ainsi que leurs valeurs quand elles existent.

Pour simplifier l'écriture du programme, on écrira une fonction discriminant(a,b,c) qui calcule le discriminant du binôme et retourne sa valeur.

5.3 Variables locales et variables globales

Une **variable locale** est une variable qui n'est définie qu'à l'intérieur d'un bloc. Elle est donc « locale » à ce bloc, et inconnue en dehors de ce bloc. Si une variable locale à un bloc porte le même nom qu'une variable locale figurant dans un bloc englobant, son adresse mémoire, allouée dynamiquement lors de l'exécution du bloc interne, sera différente de celle de la variable de même nom figurant dans le bloc externe.

Ainsi, la fonction `main` suivante

```
int main ()
{
      int i= 100, j= 4 ;

      while (j-- > 0)
      {
           int i=0 ;      /* variable i locale au while */

           printf("%d  ", i) ;
           i++ ;
      }
      printf("\nVoila i : %d\n", i) ;
      return 0 ;
}
```

donnera les impressions suivantes à la console, lors d'une exécution du programme

```
0  1  2  3

Voila i : 100
```

En effet, la variable `i` introduite dans le bloc du `while` aura une adresse différente de celle de la variable `i` introduite dans le bloc du `main`. Sa durée de vie sera temporaire : elle n'existera que le temps de l'exécution de la boucle `while`, et, à l'intérieur de ce bloc, elle cache ou **masque** toute variable `i` définie dans un bloc englobant. De la même façon, une variable locale nommée `i` dans le bloc de définition d'une fonction viendra masquer la variable `i` introduite par la fonction main. Et de la même manière, les variables introduites par les paramètres formels des fonctions se comportent comme des variables locales au bloc de définition de la fonction.

Une **variable globale** (ou **externe**) est une variable définie en dehors de tout bloc, c'est-à-dire en dehors de tout bloc de définition de fonction, et en particulier, en dehors de la fonction `main`. En C, toutes les fonctions sont des variables globales, car, (contrairement à beaucoup d'autres langages, comme Pascal, java, etc.), on ne peut pas définir de fonctions à l'intérieur du bloc d'une autre fonction pour en restreindre la portée. Toutes les fonctions C sont donc des variables globales définies au même niveau que la fonction `main` .

Nous allons pour l'instant supposer que les variables globales que nous utiliserons sont déclarées en tête d'un même fichier[10], les variables de type fonction pouvant être définies plus loin dans le fichier. En fait, une variable globale définie dans un fichier n'est connue du compilateur qu'à partir de sa déclaration, mais pour que les choses soient plus claires, nous ferons figurer les déclarations des variables qui ne sont pas des fonctions en tête de fichier (avant toute définition de fonction), .

De cette manière, toutes les variables globales seront connues de toutes les fonctions définies dans le fichier et pourront servir de liens pour communiquer des informations globales aux différentes fonctions. Mais n'oublions pas que si une variable locale à une fonction porte le même nom qu'une variable globale, c'est la variable locale qui est référencée car elle cache la variable globale.

```
/* les déclarations des librairies */
#include <file1.h>
/* les definitions de constantes */
#define TAILLE 1000
/* les déclarations des variables globales (non fonctions) */
int tableau[TAILLE]
/* puis, les déclarations et/ou définitions de fonctions (dont la fonction principale : main) */
int f(int tab[TAILLE]) ;
int main ()
{
…
}
... etc ..
```

Figure 3. Organisation d'un fichier source

[10] Nous verrons dans un autre chapitre comment faire pour définir des variables ou des fonctions dans des fichiers différents et faire de la compilation « séparée » (chaque fichier peut être compilé séparément).

6 Tableaux

Comme nous l'avons défini section 3.3.3 du polycopié n°1, un type **tableau** désigne une collection de variables indexées (c'est-à-dire ordonnées et repérables par un index) appelées **cellules**, qui ont toutes le même type.

6.1 Tableaux à 1 dimension

Une variable de type tableau à une dimension est une collection de variables ordonnées de même type, les cellules du tableau, repérées par un indice simple pouvant varier entre 0 et un entier déterminant la taille du tableau.

6.1.1 Déclaration

Pour définir une variable `tab` de type tableau à une dimension, constitué de N cellules d'un type donné, on fait la déclaration :

type `tab[N];`

Le nom `tab` correspond à un identificateur quelconque. N doit désigner une valeur entière connue du compilateur pour qu'il puisse réserver une zone mémoire contiguë (de taille égale à N fois la taille nécessaire au codage d'une valeur de ce type). `N` est donc une constante entière. Elle ne peut pas être une variable, car la valeur d'une variable n'est connue qu'à l'exécution du programme et non en phase de compilation.

Pour accéder à une cellule du tableau `tab`, il faut préciser l'indice (ou position) de cette cellule dans le tableau[11]. Cette position peut être donnée sous la forme d'une expression dont la valeur est un entier :

`tab[` *expression* `]`

Attention : en C les cellules d'un tableau de taille N sont indicées de 0 à N-1. L'évaluation de *expression* doit donc retourner un entier positif ou nul *strictement* inférieur à N (sinon il se produira une erreur à l'exécution du programme).

La cellule `tab[` *expression* `]` est une expression qui peut être utilisée comme n'importe quel autre identificateur de variable, et, en particulier, on peut la faire figurer dans la partie gauche d'une affectation, ou encore lui appliquer les opérateurs d'adresse (`&`), d'incrémentation (`++`) ou de décrémentation (`--`), à gauche comme à droite.

Exemple

Le programme ci-dessous lit une suite de réels et les affiche ensuite dans l'ordre inverse (on suppose que le nombre `nb` de réels à lire est >=1).

```
#include <stdio.h>
#include <stdlib.h>
#define TAILLE_MAX 1000
```

[11] D'un point de vue formel, les crochets sont considérés comme un opérateur sur les variables de type tableau, nommé opérateur d'indexation.

```
int main()
{
      char reponse;
      float tableau[TAILLE_MAX];
      int i ;
      int nb=0; /* pour compter le nb de reels entres */

      do
      {
           printf("Entrez un réel :");
           scanf("%f", &tableau[nb]);
           nb = nb + 1 ;
           printf(" Voulez-vous continuer (o/n) ? ") ;
           scanf("%c",&reponse) ;
      } while ((reponse == 'o') && (nb < TAILLE_MAX));

      for(i=nb-1; i>=0; i--)
           printf("%f\t",tableau[i]);
      return EXIT_SUCCESS ;
}
```

La variable `tableau` est un tableau contenant 1000 cellules (ou variables) de type `float`. Les identificateurs de ces cellules sont `tableau[indice]` où `indice` varie entre `0` et `999`. Ces cellules peuvent figurer comme tout identificateur dans n'importe quelle expression. L'expression `&tableau[n]` désigne l'adresse de la cellule `tableau[n]` et permet de lire une valeur réelle avec `scanf` (grâce au format `%f`) pour initialiser la cellule.

Dans l'exécution du programme ci-dessus, les cellules de la variable `tableau` sont initialisées au fur et à mesure des passages dans la boucle `do-while`. Le premier passage est toujours effectué puisqu'il s'agit d'une boucle do-while, et les cellules ne sont initialisées que pour les indices compris entre `0` et `nb-1` (nombre de cellules lues quand on sort de boucle). C'est pourquoi le programme n'affiche que les cellules correspondant à ces indices dans la boucle `for` suivante.

6.1.2 Initialisation

On a vu dans l'exemple du paragraphe précédent comment initialiser les cellules d'un tableau à partir d'une boucle, en lisant des données entrées par l'utilisateur. Très souvent, on initialisera les tableaux numériques en parcourant une à une leurs cellules dans une boucle :

```
void initZero(int tab[TAILLE])
{
      for (i=0 ; i < TAILLE ; i++)
           tab [ i ] = 0 ;
}
```

Mais il est également possible d'initialiser une variable de type tableau lors de sa définition comme suit :

type `tab[N] = {` *val1* `,` *val2* `, ...,` *valP* `};`

pourvu que les valeurs valI soient du bon type et que leur nombre P n'excède pas la taille N du tableau. A la suite de cette définition, les P premières cellules seront initialisées, et on aura :

`tab[0]` == *val1*, `tab[1]` == *val2*, …, `tab[P-1]` == *valP*

Remarques :

1) Si le nombre P des valeurs introduites est plus petit que N, et si le type des cellules du tableau est un type numérique, les dernières cellules seront automatiquement initialisées à 0 (parce qu'il s'agit d'un tableau d'entiers).

2) En outre, on peut aussi *ne pas préciser la taille N* du tableau et l'initialiser lors de sa définition comme suit:

type `tab[] = {` *val1* `,` *val2* `, ..,` *valP* `};`

Mais attention, dans ce dernier cas, le tableau aura *exactement* la taille P correspondant au nombre de valeurs introduites pour l'initialisation dans sa définition.

Astuces intéressantes

Pour connaître la taille d'un tableau, on peut utiliser l'opérateur `sizeof` qui prend en argument un identificateur de variable ou un type :

```
#define TAILLE  sizeof(tab)/sizeof(tab[0])
```

ou plus simplement, si on sait que `tab` est un tableau de `float` par exemple:

```
#define TAILLE  sizeof(tab)/sizeof(float)
```

En effet, l'opérateur `sizeof` retourne la taille en octets d'un type de données, mais on peut aussi l'appliquer à des variables (dans ce cas, il n'évalue pas la variable, mais retourne la taille de son type).

En outre, pour des tableaux de caractères, sizeof(char) = 1, donc la taille de la place mémoire occupée par une variable de type tableau de `char` est directement obtenue par la formule `sizeof(tab)`. Mais attention, cela ne coïncide pas nécessairement avec ce que l'on appelle la longueur de la "chaîne de caractères" rangée dans le tableau, car la fin d'une chaîne de caractère sera indiquée par la présence d'un caractère '`\0`' qui pourra se trouver rangé n'importe où dans la variable de type tableau.

6.1.3 Les tableaux de caractères

Nous avons vu que bien que le langage C ne comportait pas de type de base *chaîne de caractères*, il reconnaissait des constantes de ce type. Ces constantes sont simplement constituées de caractères entourés de doubles guillemets. Les caractères d'une chaîne ne représentent qu'eux-mêmes et ne sont pas interprétés par le compilateur, à l'inverse des autres caractères du programme qui constituent différentes unités lexicales (comme des nombres, des identificateurs, des mots réservés, etc.). Ainsi, la chaîne constante `"main"` n'a aucun rapport avec l'identificateur `main`, `"123"` n'est pas un `nombre` et la chaîne `"+"` n'est pas un opérateur`.`

Une chaîne de caractères est représentée en C par un tableau à une dimension de type `char`. Mais sa longueur pouvant être arbitraire, un caractère spécial est utilisé pour indiquer où se termine le tableau en mémoire. Ainsi, la constante de type chaîne de caractères `"bonjour"`,

est représentée en C par le tableau contenant les code des caractères qui la constituent (leur code ASCII quand il s'agit de caractères ASCII), placés les uns derrière les autres :

"bonjour" =>

'b'	'o'	'n'	'j'	'o'	'u'	'r'	'\0'

Le caractère `'\0'` est le caractère utilisé pour marquer la fin de la représentation d'une chaîne de caractères, celle-ci pouvant ainsi bénéficier longueur quelconque, tout en étant quand même rangée dans un tableau (qui, par définition, est une variable de taille fixe). On dit que `'\0'` est un caractère de contrôle, car il est utilisé pour controler la représentation, mais n'en fait pas vraiment partie. Ce n'est pas un caractère affichable, et il présente l'avantage d'être facile à tester, car c'est le premier caractère du code ASCII, c'est-à-dire celui dont le code est zéro.

En conséquence, on fera bien attention de distinguer le caractère `'c'` et la chaîne d'un seul caractère `"c"`. `'c'` est une valeur (constante) de type `char` alors que `"c"` est une constante de type tableau, dont les deux premières cellules sont de type `char` et qui contient respectivement les deux caractères `'c'` et `'\0'`.

De manière analogue, pour définir une variable correspondant au type abstrait *chaîne de caractères*, on peut utiliser une variable de type tableau à une dimension de type `char`. Par exemple, dans la définition ci-dessous,

```
char mot[80];
```

`mot` est une variable de type tableau contenant 80 cellules de type `char`.

Il est possible de l'initialiser lors de sa définition avec :

```
char mot[80]={'b','o','n','j','o','u','r','\0'};
```

qui sera exactement équivalent à

```
char mot[80]="bonjour";
```

index	0	1	2	3	4	...				79
mot	'b'	'o'	'n'	'j'	'o'	'u'	'r'	'\0'	...	

mais on peut aussi le faire avec

```
char mot[]={'b','o','n','j','o','u','r','\0'};
```

également équivalent à

```
char mot[]="bonjour";
```

Mais attention, avec les deux premières définitions, le compilateur aura alloué 80 cellules de type `char` dont les premières seulement auront été initialisées, tandis qu'avec les deux dernières, il n'aura réservé que l'espace mémoire nécessaire au stockage de la chaîne `"bonjour"`, c'est-à-dire, un tableau de taille 8 seulement.

index	0	1	2	3	4	...	7
mot	'b'	'o'	'n'	'j'	'o'	'u'	'r'	'\0'

Il est également possible, une fois la variable `mot` définie, d'accéder et d'initialiser les différentes cellules comme suit :

```
mot[0]='b'; mot[1]='o'; mot[2]='n'; mot[3]='j'; mot[4]='o';
mot[5]='u'; mot[6]='r'; mot[7]='\0';
```

On sait déjà imprimer une constante de type chaîne de caractères avec `printf` :

```
printf("Voulez-vous continuer ?");
```

Si l'on souhaite afficher le contenu d'une variable `mot` de type chaîne de caractère, on pourra utiliser le format `%s` :

```
printf("%s", mot);
```

ce format indique que l'argument suivant doit être affiché comme chaîne de caractère (s comme *string* en anglais), c'est-à-dire que les bits en mémoire sont imprimés comme caractères, jusqu'à la rencontre d'un caractère `'\0'` qui indique la fin de la chaîne.

Rappelons que pour lire une valeur de type chaîne de caractères dans une variable de type tableau, on peut aussi utiliser la fonction `scanf` (sans utiliser l'opérateur d'adresse) :

```
scanf("%s", mot);
```

Mais la fonction `scanf` s'arrêtera au premier caractères « blancs » (espace, tabulations, nouvelle ligne).

Le format de lecture `%s` de `scanf` permet en effet de lire une chaîne de caractères sans guillemets et sans caractères d'espacement. Ces caractères servent en effet de séparateurs à la fonction qui ne saurait pas où s'arrêter de lire sinon.

Il faut en outre être assuré que la valeur lue (de type chaîne de caractères) aura un nombre de caractères strictement inférieur au nombre de cellules de la variable tableau `mot`, qui, rappelons-le, est nécessairemebnt de taille fixe. Si la chaîne de caractères lue a une taille supérieure à celle du tableau `mot`, le programme sortira en erreur. Par contre, si la taille de la chaîne lue est inférieure à la taille du tableau `mot`, le caractère `'\0'` sera automatiquement inséré pour marquer la fin des caractères significatifs de la chaîne. Ainsi, supposons que le tableau de caractères `mot` ait été déclaré par

```
char mot[80] ;
```

L'instruction

```
scanf("%s", mot);
```

si l'utilisateur tape les caractères de « bonjour » suivi d'un caractère blanc, aura pour effet de stocker les caractères de « bonjour » dans la chaîne, comme si on l'avait initialisée dans le programme avec la série d'instructions :

```
mot[0]='b'; mot[1]='o'; mot[2]='n'; mot[3]='j'; mot[4]='o';
mot[5]='u'; mot[6]='r'; mot[7]='\0';
```

Signalons également une variante intéressante de `scanf` : la fonction de la librairie standard `sscanf` est équivalente à la fonction `scanf(…)` excepté que les caractères sont lus à partir

d'une chaîne de caractères donnée en premier argument à la fonction[12] (au lieu d'être lus à partir du clavier).

Rappelons également que les fonctions `getchar()` et `putchar(c)` permettent de lire ou d'écrire des caractères un à un, et l'on pourra ainsi grâce à `getchar()` écrire une boucle permettant de saisir des chaînes de caractères contenant des caractères d'espacement, si le besoin s'en fait sentir.

Exemple :

Une fonction qui lit une chaine de caractères tapée au clavier par un utilisateur et l'enregistre dans une variable de type tableau de caractères en supprimant au passage les espaces et les tabulations :

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

#define MAX 100

void lire(int tableau[MAX])
{
      int i=0 ;
      char c ;

      c = getchar();
      while ((c!='\n') && (i<MAX))
      {
                if(c!='\t' && c!=' ')
                     tableau[i++]=c;
                /* sinon i n'est pas incremente */
                /* et '\t' ou ' ' ne sont pas recopies */
                c = getchar();
      }
      tableau[i]='\0' ;
}

int main()
{
      char phrase[MAX];

      printf("Entrez une phrase et \n");
      printf("terminez la par un retour a la ligne :\n");

      lire(phrase) ;

      printf("les caractères (collés) : %s\n", phrase);
      return EXIT_SUCCESS ;
}
```

[12] Ce cas se produit relativement souvent, car, comme on le verra plus loin, les arguments du programme (ceux que l'on peut donner au lancement du programme comme paramètres de la fonction `main`) sont nécessairement de type chaîne de caractères.

6.1.4 Exemples de fonctions manipulant des tableaux de caractères

Dans la bibliothèque string (fichier d'entête `<string.h>`), on trouve de nombreuses fonctions intéressantes qui manipulent des pointeurs sur des `char`. La plupart d'entre elles peuvent s'écrire avec des variables de type tableaux (car, comme nous le verrons, un identificateur de tableau peut être traité comme une variable de type pointeur constant). Mais toutes ses fonctions ne sont pas transposables, car un certain nombre d'entre elles retournent une valeur (de type pointeur), et l'on ne peut pas retourner de valeur de type tableau.

Nous avons déjà évoqué la fonction permettant de calculer la longueur d'une chaîne de caractères (à savoir le nombre de caractères significatifs – i.e. ceux qui précèdent le caractère '`\0`'). La fonction standard s'appelle `strlen` (pour *string lenght*) et a le prototype suivant :

```
int strlen(const char chaine[]) ;
```

`chaine` peut être, soit une constante de type chaîne de caractères, soit un identificateur de tableau de caractères, (soit, comme nous le verrons plus tard, une variable de type pointeur sur un caractère).

Remarquons que la dimension du tableau n'a pas été spécifiée. Il est en effet possible d'écrire des fonctions ayant un paramètre de type tableau sans en spécifier la taille, à condition que le tableau soit de dimension 1. (L'inconvénient est que cela peut produire des erreurs d'exécution, si le programme tente d'accéder à une cellule dont la position serait supérieure à la taille effective du tableau passé en argument. Il faudra donc, de manière générale, prendre des précautions particulières).

Le mot `const` est un qualificatif qui s'applique à une déclaration de variable et qui signifie que la variable restera constante. L'appliquer à une variable globale ne présente pas beaucoup d'intérêt, car on peut alors aussi bien utiliser une constante symbolique. Mais, appliqué à un paramètre formel de fonction, ce terme permet de protéger la variable passée en argument de toutes modifications, juste pendant l'exécution de la fonction. Le compilateur signalera en effet une erreur si le programme tente d'écrire dans les cellules de cette variable, dans le bloc de définition de la fonction.

Exercice

Programmer cette fonction en utilisant une boucle `for`. Signalons que :

```
int longueur ;
longueur = strlen("bonjour") ; /* longueur vaut 7 */
```

Autre exemple : une procédure permettant de copier une chaîne dans une autre[13] (sorte d'affectation) :

---

[13] Signalons pour les amateurs de langage C sophistiqué, la version plus condensée :

```
void strcopy (char s[], const char t[])
{
      int i=0 ;

      while ( (s[i] = t[i++]) != '\0')
           ;
}
```

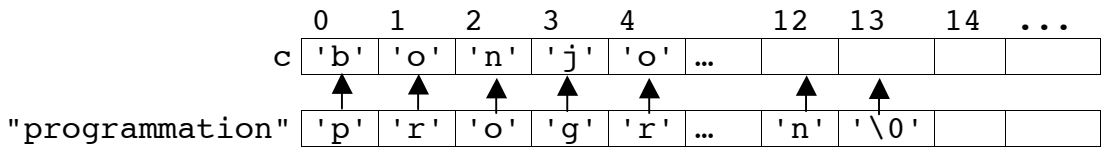
```
void strcopy (char s[], const char t[]) {
      int i ;
      i = 0 ;
      while ( t[i] != '\0') {
           s[i] = t[i] ;
           i++;
      }
}
```

Cette procédure recopie le tableau de caractères `t` (fourni en second argument), dans le tableau de caractères `s` passé en premier argument. La copie est réalisée caractère par caractère, en parcourant les caractères du tableau `t` un à un, jusqu'à rencontrer le caractère de fin `'\0'`. Bien entendu, le tableau `s` est modifié, puisque ses cellules `s[i]` seront affectées, mais il faudra aussi prendre garde à ce que sa taille soit suffisamment grande pour que tous les caractères du tableau `t` puissent être recopiés dans ses cellules. Sa taille doit donc être au moins égale à celle du tableau recopié (sinon, il se produit une erreur d'exécution, lorsque le programme essayera d'accéder à `s [i]` pour un `i` trop grand).

Exemple

```
char c[100]= "bonjour" ;

/* afficher la chaîne «programmation» en la recopiant
 * dans la variable c
 */
printf(«%s», strcpy(c, "programmation") );
```



La fonction standard de la bibliothèque string s'appelle `strcpy` et retourne un pointeur.

Exercices

1) En vous inspirant de l'écriture de la procédure `strcopy`, écrire une procédure `strcat` permettant de **concaténer** deux chaînes de caractères :

```
void strcat(char resultat[], const char source[]) ;
```

Cette fonction concatène la chaine `résultat` et la chaine `source` dans la chaine `resultat`. (On recherche d'abord la fin de la chaîne du premier paramètre formel, puis on recopie les cellules du deuxième paramètre à partir de cette position, jusqu'à rencontrer le caractère '\0').

Attention, ici aussi, la fonction suppose qu'il y a assez de place disponible dans la chaîne `résultat` pour y recopier à la fin le contenu de la chaîne source. Cette fonction devra donc toujours être utilisée avec beaucoup de précautions.

2) La fonction `strcmp` (pour *string compare*) de la bibliothèque standard permet de comparer deux chaînes de caractères. Elle pourrait avoir le prototype suivant :

```
int strcmp(char chaine1[], char chaine2[]) ;
```

Cette fonction ramène un entier qui est :

- inférieur à `0` si `<chaine1>` est inférieure à `<chaine2>` selon l'ordre lexicographique,
- égal à `0` si `<chaine1>` et `<chaine2>` ont les mêmes caractères,
- supérieur à `0` si `<chaine1>` est supérieure à `<chaine2>` selon l'ordre lexicographique.

La fonction parcourt les deux chaînes simultanément. Si elle rencontre deux caractères de même rang qui diffèrent, elle ramène la différence de code entre les deux, sinon, elle retourne zéro. Les caractères 'a' et 'b' diffèrent donc de 1, puisque b > a, et les caractères 'a' et 'z' diffèrent de 25, avec z > a. A titre d'exemple, les expressions suivantes sont vraies :

```
strcmp("b", "aie") == 1
strcmp("xxxaie","xxxb")== -1
strcmp("c","aie aie aie")== 2
strcmp("aie","c")== -2
strcmp("z","a")== 25
strcmp("a","z")== -25
```

6.2 Tableaux multi-dimensionnels

6.2.1 Tableaux à deux dimensions

Une variable de type tableau à deux dimensions est une collection de variables ordonnées de même type, repérées par deux indices pouvant respectivement varier entre 0 et deux entiers déterminant la taille du tableau. Pour définir une variable `mat` de type tableau à deux dimensions, constitué de M×N cellules, on écrit :

type `mat[M][N];`

où M et N doivent être des constantes entières.

Ce tableau est en fait un tableau de tableaux, réunissant M tableaux de taille N. Pour accéder à une cellule de ce tableau à deux dimensions, on utilise deux indices et l'opérateur d'indexation (noté par les crochets `[]`). Ainsi, la $j^{ème}$ cellule du $i^{ème}$ tableau de taille N s'obtient à partir de la variable `mat` et s'utilise comme n'importe quel identificateur de type *type* :

```
mat[i][j]
```

ici, `i` et `j` désignent des variables entières positives dont les valeurs sont comprises entre `0` et respectivement `M-1` et `N-1`. On peut bien entendu aussi accéder aux cellules du tableau à partir d'expression comme `mat[`*expr1*`][`*expr2*`]` , où les indices sont calculés à partir d'expressions quelconques, pourvu que ces dernières désignent des entiers restant dans les limites imposées par les dimensions du tableau.

On peut se représenter la variable tableau `mat` comme une matrice rectangulaire formée de M-1 lignes et N-1 colonnes (cf. Figure 4) ou encore, comme une suite de M tableaux de dimension 1 et de taille N mis bout à bout :

index 1	0				1				M-1			
index 2	0	1	…	N-1	0	…	N-1	…	0	1	…	N-1
mat	[0][0]	[0][1]	…	[0][N-1]	[1][0]	…	[1][N-1]	…	[M-1][0]	[M-1][1]	…	[M-1][N-1]

Cette dernière représentation correspond en réalité mieux à la manière dont seront stockées les valeurs des cellules d'un tableau en mémoire centrale.

mat	0	1	…	N-1
0	mat[0][0]	mat[0][1]	...	mat[0][N-1]
1	mat[1][0]	mat[1][1]	...	mat[1][N-1]
…	...	...	...	...
M-1	mat[M-1][0]	mat[M-1][1]	...	mat[M-1][N-1]

Figure 4. Le tableau mat [M] [N] à deux dimensions (taille MxN)

Exemple 1 :

La variable `mat` est utilisée ici pour représenter la matrice Identité.

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

#define DIM 5

/* ce programme initialise un tableau bi-dimensionnel
 * de taille DIMxDIM
 * qui represente la matrice identite
 * et l'affiche ensuite ligne a ligne */

void initIdentite(int tab[DIM][DIM]) ;

int main()
{
      int mat[DIM][DIM]; /* matrice identite */
      int i,j;            /* index1 et index2 */


      /* initialisation de l'identite */
      initIdentite(mat);

      /* impression de la matrice ligne par ligne */
      for(i=0; i<DIM; i++)
      {
           for(j=0; j<DIM; j++)
                printf("%d\t",mat[i][j]);
           printf("\n");
      }
      return EXIT_SUCCESS ;
```

```
}

void initIdentite(int tab[DIM][DIM])
{
      int i,j;

      for(i=0; i<DIM; i++)
      {
           for(j=0; j<DIM; j++)
                if(i==j)
                     tab[i][j]=1; /* sur la diagonale */
                else
                     tab[i][j]=0; /* ailleurs */
      }
}
```

Exemple 2 :

Le produit `mProd` de deux matrices, `mat1` et `mat2`, de taille respectives m1×n1 et m2×n2 (où chaque dimension est inférieure à `MAX`) est effectué par la fonction suivante :

```
/* calcul du produit de mat1 et mat2 dans mProd */
/* retourne 1 si effectue et 0 si echec */

int produitMat(int mat1[], int mat2[], int mProd[],
                  int m1, int n1, int m2, int n2)
{
      int i, j, k;

      if(n1 == m2)
           for (i=0; i <m1 ; i=i+1)
                for(j=0; j<n2 ; j=j+1)
                {
                     mProd[i][j]=0;
                     for(k=0; k<n1; k=k+1)
                          mProd[i][j] = mProd[i][j] +
                                    mat1[i][k]*mat2[k][j];
                }
      else
      {
      /* le produit n'est defini que si les tailles
       * n1 et m2 coincident. Dans ce cas, la matrice
       * produit est de taille m1xn2
       */
           printf("impossible : dim %d <> dim %d\n",n1,m2)
;
           return 0 ;
      }
      return 1 ;
}
```

Exercice :

Compléter le programme suivant pour qu'il calcule et imprime le produit de deux matrices :

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10


void initTab(int mat[], int m, int n) ;
void imprimTab(int mat[], int m, int n) ;
int produitMat(int mat1[], int mat2[], int mProd[],
                  int m1, int n1, int m2, int n2) ;
int main()
{
      int mat1 [MAX] [MAX];
      int mat2 [MAX] [MAX];
      int tab  [MAX] [MAX];

      int m1,n1,m2,n2, res;

/*lectures de m1, m2, n1 et n2 <= MAX */

      printf("entrez m1 et n1 ") ;
      printf("(dimensions  de la 1ere matrice) : ") ;
      scanf("%d%d", &m1, &n1) ;

      if ((m1 > MAX) || (n1 > MAX))
      {
           printf("taille maximum = %d", MAX);
           return EXIT_ERROR ;
      }

 ... /*ibid pour lire n2 et m2 */

/* puis initialisation des matrices mat1 et mat2 */
      initTab(mat1, m1, n1) ;
      initTab(mat2, m2, n2) ;

      res = produitMat(mat1, mat2, tab, m1, n1, m2, n2) ;
      if (res != 1)
           return EXIT_ERROR ;
      imprimTab(tab, m1, n2) ;
}
```

On peut initialiser un tableau à deux dimensions en faisant deux boucles for imbriquées. Pour compléter le programme précédent, on pourrait lire à la console la première matrice, entrée ligne par ligne, en faisant une (double) boucle du type :

```
      for(i=0; i<m; i++)
           for(j=0; j<n; j++)
                scanf("%d",&mat[i][j]);
```

Remarque

On peut aussi initialiser un tableau d'entiers à deux dimensions au moment de sa définition, en séparant les éléments du tableau par des virgules, et les lignes par des accolades. Par exemple les déclarations :

```
int mat[2][3]={ {1,2,3},{4,5,6} };
```

ou

```
int mat[2][3]={1,2,3,4,5,6};
```

effectuent les mêmes initialisations. Il faut cependant prendre garde à ne pas dépasser les limites des dimensions déclarées du tableau. Par contre, si des lignes ou des colonnes sont incomplètes, les termes manquants seront initialisés à 0. Ainsi,

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

initialise la première ligne du tableau à `{1, 0, 0}` et la seconde à `{4, 5, 0}`.

6.2.2 Tableaux à plus de deux dimensions

Ce que l'on a vu pour deux dimensions se généralise, et l'on peut définir une variable de type tableau à `k` dimensions, `k > 2`, de la façon suivante :

type identificateur `[N1][N2]...[Nk];`

On introduit ainsi l'identificateur d'une variable de type tableau contenant N1×N2×...×Nk cellules, les Ni étant des constantes positives entières. Chacune des cellules du tableau est une variable de type *type* qui peut être utilisée dans n'importe quelle expression. Pour accéder à une cellule du tableau, on utilise une expression

`identificateur [`*expr1*`][`*expr2*`]...[`*exprk*`]`

où les *expr1* désignent des entiers compris respectivement entre `0` et `NI-1`.

Un tableau à k dimensions peut toujours être considéré comme un tableau à une dimension dont les éléments sont des tableaux multidimensionnels de dimension k-1. On manipule alors plus aisément cette variable à l'aide de pointeurs, et nous reviendrons plus loin sur les tableaux multi-dimensionnels, dans la section qui traite du rapport entre les pointeurs et les tableaux.

6.3 Fonctions et tableaux

Il n'est pas possible d'écrire une fonction retournant comme valeur un objet de type tableau. On verra plus loin que l'on peut contourner cette difficulté en retournant dans ce cas un objet de type pointeur.

Par contre, il est possible de définir une fonction avec un paramètre formel de type tableau, et il est possible de passer une variable de type tableau en paramètre effectif à cette fonction. Dans ce cas, le paramètre formel de la fonction peut avoir été, soit complètement déclaré (c'est-à-dire, toutes dimensions indiquées), soit incomplètement déclaré.

Dans le premier cas (déclaration complète du paramètre formel), le paramètre effectif doit avoir *exactement* le même nombre de dimensions *et les mêmes bornes* que celles du paramètre formel.

Dans le deuxième cas, il doit aussi avoir le même nombre de dimensions, mais peut n'avoir déclaré aucune borne pour sa première (éventuellement unique) dimension. Bien entendu, pour ne pas risquer de débordement, il faut alors que la fonction ait un moyen de connaître cette borne, par exemple, en la passant aussi en paramètre.

Rappelons aussi, qu'à la différence de tout autre type de paramètres, *le langage C passe dans ce cas l'adresse du tableau à la fonction*, c'est-à-dire l'adresse de la première cellule, et non pas l'ensemble ordonné des valeurs contenues dans ces cellules, comme le voudrait l'appel par valeur. Dans ce cas, il y a **passage de paramètres par référence**, et, si l'on modifie le paramètre formel dans la fonction, c'est le paramètre effectif qui sera modifié.

Pour se prémunir (soi-même, en tant que programmeur) de la modification d'un paramètre formel de type tableau dans la fonction, on peut néanmoins qualifier ce paramètre formel dans sa déclaration du mot-clé `const` : dans ce cas, le compilateur signalera une erreur à toute tentative de modification des cellules du tableau dans la fonction. (On peut aussi initialiser un tableau au moment de sa déclaration et le déclarer `const` pour qu'il ne puisse plus ensuite être modifié).

Exemple

```
#define DIM1  12
#define DIM2  3

static void initTab (int tab[], int taille)
{
      while (taille != 0)
           tab[--taille] = 0 ;
}

static void imprimTab (int tab[], int taille)
{
      int i;
      for (i=0 ; i < taille; i++)
           printf("%d ",tab[i]) ;
}

int main ()
{
      int mois[DIM1] = { 1, 2, 3, 4, 5, 6,
                         7, 8, 9,10, 11, 12} ;
      const int protect[DIM2] = {4, 5, 6} ;

      initTab (mois, DIM1) ;
      imprimTab (mois, DIM1) ;
//    initTab (protect, DIM2) ;
      return 0 ;
}
```

Une exécution de ce programme imprimera la ligne :

```
0 0 0 0 0 0 0 0 0 0 0 0
```

ce qui montre que le tableau `mois` a été modifié (toutes ses cellules ont été réinitialisées à zéro) par la procédure `initTab`. Mais la compilation de ce programme avec la ligne supplémentaire (ici en commentaires), qui tente d'initialiser le tableau `protect`, détecterait

une erreur sur cette ligne, car `protect` est déclaré `const` contrairement au paramètre formel `tab`.

7 Structures et unions

7.1 Structures

Nous avons vu au paragraphe 3.3.2 que la définition d'un type structure pouvait prendre dans les cas simples la forme suivante :

```
 typedef struct {
      float angle;
      unsigned int  distance;
} Point;
Point P1, P2 ;
```

Mais dans les cas récursifs, cette méthode ne pourra pas être utilisée. C'est pourquoi, on lui préfère souvent la déclaration plus générale :

```
struct tag {
      type1  ListeDeChamps1 ;
      ...
      typeN  ListeDeChampsN ;
} ;
```

où *tag* est un identificateur permettant de donner un nom au bloc de déclarations des champs qui suit entre accolades. On peut ensuite utiliser ce nom *précédé du mot* `struct`, pour déclarer des variables de ce type de structures. Par exemple :

```
struct _Point {
      float angle;
      unsigned int  distance;
};
struct _Point P1, P2 ;
```

ou encore, définir le type `Point` et l'utiliser de la façon suivante :

```
typedef struct _Point Point ;

struct _Point {
      float angle;
      unsigned int  distance;
};
Point P1, P2 ;
```

La première ligne est une déclaration de synonymie qui permet d'utiliser le mot `Point` ensuite. Une telle déclaration ne définit par le type `Point`, mais permet de l'utiliser dans les déclarations qui suivent. Le type `Point` est synonyme du type `struct _Point`, et le type `struct _Point` est défini par les déclarations de champs figurant sur les quatre lignes suivantes.

Cette dernière manière de procéder peut être utilisée de manière systématique, et nous vous recommandons de l'adopter. Elle présente l'avantage de permettre d'introduire les définitions

des objets de type structure dans n'importe quel ordre (même si ces définitions font référence les unes aux autres[14]).

7.2 Opérations sur les structures

Supposons que l'on ait fait les déclarations suivantes :

```
typedef unsigned int Jour, Annee ;
typedef enum {JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN,
                JUILLET, AOUT, SEPTEMBRE, OCTOBRE,
NOVEMBRE,
                DECEMBRE }
typedef struct {
      Jour jour ;
      Mois  mois ;
      Annee annee ;
} Date ;
typedef struct {
      Date dateNaiss ;
      char nom [30] ;
      float taille ;
} Personne ;
```

Les déclarations de variables :

```
Date date ;
static const Date dateMarignan = {14, SEPTEMBRE, 1515};
Personne toi, moi = { {14, MARS, 1955}, "Cathy", 1.68 } ;
```

permettent d'introduire des variables dont certaines ont été directement initialisées au moment de leur déclaration. Les virgules permettent en effet de séparer les valeurs attribuées aux différents champs, et les accolades permettent de délimiter les structures (ou les tableaux).

Pour accéder aux différents champs d'une structure, on dispose de l'opérateur de sélection de champs '.', et en C ANSI, on dispose aussi d'une affectation généralisée permettant d'affecter tous les champs d'une structure avec les valeurs des champs d'une autre structure. De sorte que l'on pourra initialiser les variables `date` et `toi` de la façon suivante :

```
date = dateMarignan ;  /* affectation généralisée */

toi.nom = "Ahmed" ;
toi.dateNaiss.jour  = 2 ;
toi.dateNaiss.mois = OCTOBRE ;
toi.dateNaiss.annee = 1972 ;
toi.taille = 1.82 ;
```

Si l'on exécute ensuite l'instruction :

```
date.jour++ ;
```

---

[14] On perd apparemment en simplicité, mais cette deuxième manière de procéder permet aussi de définir des types de structures **récursives**, c'est-à-dire des définitions de types faisant référence à leurs propres types. (C'est en effet de cette manière que seront définies des listes ou des arbres d'éléments associés les uns aux autres).

on incrémentera de 1 les champs `jour` de la variable `date` (il vaudra donc 15). Mais on ne pourra pas écrire `date++`, car l'opérateur d'incrémentation ne peut porter que sur des variables de type entier (ici, le type de `date` est un type `struct` nommé `Date`).

7.3 Fonctions à paramètre de type structure

Depuis ANSI C, il existe une affectation généralisée sur les structures (celle-ci n'existait pas dans le langage C d'origine) et les paramètres de type `struct` sont passés par valeur (ce n'était pas le cas avant). Ces deux « nouveautés » du langage ANSI C font que l'on peut manipuler plus facilement des fonctions ayant des paramètres de type struct ou retournant des valeurs de type struct. En voici un exemple :

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

typedef struct {
      float re; /* partie reelle */
      float im; /* partie imaginaire */
} Complex;

Complex somme (Complex z1, Complex z2)
{
      Complex som ;

      som.re = z1.re + z2.re ;
      som.im = z1.im + z2.im ;
      return som ;
}

int main ()
{
      Complex a,b,z;

      printf("Entrez un nb complexe \n") ;
      printf("(partie réelle puis partie imaginaire) :");
      scanf("%f%f",&a.re,&a.im);
      printf("Entrez un autre nb complexe \n") ;
      printf("(partie réelle puis partie imaginaire) :");
      scanf("%f%f",&b.re,&b.im);
           /* calcul de la somme de a et b */
      z = somme (a, b);
      printf("Leur somme est : %.2f + %.2f i\n",
z.re,z.im);
      return EXIT_SUCCESS ;
}
```

Exercice

Complétez le programme précédent avec une procédure d'impression d'un nombre complexe et une fonction qui retourne le produit complexe de deux nombres complexes.

7.4 Unions

Il arrive parfois que seul un champ soit significatif à la fois dans une structure. Allouer de l'espace à chacun des champs est alors une perte d'espace mémoire. Une déclaration de type **union** est précisément faite pour donner une solution à ce problème. Une union est une sorte de structure (la déclaration est la même, en remplaçant le mot `struct` par le mot `union`), et elle donne de la même façon accès à des champs, mais en réalité ces champs sont comme, en quelque sorte, disposés les uns sur les autres, la zone mémoire dont on dispose étant simplement égale à celle du champ le plus encombrant. L'accès aux champs d'une union se fait avec le même opérateur de sélection que pour les struct : l'opérateur noté '.'. Un objet de type union est donc en réalité un objet dont le type est l'un quelconque des types proposés par les différents champs de l'union.

Par exemple, pour définir un type union réunissant différentes sortes de nombres, à partir des trois types de structures :

```
typedef struct {
      int isPremier; /* vaut 0, ou 1 selon le nb */
      int value ;
} Entier ;
typedef struct {
      int isRatio; /* vaut 0, 1 selon le nb */ ;
      float value ;
} Reel ;
typedef struct {
      int sorte ; /* indique si imagin = 0 ou non */
      float reelle ;
      float imagin ;
} Complex ;
```

Il suffit de définir une union des trois structures considérées :

```
typedef union {
      Entier entier ;          /* si c'est un entier */
      Reel reel ;         /* si c'est un reel */
      Complex complex ;   /* si c'est un nb complexe */
} Nombre ;

Nombre  nb;
```

On peut alors avoir accès aux champs `nb.entier.value`, `nb.reel.value` ou `nb.complex.reelle` et `nb.complex.sorte`. Les valeurs des champs d'une structure sont placées, comme pour les cellules d'un tableau, de manière contiguës en mémoire. (La seule différence avec un tableau et que les différents champs peuvent occuper des zones mémoires de taille différentes). On peut donc imaginer qu'en fait que ces différentes structures sont placées les unes sur les autres. Le fait d'utiliser à la fois l'union comme abritant une structure de type `Entier`, `Reel` ou `Complex` (lorsqu'on fait un accès à `nb.entier.value` par exemple) n'est pas problématique si on fait un usage cohérent de ces structures de données.

Mais le problème est qu'on ne saura pas nécessairement toujours quelle est la nature de l'objet qu'on manipule. La manière la plus simple de résoudre cette difficulté est d'introduire un champ supplémentaire, auquel aucun objet ne se réduira réellement, mais qui permettra d'en

identifier le type. Ici, par exemple, on peut définir nos différentes sortes de nombre de la façon suivante

```
typedef enum { ENTIER, REEL, COMPLEX} TypeNombre ;
typedef struct {
      const TypeNombre type=ENTIER ;
      int isPremier; /* vaut 0, ou 1 selon le nb */
      int value ;
} Entier ;
typedef struct {
      const TypeNombre type=REEL ;
      int isRatio; /* vaut 0, 1 selon le nb */ ;
      float value ;
} Reel ;
typedef struct {
      const TypeNombre type=COMPLEX ;
      int sorte ;
      float reelle ;
      float imagin ;
} Complex ;
```

Avant d'accéder aux différents champs, on effectuera un switch sur ce nouveau champ type :

```
switch(nb.type)
{
      case ENTIER :  on peut accéder à nb.entier.xxx ;
                      break ;
      case REEL :    accès à  nb.reel.yyy ;
                     break ;
      case COMPLEX : accès à nb.complex.zzz ;
                     break ;
      default :      break ;
}
```

De cette manière, on est assuré d'avoir toujours accès à la catégorie (le type) dont participe l'objet de ces différentes sortes, et une fois ce type déterminé, on en connaît en réalité la vraie nature, on peut accéder sans risque aux champs concernés.

Remarque 1

Les noms des champs sont complètement indépendants d'une structure/union à une autre, c'est-à-dire que si l'on utilise un même nom de champ pour deux structures différentes, cela ne pose pas de problème au compilateur. Ce qui est important dans ce qui précède n'est donc pas que les trois structures aient un même champ `sorte`, mais simplement que ce premier champ soit d'un même type (et occupe donc la même place). Mais on aurait pu aussi bien appeler ces champs `type`, comme dans l'union, ou au contraire prendre quatre noms totalement différents.

On traduit cette indépendance de noms en disant que chaque structure a son propre espace de noms. C fournit les espaces de noms suivants :
- un espace de noms pour les champs de chaque structure/union,
- un espace de noms pour les tags des structures/unions/enum,
- un espace de noms pour les étiquettes (pour les `goto`),

- un espace de noms pour les autres identificateurs : variables, fonctions, types, constantes d'énumération.

Ainsi, si on avait déclaré :

```
typedef struct {
      double re; /* partie reelle */
      double im; /* partie imaginaire */
} Complex;
Complex nb ;
...
int re, im ;
...

re = 1 ;        /* re est la variable simple de type int */
nb.re = -1 ;    /* re est ici le premier champ de la
                     variable nb de type Complex */
```

Remarque 2

On se sert souvent d' `union` de structures ayant en commun le type de leur premier champ (qui est d'ailleurs aussi généralement utilisé pour catégoriser la nature de l'objet). Cela permet parfois d'économiser de la place. Dans l'exemple des nombres que nous avons développé, on pourrait faire l'économie d'un champ dans chaque structure avec ce nouveau typage des nombres :

```
typedef enum { PREMIER, MULT, RATIO, REEL,
                                IMAG,REELC} TypeNombre ;
typedef struct {
      int type ; /* PREMIER ou MULT */
      int value ;
} Entier ;
typedef struct {
      int type; /* RATIO ou REEL */ ;
      float value ;
} Reel ;
typedef struct {
      int sorte ; /* IMAG ou REELC */
      float reelle ;
      float imagin ;
} Complex ;
```

Bibliographie

[1] *Le langage C, norme ANSI*, Kernigham & Ritchie, 2e ed., Dunod.

[2] *Belle programmation et langage C*, Yves Noyelle, cours de l'école Supérieur d'Electricité dans la collection TECHNOSUP, Ellipses Editions, 2001.

[3] *Initiation à la programmation*, C. Delannoy, Editions Eyrolles.