Chapitre 9

Procédures et fonctions (suite et fin)

9.1 Retour sur les paramètres

Lors de l'appel d'un sous-programme, nous avons vu que le programme appelant devait pouvoir dialoguer avec le sous-programme. Ce dialogue se faisait dans les deux sens :

1. le programme passait des valeurs au sous-programme (des paramètres),
2. le programme recevait une valeur de retour (à condition que le sous-programme soit une fonction).

En réalité, la situation est un peu plus complexe que cela. Il existe en fait, outre la valeur de retour d'une fonction, trois manières de faire passer un argument entre le programme appelant et le sous-programme [1]. Chaque paramètre peut être passé d'une des manières suivantes :

- passage du *paramètre par valeur* : le sous-programme reçoit son argument sous la forme d'une valeur ;
- passage du *paramètre par adresse* : le sous-programme reçoit l'adresse de l'argument, ce qui l'autorise à en modifier le contenu ;
- passage par *variable globale* : en accédant aux variables globales, le sous-programme peut modifier, parfois de manière insidieuse, le comportement du programme principal [2].

1. Nous ne nous occuperons pas du quatrième cas, celui des passages de paramètres par nom, parce que cette méthode est rarement implémentée dans les langages courant (Algol 60 possédait cette caractéristique, mais ses héritiers ne l'ont pas conservée). Ce passage ressemble au passage par adresse dans la plupart des cas, mais il y a des différences. Si on passe à un paramètre `iParaNom` l'argument `iTab[iI]`, la séquence suivante affectera la valeur 0 à la cellule `iTab[7]`.

```
iI<- 7
iParaNom <- 0
```

On peut raisonnablement s'interroger sur la lisibilité d'une telle sous-routine. L'expansion des macros en ADL, langage de dBase II, présentait un comportement similaire. Cette caractéristique avait été retirée de la version compilée, parce qu'elle produisait des résultats imprévisibles. J'ai connu personnellement quelques plantages de ce type.

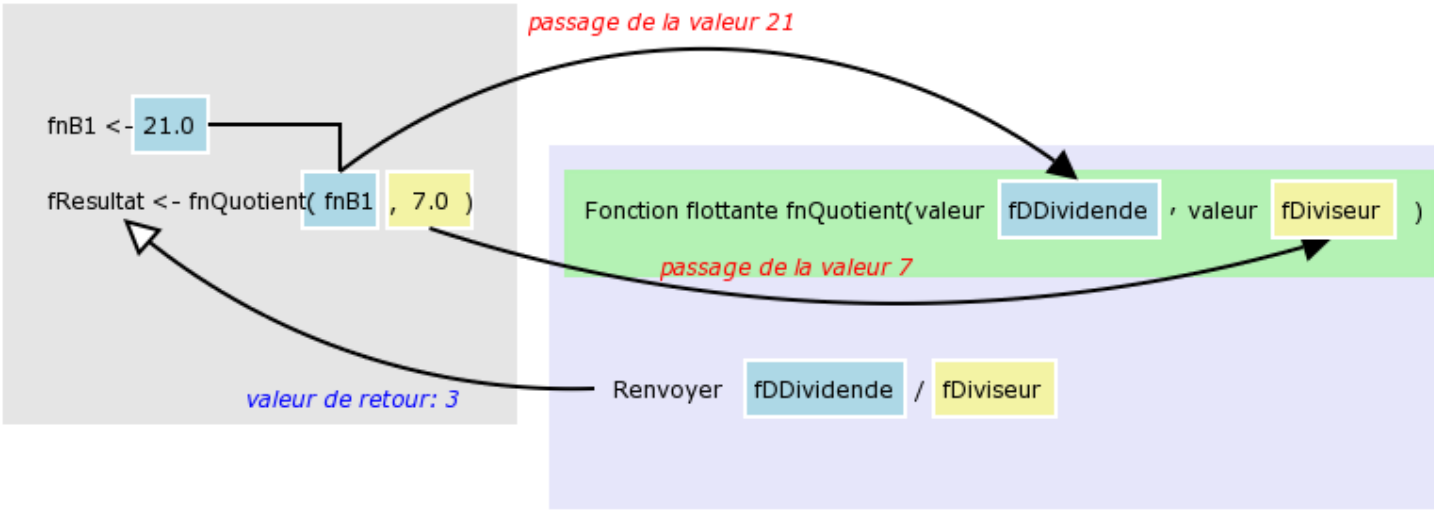
2. Une autre typologie parle de trois modes de passages de paramètres explicites :
   - par *valeur*
   - par *valeur/résultat*, ce qui donne le même résultat que par adresse. On insiste plus ici sur les résultats que sur la manière. On pourrait arguer que LAF utilise un passage par valeur/résultat puisqu'il ne manipule pas l'adresse. En C, si on passe l'adresse d'une variable, il est parfaitement possible de l'utiliser comme adresse de base d'un tableau, ce que LAF ne permettrait pas.
   - par *résultat* : la valeur initiale de la variable du programme appelant n'est pas prise en compte, elle n'est donc pas utilisable dans le corps de la sous-routine. On peut simuler ce comportement en n'utilisant pas la valeur d'un paramètre passé par adresse.

9.1.1 Passage par valeur

Nous avons vu qu'une fonction peut avoir un ou plusieurs paramètres.

```
/* == Etudes des passages de paramètres 1 ==*/
Prototype Fonction Flottante fnQuotient
  (Valeur Flottante fDividende, Valeur Flottante fDiviseur)
Début du traitement
   Déf Locales
      Variables Flottantes fNb1, fResultat
   Fin des Locales
   fNb1 <- 21.0
   fResultat <-  fnQuotient(fNb1,7)
   Afficher "Résultat : " & fResultat
Fin du traitement
Code Fonction Flottante fnQuotient
   (Valeur Flottante fDividende, Valeur Flottante fDiviseur)
   Renvoyer fDividende/fDiviseur
Fin de fonction
```

Que se passe-t-il exactement lors de l'appel de la fonction ? En fait, les paramètres formels `fDividende` et `fDiviseur` sont considérés dans le corps de la fonction comme des variables [3]. Au moment de l'appel de la fonction, les valeurs placées entre parenthèses (les arguments) sont attribuées aux paramètres formels.



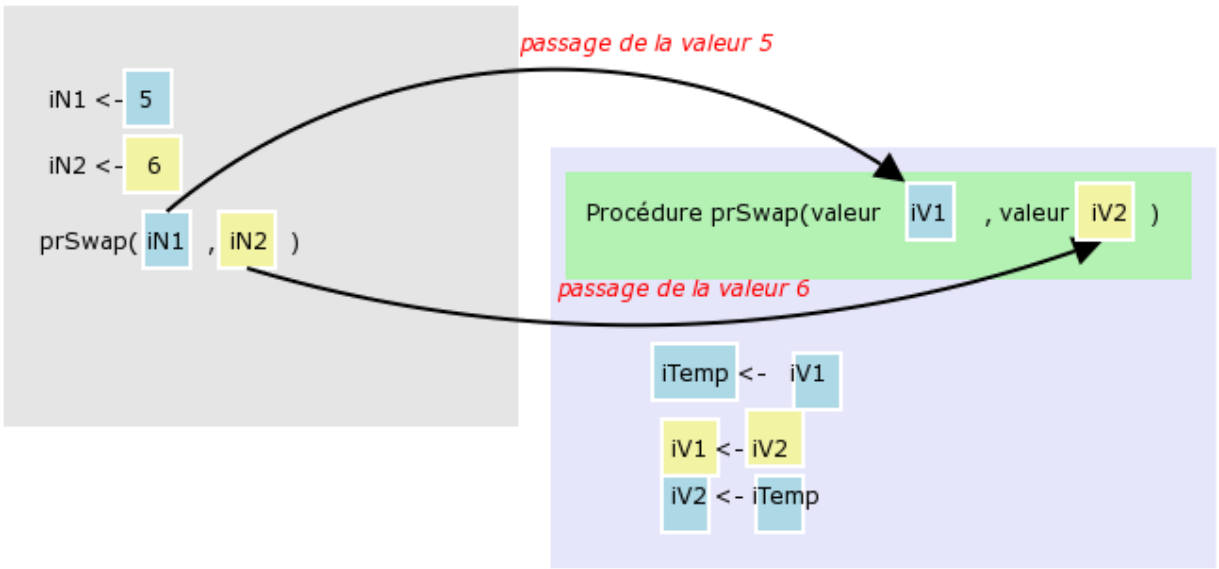
Dans cet exemple, les paramètres du programme principal sont « passés » à la fonction `fnQuotient()` sous la forme de valeurs. Quoi que nous fassions subir aux paramètres `fDividende` et `fDiviseur`, la variable `fNb1` reste intacte, puisqu'elle appartient au programme principal.

3. Ce ne sont pas vraiment des variables dans la mesure où le programme les traite différemment, spécialement en leur attribuant une valeur initiale, calculée dans le programme appelant. En outre, si une variable locale de même nom est créé dans le sous-programme, certains langages, dont le C, ne signalent pas d'erreur. Ils se contentent de masquer les paramètres comme s'il s'agissait d'une variable globale.

9.1.2 Passage par adresse ou référence

Examinons un autre exemple, dans lequel nous allons essayer d'écrire une procédure qui inverse les valeurs de deux variables.

```
/*== Version expérimentale  et inutile !!! ==*/

Prototype Procédure prSwap1
  (Valeur Entière iV1, Valeur Entière iV2)

Début du traitement
   Déf Locales
      Variables Entières iN1, iN2
   Fin des Locales
   iN1 <- 5
   iN2 <- 6
   prSwap1(iN1, iN2)
   Afficher iN1   Fin de Ligne
   Afficher iN2
Fin du traitement

Code Procédure prSwap1

(Valeur Entière iV1, Valeur Entière iV2)

  Déf Locales
     Variable Entière iTemp
  Fin des Locales
  iTemp <- iV1
  iV1 <- iV2
  iV2 <- iTemp
Fin de Procédure
```



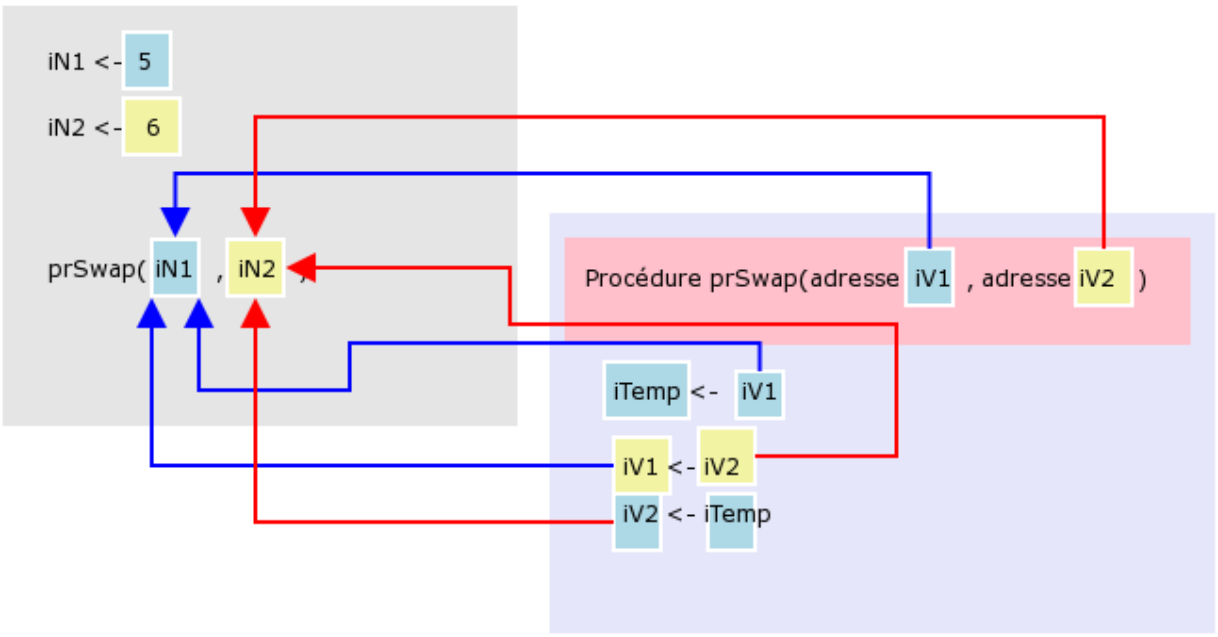
Nous voyons que la procédure inverse les valeurs de `iV1` et `iV2`, mais que les variables `iN1` et `iN2` ne sont pas concernées par cette manipulation. Nous constatons, avec dépit, que

le programme principal affiche 5 et 6. Notre procédure ne sert à rien. Il nous faut trouver un autre moyen pour réaliser notre inversion de variables.

Au lieu de passer la valeur des deux variables, nous allons passer l'adresse des variables. Pour cela, nous utiliserons une notation particulière `Adresse` devant chaque paramètre.

```
Code Procédure prSwap
(Adresse Entière iV1, Adresse Entière iV2)
```

En précisant cela, nous signalons que toute mention des paramètres `iV1`ou `iV2` dans la procédure se réfère en fait aux variables mentionnées lors de l'appel de la procédure (`iN1` ou `iN2` dans notre exemple).




```
/*== Version correcte complète ==*/
Prototype Procédure prSwap
   (Adresse Entière iV1, Adresse Entière iV2)

Début du traitement
   Déf Locales
     Variables Entières iN1, iN2
   Fin des Locales
   iN1 <- 5
   iN2 <- 6
   prSwap(iN1, iN2)
   Afficher iN1   Fin de Ligne
   Afficher iN2
   Attente
Fin du traitement

Code Procédure prSwap
(Adresse Entière iV1, Adresse Entière iV2)
  Déf Locales
    Variable Entière iTemp
  Fin des Locales
  iTemp <- iV1
```

```
  iV1 <- iV2
  iV2 <- iTemp
Fin de Procédure
```

Remarquons immédiatement que tout sous-programme qui reçoit un paramètre par adresse est susceptible de produire des résultats imprévisibles dans le module appelant. Ainsi, dans notre prochain exemple, rien ne nous dit que la procédure `prMaProcedure` ne modifie pas la valeur de la variable `iDiviseur`, par exemple, en lui affectant une valeur nulle. La valeur de `fQuotient` est par conséquent imprévisible au seul vu des lignes qui suivent :

```
iDividende <- 10
iDiviseur <- 5
prMaProcedure(iDiviseur)
fQuotient <- iDividende / iDiviseur
```

Cela nous impose deux mesures :
- utiliser les passages par adresse avec une grande prudence et uniquement lorsque c'est nécessaire, par exemple dans le cas de notre procédure `prSwap`.
- prendre soin de documenter les parties de code qui utilisent les passages par adresse. Le nom de la sous-routine est déjà une manière de documenter.

9.1.3 Passages par variables «globales»

A proprement parler, l'utilisation d'une variable globale dans un sous-programme n'est pas un passage de paramètre ou alors il faut considérer que toutes les variables globales sont passées comme paramètres à tous les sous-programmes. L'utilisation des variables globales dans une sous-routine présente plusieurs inconvénients :
- la sous-routine ne peut fonctionner que dans un programme qui possède des variables portant un nom précis, ce qui rend la réutilisation de la routine problématique dans un autre programme.
- la modification d'une variable globale peut entraîner des inconvénients sérieux au niveau du programme principal.
- un tel mode de passage interdit tout accès d'une sous-routine aux valeurs d'une variable locale dans le programme principal ou dans une autre routine.

9.1.4 Choix du type de passage des paramètres

Il faut insister sur le fait que le type de passage concerne chaque paramètre individuellement et qu'on peut donc, pour une même sous-routine, mélanger les trois types, si la raison s'en fait sentir.

le choix par défaut : par valeur

Le passage d'un paramètre par valeur présente beaucoup d'avantages :
- il interdit toute modification inattendue d'une variable locale par une sous-routine mal conçue ;
- il permet d'appeler la sous-routine avec des paramètres qui sont des expressions (sans nous obliger à déclarer ni à initialiser une variable).

Il présente néanmoins un inconvénient majeur :
- il ne permet pas de modifier une variable. C'est gênant si la raison d'être de la sous-routine est de réaliser une telle modification.

En langage C, nous allons voir que tous les arguments sont passés par valeur, à l'exception des tableaux [4]. Il est par contre plus étonnant de voir que Visual BASIC a choisi le passage par valeur (`ByVal`) comme une exception.

le choix de raison : par adresse

Quand il n'est pas possible de faire autrement, on passera l'argument par adresse. La raison en est toujours la même :

– nécessité pour la sous-routine de modifier une variable (exemple de `scanf()` en C).

On sait cependant que ce passage par adresse équivaut à donner un chèque en blanc à la sous-routine. L'utilisateur d'une sous-routine qui demande des arguments par adresse n'est jamais certain que ceux-ci lui seront rendus intacts.

le choix suicidaire : par variable globale

La définition d'une variable globale revient à autoriser n'importe quelle sous-routine à la modifier. Si, en pratique, il est possible de limiter cette autorisation à certains contextes, l'utilisation des variables globales doit obéir aux règles suivantes :

– on doit limiter l'utilisation des variables globales au minimum nécessaire ;
– les sous-programmes doivent se limiter dans la mesure du possible à **lire** les variables globales ;
– les rares cas où un sous-programme modifie les variables globales doivent être pleinement justifiés et clairement documentés (ne serait-ce que par le nom du sous-programme, par exemple `prMiseAJour_NomClient` pour modifier la variable `sNomClient`). On pourrait même envisager d'obliger ces sous-programmes à utiliser un paramètre par adresse pour ces cas-là.

Prenons l'exemple d'une application qui gère la sécurité dans une base aérienne. On pourrait imaginer qu'une variable globale contienne le niveau d'alerte (de 0 à 5) pour éviter de devoir passer à chaque procédure la valeur de l'état d'alerte et alourdir ainsi l'écriture du programme. On pourrait par contre spécifier que personne n'a le droit d'écrire une instruction du genre :

```
iEtatAlerte <- X
```

La modification de l'état d'alerte se ferait uniquement à l'aide d'une procédure présentant plusieurs paramètres, comme par exemple :

```
Prototype Procédure prChangeAlerte
  (Valeur Entière iNouvelEtat, Valeur Entière iCause)
```

La programmation objet permettrait de résoudre notre problème d'alerte dans une base au moyen d'une classe *Alerte*. Celle-ci comporterait un attribut caché (une sorte de variable locale) et deux méthodes : *NiveauAlerte()* serait une fonction renvoyant le niveau d'alerte courant, qui pourrait être utilisée par n'importe quelle autre partie du code et une méthode *ChangeAlerte()* semblable à la procédure ci-dessus, dont on pourrait éventuellement essayer de limiter la portée.

9.2 Les sous-programmes en C

C est le pire langage qui soit pour l'étude des sous-programmes. Il ne possède pas de procédures et ne dispose que d'un seul type de passage des paramètres, ce qui le rend peu apte

4. Cette exception n'en est pas une, puisqu'une variable tableau en C est en fait un pointeur constant sur une variable du type constitutif du tableau. C'est donc en fait la valeur de cette constante qui est passée.

à illustrer notre propos théorique. Nous reviendrons sur le problème des paramètres dans la section suivante.

9.2.1 Absence des procédures

Le langage C ne dispose que d'un seul type de sous-programme : les fonctions. Les premières versions du langage imposaient donc toujours l'écriture de fonctions. On a, par la suite, introduit un type particulier, `void`, qui permet d'écrire des fonctions qui ne renvoient pas de valeur définie. Une procédure en C sera donc une fonction de type `void` qui ne comporte pas d'instruction de renvoi de valeur (en C `return`). Il en résulte que syntaxiquement, les procédures sont des fonctions.

9.2.2 Déclaration du prototype d'une fonction

Il n'existe pas de mot-clé pour définir une fonction en C (un fait rare). On distinguera une fonction d'une variable du même type par la présence des parenthèses encadrant les paramètres.

```
int Var1 ;       // Déclaration d'une variable
int Func() ;     // Déclaration d'une fonction
```

Le prototype de la fonction se termine par un point-virgule. Les paramètres éventuels figurent entre parenthèses et sont séparés par des virgules. A la différence des déclarations de variables, chaque paramètre doit être explicitement précédé de son type.

```
int Double(int N) ;
float Puissance(float Nombre, float Exposant) ;
void Effacement() ;
```

9.2.3 Le corps d'une fonction

La définition complète d'une fonction reprend l'en-tête suivi d'un bloc. Le bloc sera composé d'une paire d'accolades encadrant une série de déclarations de variables locales et une suite d'instructions. Parmi celles-ci, il ne faudra pas oublier `return`.

```
int Double(int iN)
{
  int iTemporaire ;
  iTemporaire = iN * 2 ;
  return iTemporaire ;
}
```

Notons que cette fonction pourrait s'écrire plus simplement :

```
int Double(int iN)
{   return iN * 2 ;
}
```

9.2.4 L'appel d'une fonction

L'appel d'une fonction se fait naturellement dans une expression. Au cas où la fonction est en fait une procédure, elle sera suivie d'un point-virgule.

```
main()
{X = Double(7) ;
 Effacement() ;
}
```

9.2.5 Les variables locales

Les variables locales ne présentent pas d'originalité par rapport aux variables globales que nous connaissions jusqu'à présent. Leur portée est simplement limitée à l'intérieur de la fonction.

9.3 Les passages de paramètres en C

9.3.1 Passage par variables globales

Le langage C autorise la définition de variables globales que n'importe quelle fonction peut modifier, avec les conséquences néfastes que nous avons vues. En pratique, nous verrons qu'il est possible de limiter la visibilité des certaines variables à une portion de programme :

- soit en plaçant la déclaration de cette variable après le corps de certaines fonctions (dans ce cas, la variable n'est vue que par les fonctions dont le corps suit sa déclaration).
- soit en limitant, au moyen du mot-clé `static`, la portée de la variable au module dans lequel elle est déclarée.

Voici un exemple de programme avec deux variables locales déclarées à des endroits différents.

```
int V1;
void machin();
float truc();
main()
{...
/* ne voit que V1 */
}
void machin()
{...
/* ne voit que V1 */
}
int V2;
float truc()
{...
/* voit V1 et V2 */
}
```

9.3.2 Passage par valeur

Sauf dans le cas d'un tableau, un passage de paramètre se fait toujours par valeur. Voici ce que deviendrait notre version erronée de `prSwap` en C :

```
void prSwap1(int iV1, int iV2)
{
 int iTemp;
```

```
 iTemp = iV1;
 iV1 = iV2;
 iV2 = iTemp;
}
```

Nous avons vu qu'une telle fonction se révélait incapable de modifier les variables de la fonction appelante.

9.3.3 Passage par adresse

Le passage par adresse (sauf dans le cas d'un tableau) est impossible en C. Il faut donc se résoudre à utiliser une autre technique.

Définition de la fonction

Pour chaque variable dont on veut passer l'adresse, il faut définir au niveau de la fonction, une variable pointeur correspondant au type du paramètre.

Type de variable transmise		
`int`	`pointeur sur int`	`int *`
`float`	`pointeur sur float`	`float *`
`char`	`pointeur sur char`	`char *`

Dans le cas de notre procédure `prSwap`, il aurait fallu définir son prototype de la manière suivante :

```
void prSwap(int * iV1, int * iV2);
```

Un pointeur est une variable d'un type particulier, qui, au lieu de contenir une valeur, contient l'adresse d'une valeur [5].

Appel de la fonction

Comme les paramètres ne sont plus des variables, mais des pointeurs, il ne faut plus transmettre la valeur qu'on veut utiliser, mais l'adresse de la variable [6]. L'adresse d'une variable s'obtient grâce à l'opérateur `&`, que nous avons déjà utilisé avec `scanf()`. On notera que dans ce cas, sans le savoir, nous avions déjà passé comme paramètre l'adresse de la variable destinée à recevoir la valeur entrée par l'utilisateur.

```
#include <stdio.h>

void prSwap(int * iV1,int * iV2);
int main()
  {
    int iN1    , iN2;

    iN1 =  5;
```

5. Cette notion ne peut pas être vue en détail ici. Elle fera l'objet d'un long développement dans le cours de Langage procédural.

6. On notera que, de cette manière, on transmet toujours une valeur, mais cette valeur est **l'adresse** d'une variable.

```
    iN2 =  6;
    prSwap(&iN1,&iN2);
    printf("%d\n",iN1);
    printf("%d\n",iN2);
    return 0;
  }
```

On voit donc, qu'à la différence de notre pseudo code, les arguments passés en C par adresse doivent s'écrire différemment [7].

Corps de la fonction

Il paraît plus difficile de rendre compte de l'écriture de la fonction. On notera que chacune des occurrences des paramètres subit une modification d'écriture. Par exemple, au lieu d'écrire

```
iTemp = iV1;
```

ce qui aurait pour conséquence d'essayer de placer l'adresse de la variable entière `iN1` dans la variable entière `iTemp`, on écrira :

```
iTemp = * iV1;
```

qui place la valeur pointée par `iV1` dans la variable `iTemp`. De même

```
* iV1 =  * iV2;
```

signifie : placer le contenu de l'adresse pointée par `iV2`, à l'adresse pointée par `iV1`.

```
void prSwap(int  * iV1,int  * iV2)
  {
    int iTemp;

    iTemp =  * iV1;
    * iV1 =  * iV2;
    * iV2 =  iTemp;
  }
```

9.3.4 Le passage des arguments tableaux.

Il est possible de passer des tableaux par valeur ou par adresse (ce que le langage Pascal implémente). Il est en pratique peu raisonnable de passer des tableaux par valeur, parce que cela consomme beaucoup de mémoire. En outre, de nombreuses procédures vont vouloir modifier les tableaux (insertion, suppression, tri), ce qui nécessite un passage par adresse.

On notera donc le parallélisme entre notre pseudo-code :

```
Prototype Procédure prAfficherTableau(
  Tableau Entier iT)
```

et le langage C :

```
void prAfficherTableau(int iT[]);
```

7. Cette transparence de l'appel existe également en Pascal et Delphi (où les paramètres passés par adresse sont caractérisés ainsi dans le seul en-tête de la fonction) et en C++, où l'on peut définir des paramètres passés par référence.

La paire de crochet qui suit la définition du tableau permet de donner une taille, mais celle-ci n'est en fait pas utilisée, puisque le paramètre est une adresse et qu'on réserve pas de place pour les données. On peut donc y placer une valeur quelconque, y compris 0.

```
void prAfficherTableau(int iT[0]);
void prAfficherTableau(int iT[1000]);
```

Pour ce qui concerne l'appel de la fonction, il est inutile d'utiliser l'opérateur `&`, puisque la variable `iTab` est déjà une adresse [8].

```
main()
  {
    int iC;
    int iTab1[10];
    int iTab2[10];

/* Initialisation du tableau 1 avec les Nombres De 1 à 10 */
    for(iC=0;iC<=9;iC++)
      {
        iTab1[iC] =  (iC + 1);
      }
   prAfficherTableau(iTab1);
/* Placement des carrés des  nombres dans le tableau  2*/
    for(iC=0;iC<=9;iC++)
      {
        iTab2[iC] =  ((iC + 1) * (iC + 1));
      }
    prAfficherTableau(iTab2);

  }
```

A l'intérieur de la fonction, on doit se référer au tableau sans utiliser l'opérateur `*`.

```
void prAfficherTableau(int iT[])
  {
    int iI;

    for(iI=0;iI<=9;iI++)
      {
        printf("%d",(iT[iI]));
      }
  }
```

Exercices

IX-1. Écrire une procédure `prReset` ou une fonction `fnReset` qui réinitialise une variable à zéro.

IX-2. Écrire une procédure ou une fonction qui rend positive toute variable entière qu'elle reçoit en argument.

8. Nous verrons plus loin que lorsque `iTab` désigne un tableau, les deux notations `iTab` et `&iTab` sont équivalentes et désignent l'adresse du premier élément du tableau. Par contre, lorsqu'il s'agit de pointeurs, `iTab` désigne l'adresse pointée et `&iTab`, l'adresse qui contient l'adresse. Comme on passe fréquemment de notations tableaux à des notations pointeurs, il convient d'éviter l'emploi superflu de `&` devant un nom de tableau.

IX-3. Écrire le **prototype** d'une sous-routine qui prend deux entiers comme arguments et renvoie leur moyenne.

IX-4. Écrire le **prototype** d'une sous-routine qui parcourt un tableau de 10 nombres, affiche, s'il y en a, tous ceux qui sont pairs et renvoie le nombre d'affichages réalisés.

IX-5. Remplir un tableau avec les N premiers nombres premiers. On utilisera le contenu du tableau pour déterminer ces nombres premiers (au lieu d'utiliser tous les nombres inférieurs à la racine).

Le jeu de Master Mind consiste à deviner un code de couleurs imaginé par un adversaire (joueur passif). Le code se compose à l'origine de quatre pastilles d'une couleur choisie parmi six couleurs, placées dans un certain ordre par le joueur passif. Le joueur actif propose à son adversaire une série de combinaisons de couleurs, que ce dernier analyse. Il répond en plaçant autant de fiches noires qu'il y a des pastilles bien placées et autant de fiches blanches qu'il y a de pastilles de la bonne couleur mais mal placées. Chaque pastille n'est évidemment comptabilisée qu'une seule fois, dans l'une des deux catégories. En variant les combinaisons et en tenant compte des informations des coups précédents, le joueur actif essaie de deviner le code en un nombre minimum de coups.

IX-6. (*Master Mind*) Prévoir la structure générale d'un programme remplaçant le joueur passif, c'est-à-dire :
- déclarer les variables globales ;
- envisager le déroulement du programme principal (le joueur actif ayant droit à plusieurs essais) ;
- prévoir les sous-programmes qui seront nécessaires à la réalisation du jeu ;
- écrire le programme principal.

IX-7. (*Master Mind*) Écrire une fonction ou procédure qui invente un code (pour simplifier, on proposera un code numérique en limitant les nombres à un ensemble précis (de 1 à 6). Écrire également une routine d'affichage du code. Variante : On pourra éventuellement autoriser ou non les répétitions de nombres dans le code.

IX-8. (*Master Mind*) Écrire une fonction ou procédure qui teste la réponse du joueur et renvoie le nombre de pastilles bien placées et le nombre de pastilles mal placées. Tester la sous-routine. Attention : quand une pastille de couleur a été identifiée comme bien placée, elle ne doit plus être identifiée comme mal placée si sa couleur figure plusieurs fois dans le code. Astuce : si on veut mémoriser toutes les phases de la partie, il est bon de déclarer une variable supplémentaire pour faire une copie de la réponse du joueur actif.

IX-9. Rédiger une sous-routine qui lit une valeur entière à l'écran telle qu'elle soit comprise entre 1 et un nombre donné. On supposera que la valeur se compose d'un seul chiffre. L'utilisateur pourra choisir la valeur qu'il veut ou manifester sa désapprobation par la touche Esc (code 27 ou encore `'\e'`). Dans ce cas, le sous-programme renvoie un code STOP, dans tous les autres cas il renvoie un code OK. Remarque : il faut trouver un moyen de renvoyer la valeur tapée. Au cas où l'utilisateur tape un chiffre non conforme ou un autre caractère, la routine se contente de faire un bip et attend une autre réponse.

IX-10. Un tableau global contient les cinq options d'un menu principal (une chaîne de caractères commençant par un chiffre). Écrire un sous-programme qui af-

fiche ces cinq options, centrées dans un écran vierge (voir question suivante pour un exemple d'utilisation).

IX-11. Au moyen des sous-programmes rédigés précédemment, rédiger un programme qui affiche le menu suivant :

```
1. Revoir les boucles
2. Revoir les tableaux
3. Revoir les enregistrements
4. Revoir les sous-programmes
5. Fin des révisions
```

On testera la réponse de l'utilisateur et on affichera un petit message permettant de voir que l'option a été bien comprise. Le programme bouclera jusqu'à de qu'on choisisse la dernière option (Sortie).

Les deux exercices suivant supposent la connaissance des opérateurs ++ et += du langage C. Si on ne connaît pas, il vaut mieux les éviter.

IX-12. Écrire une version «fonctionnelle» de l'opérateur ++. En effet l'opérateur ++ renvoie une valeur en C. Soient les exemples suivants :

```
iVar1 = 10;
iVar2 = 10;
printf("iVar vaut : %d\n",iVar1++); /* affiche 10 */
printf("iVar vaut : %d\n",++iVar2); /* affiche 11 */
```

Dans le premier cas, l'opérateur est dit de *post-incrémentation*, ce qui signifie qu'il incrémente la variable après l'avoir préparée pour le renvoi. Dans le second cas, on parle de *pré-incrémentation*, puisque la valeur renvoyée est celle de la variable déjà incrémentée.

IX-13. L'opérateur += ajoute la valeur située à sa droite à la variable contenue à sa gauche. Écrire une procédure qui réalise la même chose.