Les sous-programmes
Corrigé

Résumé

Ce document décrit les sous-programmes, outil essentiel pour structurer un programme car il permet au programmeur de définir ses propres instructions et opérateurs.

Table des matières

Liste des exercices

1 Introduction

Motivation : Un sous-programme permet au programmeur d'étendre le langage de programmation en définissant ses propres instructions et ses propres opérateurs qu'il pourra ensuite réutiliser. Il s'agit de regouper plusieurs instruction sous un nom commun.
Définition : Un sous-programme est un programme indépendant de son contexte (autonome), réutilisable dans plusieurs contextes.
Terminologie : Un *sous-programme* est un regroupement d'instructions qui peuvent ensuite être exécutées de manière atomique. Un sous programme permet ainsi de définir des instructions ou des expressions utilisateurs. Dans le premier cas on parle de procédure et dans le second cas de fonction.

Le terme « sous-programme » est un terme générique qui englobe à la fois les **procédures** et les **fonctions**. Les procédures et les fonctions peuvent être respectivement considérées comme des instructions et des opérateurs (ou expressions) définis par le programmeur qu'il pourra ensuite utiliser comme si elles faisaient partie du langage, donc comme des instructions élémentaires.

Les notions de procédure et de fonction sont très proches. La principale différence est qu'une procédure réalise un traitement et est assimilable à une instruction alors qu'une fonction (calcule et) renvoie une information et correspond donc à une expression. Un appel à une fonction peut apparaître là où une expression peut être utilisée (à droite de l'affectation, dans une expression, dans une condition, etc.)

Dans la section 2, nous introduisons les notions de procédure et de fonction en nous appuyant sur des exemples concrets. Dans les sections suivantes, nous définissons plus précisément ce que sont une procédure et une fonction.

Exercice 1 : Sous-programmes déjà rencontrés
Donner des exemples de sous-programmes que vous avez déjà rencontré. Préciser si ce sont des procédures ou des fonctions.

Solution : Lire et Écrire sont des exemples de procédure.
C system, printf et scanf sont des exemples de procédure.
En fait printf et scanf sont des fonctions mais nous n'avons jamais exploité la valeur retournée.
abs est un exemple de fonction
C abs, sqrt, rand, etc. sont des exemples de fonctions qui permettent respectivement d'obtenir la valeur absolue, la racine carrée ou un nombre aléatoire.
Conclusion : On a déjà rencontré et utilisé des sous-programmes.

2 Exemple introductif

2.1 Définition de nos premiers sous-programmes

Reprenons l'exemple du robot. Dans les niveaux de raffinage intermédiaires nous avons identifié des étapes telles que « tourner à gauche », « tourner à droite », « progresser de n cases », n étant un entier positif.

Puisque ces étapes ne sont pas comprises par le robot, il était nécessaire dans l'algorithme final de remplacer chacune de leurs occurrences par les instructions élémentaires correspondantes. Ceci est fastidieux (on répète les mêmes instructions), conduit à un algorithme long et difficile à comprendre (il faut absolument mettre les commentaires correspondant aux traces du raffinage).

Les procédures (et fonctions) proposent une solution élégante et économique. Nous allons définir ces trois étapes comme étant des procédures (des instructions programmeur) en expliquant une fois pour toutes au processeur comment les comprendre (les interpréter, les exécuter). Elles constituent donc de nouvelles instructions que nous pourrons ensuite utiliser pour guider le robot.

Comment peut-on définir de telles opérations ?

1. On commence par préciser ce que doit faire le sous-programme en définissant son interface qui comprend, entre autre, le nom du sous-programme, son type (ici procédure) et un commentaire qui décrit son objectif informellement.
2. On décrit comment réaliser le traitement décrit dans l'interface. Ceci correspond à un algorithme obtenu de la même manière que pour un programme dont le R0 serait l'interface du sous-programme.

```
1  Procédure tourner_droite Est
2          -- faire tourner le robot à droite
3      Début
4          PIVOTER
5      Fin
6
7  Procédure tourner_gauche Est
8          -- faire tourner le robot à gauche
9      Début
10         PIVOTER
11         PIVOTER
12         PIVOTER
13     Fin
```

Remarque : Si on suppose qu'on a la notion de compteur (entier, incrémentation et test) alors on peut écrire la procédure de cette manière.

```
1  Procédure tourner_gauche Est
2          -- faire tourner le robot à gauche
3      Variable
4          i: Entier       -- compter le nombre de fois où l'on tourne
5      Début
6          Pour i ← 1 JusquÀ i = 3 Faire
7              PIVOTER
8          FinPour
```

```
9       Fin
```

Dans ce cas, pour réaliser la procédure, on utilise une variable i. Cette variable est dite **variable locale** à la procédure car elle est seulement utilisée dans le traitement réalisé par la procédure. Le programme appelant cette procédure ne connait pas (et n'a pas à connaître) l'existence de cette variable. Ainsi :

- sa **durée de vie** est celle du sous-programme (elle est créée quand le sous programme est appelé et est détruite lorsqu'il se termine : c'est l'**allocation statique** de mémoire réalisée automatiquement par le compilateur).
- sa **portée** est seulement les instructions du sous-programme. La variable n'est visible, et donc ne peut être utilisée, que dans le sous-programme.

De la même manière on peut définir la procédure progresser.

```
1  Procédure progresser(n : in Entier)
2          -- Faire avancer le robot de n cases dans la direction courante.
3          --
4          -- Nécessite :
5          --     n > 0    -- n doit être un entier strictement positif
6      Variable
7          i : Entier
8      Début
9          Pour i ← 1 JusquÀ i = n Faire
10             AVANCER
11         FinPour
12     Fin
```

La procédure `progresser` ainsi définie possède un paramètre formel nommé `n`. Chaque paramètre formel correspond à une information échangée entre le programme appelant et le sous-programme. Il est caractérisé par un nom, un type et un mode de passage qui décrit dans quels sens la communication à lieu (section 6.1). Ici le mode de passage est **`in`** (en entrée), ce qui signifie que c'est au programme appelant de fournir la valeur de `n` et que le sous-programme n'a pas le droit d'en changer la valeur. La valeur fournie par le programme appelant est appelée le *paramètre effectif*.

Ainsi, lorsque l'on écrit un sous-programme, on raisonne sur la valeur des paramètres formels, sachant que les valeurs effectives (réelles) ne sont pas encores connues. Dans le cas, présent on pourrait formuler le sous-programme de la manière suivante : « étant donné un entier n, faire avancer le robot de n cases dans la direction courante ».

La clause `Nécessite` qui fait partie du commentaire indique les conditions que doivent vérifier les paramètres effectifs fournis par le programme appelant. Ici, il faut nécessairement donner une valeur strictement positive pour le paramètre formel `n`. Si ceci n'est pas respecté, c'est le programmeur du programme appelant qui est en faute et non l'auteur du sous-programme `progresser`.

2.2 Écriture de l'algorithme

Maintenant que les sous-programmes précedents sont écrits, l'algorithme qui consiste à déplacer le robot de la salle de cours vers le secrétariat s'écrit ainsi.

```
1  Algorithme Guider_robot
2
3      -- Guider le robot de la salle de cours vers le secrétariat
4
5      -- la définition des sous-programmes est omise.
6
7  Début
8      -- Sortir de la salle de cours
9      progresser(2)
10     tourner_gauche
11     progresser(3)
12
13     -- Longer le couloir (jusqu'à la porte du vestibule)
14     tourner_droite
15     progresser(9)
16
17     -- Traverser le vestibule (et entrer dans le secrétariat)
18     tourner_droite
19     progresser(3)
20     tourner_droite
21     progresser(1)
22     tourner_gauche
23     progresser(1)
24 Fin.
```

2.3 Déplacer un robot avec des sous-programmes

Voici l'algorithme complet du déplacement du robot faisant apparaître les sous-programmes définis ci-dessus et le programme principal.

```
1  Algorithme Guider_robot
2
3      -- Guider le robot de la salle de cours vers le secrétariat
4      -- Solution algorithmique utilisant les sous-programmes.
5
6  Procédure tourner_droite Est
7          -- Faire tourner le robot à droite
8      Début
9          PIVOTER
10     Fin
11
12 Procédure tourner_gauche Est
13         -- Faire tourner le robot à gauche
14     Début
15         PIVOTER
16         PIVOTER
17         PIVOTER
18     Fin
19
20 Procédure progresser(n: in Entier) Est
21         -- Faire avancer le robot de n cases dans la direction courante.
```

```
22          --
23          -- Nécessite :
24          --     n > 0    -- n doit être un entier strictement positif
25      Variables
26          i: Entier       -- pour compter le nombre de fois où le robot avance
27      Début
28          Pour i ← 1 JusquÀ i = N Faire
29              AVANCER
30          FinPour
31      Fin
32
33  ----------------  L'algorithme principal  --------------------
34
35  Début
36      -- Sortir de la salle de cours
37      progresser(2)
38      tourner_gauche
39      progresser(3)
40
41      -- Longer le couloir (jusqu'à la porte du vestibule)
42      tourner_droite
43      progresser(9)
44
45      -- Traverser le vestibule (et entrer dans le secrétariat)
46      tourner_droite
47      progresser(3)
48      tourner_droite
49      progresser(1)
50      tourner_gauche
51      progresser(1)
52  Fin.
53
54  -- Remarque : on aurait pu remplacer progresser(1) par AVANCER.  Ceci améliore
55  -- l'efficacité (on évite un appel à un sous-programme) mais nuit peut-être à
56  -- l'homogénéité de l'algorithme et donc à sa clarté.  Un autre avantage à utiliser
57  -- progresser(1) est que l'algorithme dépend alors seulement des procédures définies e
58  -- pas directement des actions élémentaires.  Il est donc plus stable à un changement
59  -- les instructions élémentaires du robot.
```

2.4 Exécution d'un sous-programme

Cet aspect est abordé en détail à la section 7. Nous nous contentons ici d'en esquisser le principe.

L'appel d'un sous-programme correspond à un déroutement de l'exécution du programme. L'instruction suivante à exécuter est alors la première instruction du sous-programme. Il s'exécute comme un sous-programme mais ne peut manipuler que les données qui correspondent à ses paramètres formels (initialisés à partir des paramètres effectifs) et ses variables locales. Lors de la fin de l'exécution du sous-programme, l'exécution du programme appelant reprend. Il est donc nécessaire de savoir qu'elle est l'instruction du programme appelant qui a provoqué l'ap-

pel. Cette information doit être conservé lors de l'appel du sous-programme. Elle est stockée dans la pile d'exécution avec les paramètres et variables locales du sous-programme, l'ensemble constituant le bloc d'activation.

2.5 Procédures utilisant d'autres procédures

Le corps d'une procédure est composé d'instructions qui peuvent être élémentaires ou définies par le programmeur (sous-programmes).

Ainsi, on peut définir une procédure `reculer` de la manière suivante :

```
1  Procédure reculer(n: in Entier) Est     -- n : paramètre formel de reculer
2          -- faire avancer le robot de n cases vers l'arrière, n étant
3          -- strictement positif.  La direction du robot n'est pas modifiée.
4          --
5          -- Principe : avancer dans la direction opposée.
6      Début
7          faire_demi_tour
8          progresser(n)           -- n est le paramètre effectif de progresser
9          faire_demi_tour
10     Fin
```

Il faut bien évidemment avoir défini **au préalable** la procédure `faire_demi_tour`. Une définition possible est donnée ci-dessous.

```
1  Procédure faire_demi_tour Est
2          -- faire faire demi tour au robot
3      Début
4          PIVOTER
5          PIVOTER
6      Fin
```

En effet, on ne doit pas avoir de référence en avant car elles nuisent à la bonne compréhension de l'algorithme : il faut toujours avoir défini les entités utilisées. Une deuxième raison (certainement la plus importante) est que ceci facilite l'écriture des compilateurs.

2.6 Exemples de fonctions

Une fonction est une expression définie par le programmeur. Elle renvoie donc une valeur. Aussi, son identifiant ne sera pas un verbe mais un nom commun ou un adjectif si la valeur est booléenne (`somme`, `max`, `bissextile`, `égales`, `est_opération`, etc.) qui caractérise l'information renvoyée. Par exemple, nous pouvons appeler `carré` la fonction qui associe à un réel son carré.

```
1  Fonction carré(x : in Réel) : Réel Est
2          -- le carré d'un nombre réel
3      Début
4          Résultat ← x * x
5      Fin
```

Comme une procédure, une fonction est définie par un identifiant (son nom), un commentaire expliquant ce qu'elle renvoie, des paramètres formels et un corps qui calcule le résultat à ren-

voyer. Lorsqu'on définit une fonction il faut en plus préciser le type de l'information renvoyée. C'est le **type de retour** de la fonction.

En algorithmique, nous considérerons que pour chaque fonction, une variable de même type que le type de retour de la fonction est définie. Le nom de cette variable est **`Résultat`**. Elle peut être utilisée pour réaliser les calculs. La valeur renvoyée à la fin de l'exécution de la fonction est la valeur de la variable **`Résultat`**.

Remarque : Une fonction n'est pas forcément une fonction mathématique. Exemple : la fonction qui indique si un caractère correspond à une opération (`+ - * /`), la fonction qui indique si l'utilisateur répond oui ou non...

2.7 Nombre de paramètres formels

Un sous-programme peut ne pas avoir de paramètres (`tourner_droite` et `tourner_gauche`), avoir un paramètre (`progresser` prend en paramètre un nombre de cases), avoir plusieurs paramètres (`max` a deux paramètres correspondant à ses deux opérandes). Les paramètres formels sont séparés par des points virgules.

```
1 Fonction max(a : in Entier ; b : in Entier) : Entier Est
2         -- le plus grand de deux nombres entiers
3     Début
4         Si a > b Alors
5             Résultat ← a
6         Sinon
7             Résultat ← b
8         FinSi
9     Fin
```

Il faut éviter d'avoir trop de paramètres car il devient difficile de savoir à quoi ils correspondent. D'ailleurs certains langages limitent le nombre de paramètres.

Remarque : Pour diminuer le nombre de paramètres, on peut utiliser un type enregistrement. Ainsi la comparaison de deux dates (l'égalité par exemple) ne prend pas 6 paramètres (j1, m1, a1, j2, m2, a2) mais deux (d1 et d2) qui ont chacun trois champs correspondant au jour, au mois et à l'année.

3 Intérêt des sous-programmes

L'utilisation des sous-programmes (procédures et fonctions) est intéressante à plus d'un titre. Nous en précisons quelques uns ci-dessous.

Structuration de l'algorithme En l'absence de procédure et de fonction, l'algorithme est monolithique : les instructions sont toutes écrites les unes à la suite des autres et ce ne sont que les commentaires, traces du raffinages, qui le structurent et permettent de le comprendre. Chaque procédure ou fonction correspond à une sous-partie (un sous-programme) de l'algorithme initial. Il permet donc de le découper en « morceaux » et donc de le structurer en entités relativement indépendantes. Pour s'en convaincre, il suffit de regarder l'algorithme 2.3.

Compréhensibilité de l'algorithme La structuration permet d'améliorer la compréhension de l'algorithme, car il est découpé en procédures et fonctions de taille réduite qui sont donc plus facile à comprendre. Les algorithmes ne font donc plus nécessairement appel aux instructions élémentaires mais à des instructions de plus haut niveau les procédures et les fonctions qui, de part leur nom significatif, en améliore la compréhension.

Factorisation du code Les sous-programmes permettent de factoriser des parties de l'algorithme. Par exemple, `tourner_gauche` remplace les trois occurrences successives de `PIVOTER` dont l'objectif était de faire tourner à gauche le robot. La manière de faire tourner à gauche le robot n'est donnée qu'à un seul endroit.

Mise au point Dès qu'un sous-programme est écrit, il peut (et doit) être testé. Ainsi, le programme est testé petits bouts par petits bouts (sous-programme après sous-programme). Les erreurs et surtout leur origine sont alors beaucoup plus facilement identifiées que si l'ensemble du l'algorithme était testé d'un seul coup.

Ceci est également vrai lors de la modification de sous-programmes (maintenance corrective ou évolutive).

Amélioration de la maintenance Comme la compréhension du programme, la maintenance est automatiquement amélioré, car il sera plus facile d'identifier les parties de l'algorithme à modifier et d'en évaluer l'impact. L'idéal est bien entendu que la modification puisse être limitée à un petit nombre de sous-programmes.

La factorisation améliore également notablement la maintenance. En effet, si une modification est faite dans un sous-programme, elle sera automatiquement prise en compte dans toutes les autres parties qui utilisent ce sous-programme. Si le code avait été dupliqué, il aurait fallu le changer à tous les endroits.

Dans le même ordre d'idée, les sous-programmes permettent au programmeur de définir ses propres instructions et expressions. Ainsi, s'appuyant sur ses instructions, il peut écrire des algorithmes indépendants des instructions élémentaires du processeur.

Prenons un exemple concret, les déplacements d'un robot. Tout déplacement du robot peut être fait en utilisant tourner à droite, tourner à gauche et progresser de n cases. Si les actions élémentaires du robot changent (par exemple PIVOTER ne le fait plus tourner d'$1/4$ de tour dans le sens des aiguille d'une montre mais d'$1/8$ de tour dans le sens inverse des aiguilles d'une montre), il suffit de modifier le corps (l'implantation) de ces trois procédures sans modifier le reste de l'algorithme (le déplacement de la salle de cours vers le secrétariat reste inchangé).

Réutilisation Un sous-programme décrit un traitement ou le calcul d'une information qui peut ensuite être utilisé autant de fois que nécessaire dans l'algorithme en cours de construction mais également dans les algorithmes futurs. Par exemple, à chaque fois qu'on doit faire tourner le robot à gauche, il suffit d'appeler la procédure `tourner_gauche`.

4 Portée des variables

4.1 Portée des variables

On appelle *portée d'une variable* la partie du programme où on peut y faire référence.

La portée d'une variable (la zone du programme dans laquelle elle peut être utilsiée, on dit aussi *visible*) commence avec sa déclaration et se termine avec l'entité (programme, module, procédure, etc.) dans laquelle elle a été déclarée.

Par exemple, une variable déclarée dans un sous-programme comme le `i` de progresser ne peut être utilisée que dans le sous-programme progresser.

4.2 Variable globale

On appelle variable globale une variable qui est visible (et donc utilisable et modifiable) par plusieurs sous-programmes (le programme principal étant considéré comme un sous-programme).

Les variables globales sont à éviter car elles correspondent à des entrées et/ou des sorties des sous-programmes qui ne sont pas identifiées dans leurs interfaces. Elles rendent alors les programmes plus difficiles à comprendre et à maintenir (effets de bord).

4.3 Variable locale

Une variable locale est une variable déclarée à l'intérieur d'un sous-programme. Sa portée est donc ce sous-programme.

```
1  Algorithme ExemplePortée
2
3  Variable
4      x: Entier    -- x est une variable globale de type Entier
5
6  Procédure proc(n: in Réel) Est
7      Variable
8          i: Réel  -- la variable locale masque la variable globale i
9      Début
10         i ← 5.0 -- OK
11         x ← 3           -- OK
12         l ← '0' -- ERREUR : l n'est pas visible
13     Fin
14
15 Variable
16     l: Caractère        -- variable « locale » au programme principal
17
18 Début
19     n ← 3       -- ERREUR : n n'est pas visible
20     i ← 1.2     -- ERREUR : i n'est pas visible
21     x ← 1       -- OK   (x est la variable globale)
22     l ← 'A'     -- OK
23 Fin
```

Dans l'exemple ci-dessus, la portée de la variable x englobe la procédure proc et la programme principal. Il s'agit donc d'une variable globale.

En revanche la portée de l n'englobe que le programme principal, il s'agit donc pas d'une variable globale mais d'une variable « locale » au programme principal. Comme est est déclarée après la procédure proc, elle n'est donc pas visible dans ce sous-programe ce qui explique l'erreur lors de l'affectation l ←'0'.

Le paramètre formel n et la variable locale n ont pour portée la procédure proc dans laquelle ils sont déclarés. En conséquence, ils ne peuvent pas être utilisés dans le programme principal, d'où les deux erreurs.

C La portée en C fonctionne comme en algorithmique. Notons qu'à l'exception des variables globales, toutes les autres variables sont déclarées en C dans des accolades (elles doivent même être en début d'accolades avant les instructions). Leur portée est donc délimitée par ces accolades.

4.4 Masquage

Pour nommer une variable locale, on peut utiliser un nom déjà utilisé pour une variable globale. La conséquence est que la nouvelle déclaration masque la précédente. Ainsi, la variable globale ne peut pas être utilisée. Son nom désignera la variable locale.

```
1  Algorithme ExempleMasquage
2
3  Variable
4      i: Réel      -- i est une variable globale de type Réel
5      n: Caractère -- n est une variable globale de type Caractère
6      x: Entier    -- x est une variable globale de type Entier
7
8  Procédure proc(n: in Réel) Est
9          -- le paramètre formel n masque la variable globale n
10     Variable
11         i: Réel  -- la variable locale masque la variable globale i
12         j: Réel  -- variable locale de type Réel
13     Début
14         i ← 5.0 -- OK (i est la variable locale)
15         j ← 5.0 -- OK (j est la variable locale)
16         n ← 1           -- OK (n est le paramètre formel)
17         x ← 3.14        -- OK (x est la variable globale)
18     Fin
19
20 Début
21     n ← 'V'     -- OK   (n est la variable globale)
22     n ← 1.5     -- ERREUR : n est de type Caractère (var. globale)
23     j ← 1.2     -- ERREUR : j n'est pas visible
24     x ← 1.3     -- OK   (x est la variable globale)
25 Fin
```

C Le masquage en C peut également avoir lieu dans un sous-bloc.

```
1  int main()
```

```
2 {
3     double x;
4     {
5         char x;
6             // ce x masque le x de type réel ci-dessus.
7     }
8 }
```

Il faut absolument éviter ce genre de masquage car il est source d'erreurs difficile à identifier.

4.5 Durée de vie d'une variable

La durée de vie est le temps pendant lequel la variable existe, c'est-à-dire pendant lequel elle peut être utilisée. Elle commence par la réservation de sa place en mémoire et s'arrête lorsque cette place est libérée.

Une variable a la même durée de vie que l'entité dans laquelle elle a été déclarée. Ainsi, une variable du programme principal a la même durée que le programme principal : elle est créée quand le programme est lancé et est « détruite » quand le programme s'arrête. Une variable locale à un sous-programme est créée quand le sous-programme est appelé. Quand le sous-programme est terminé, elle est détruire et la place qu'elle occupait est récupérée.

4.6 Illustration

Illustrons ceci par un exemple. Cet exemple montrera également le risque à utiliser des variables globales. Une **variable globale** est une variable qui peut être utilisée (lue ou modifiée) par plusieurs entités (procédure, fonction, programme principal...).

On souhaite écrire un algorithme pour la table traçante qui dessine un carré dont le côté a une longueur de 4 cases. On propose l'algorithme de la figure 1. Exécuter l'algorithme et indiquer quel est le dessin qui apparaît sur la feuille. (On suppose que la table fournie les opérations élémentaires pour écrire une répétition.)

Le problème vient de ce que dans `progresser`, nous avons oublié de déclarer la variable locale `i`. C'est donc la variable globale (celle du programme principal) de même nom qui est utilisée. Après l'exécution de la procédure `progresser`, la valeur de la variable `i` du programme principal ne vaut plus `1` mais `5` ! Le **`Pour`** s'arrête donc (`i` vaut `6` après l'incrémentation) !

Attention : En algorithmique, nous déclarons les variables du programme principal juste devant le **`Début`** du programme principal. Ces variables ne sont donc pas connues des procédures mais seulement du programme principal.

C Nous n'utiliserons pas de variables globales en C (sauf cas particulier qui devra être justifié).

D'autre part, les variables du programme principal ne risque pas d'être considérées comme globales car elles sont déclarées comme variables locales de la fonction `main()`.

Listing 1 – Algorithme pour tracer un carré (avec erreur)

```
1  Algorithme Tracer_carre
2
3      -- Algorithme pour tracer un carré.
4      -- Attention cet algorithme contient une erreur
5
6  Variables
7      i : Entier  -- pour déplacer le robot sur chaque côté du carré
8
9      -- i est une variable globale car elle est potentiellement accessible (visible) de
10     -- deux procédures qui suivent et du programme principal
11
12 Procédure tourner_gauche Est
13         -- Faire tourner le robot à gauche
14     Variables
15         i : Entier      -- pour compter le nombre de fois où il faut tourner
16     Début
17         Pour i ← 1 JusquÀ i = 3 Faire
18             TOURNER
19         FinPour
20     Fin
21
22 Procédure progresser(n : in Entier) Est
23         -- Faire avancer le robot de n cases dans la direction courante.
24         -- Nécessite :
25         --    N > 0 : N doit être un entier strictement positif
26     Début
27         Pour i ← 1 JusquÀ i = N Faire
28             AVANCER
29         FinPour
30     Fin
31
32 ---------- programme principal ----------
33
34 Début
35     BAISSER
36     Pour i ← 1 JusquÀ i = 4 Faire
37         tourner_gauche
38         progresser(4)
39     FinPour
40     LEVER
41 Fin.
```

5 Structure d'un algorithme

Maintenant que nous avons vu les procédures et les fonctions, nous pouvons donner la structure générale d'un algorithme et l'enchaînement des différentes rubriques qui le composent.

```
1  Algorithme <Nom_algo>
2
3  Constantes      -- Définition des constantes
4      <NOM_CONSTANTE> = <valeur_constante>        -- ce qu'elle représente
5      ...
6
7  Types           -- Définition des types utilisés dans l'algorithme
8      -- explication du type
9      <Type> = <définition du type>
10     ....
11
12 Variable
13     -- Variables globales car potentiellement accessibles par les sous-programmes
14     <nom_variable> : <Type>              -- ce que représente la variable
15     ...
16
17 -- Définition des sous-programmes dans n'importe quel ordre pourvu qu'il n'y
18 -- ait pas de déclarations en avant
19
20 Procédure <verbe> <paramètres_formels> Est
21         -- description de ce que fait la procédure
22     Variable            -- variables locales à la procédure
23         <nom_var_locale> : <Type>        -- ce que représente la variable
24     Début
25         <instruction>           -- élémentaire ou sous-programme
26         ...
27     Fin
28
29 Fonction <nom_fonction> <paramètres_formels> : <Type retour> Est
30         -- description de ce que calcule la fonction
31     Variable            -- variables locales à la fonction
32         ...
33     Début
34         ...
35         Résultat ← ...           -- variable prédéfinie pour toute fonction
36     Fin
37
38 -----------  programme principal -----------
39
40 Variable        -- variables du programme principal
41     <nom_variable> : <Type>      -- ce qu'elle représente
42 Début
43     <instruction>
44     ...
45 Fin.
```

6 Communication entre programmes et sous-programmes

Les sous-programmes communiquent avec le programme principal. La première communication est l'appel de sous-programme lui-même (par exemple `tourner_gauche` ou `tourner_gauche`). Cependant, des informations doivent être échangées entre les sous-programmes et le programme (par pour faire progresser le robot, il faut préciser de combien de cases on peut qu'il avance). Il existe deux façons de transmettre cet information :

- les **paramètres des sous-programmes** : c'est la « bonne » façon de faire. Le programme fournit explicitement les données au sous-programme lors de son appel. Le sous-programme est alors assimilable à une boîte noire dont l'effet ne dépend que de ses paramètres. Ceci favorise la compréhension, l'évolution, la réutilisation, etc.
- les **variables globales** : c'est la mauvaise façon de faire. Les variables globales sont des variables qui sont accessibles à la fois du programme et du sous-programme (variables partagées). Ainsi le programme peut modifier la valeur d'une variable globale, lancer le sous-programme qui utilisera cette variable globale et, enfin, lire le contenu de la variable globale.

  Cette façon de faire est très mauvaise car :
  - *elle limite la réutilisation* : la fonction est très dépendante de son contexte (les variables globales).
  - *la compréhension du programme est difficile* : le passage des paramètres, l'appel à la fonction et la récupération des résultats se fait en plusieurs lignes pas forcément proches (consécutives).
  - *le programme est difficile à maintenir* car le résultat d'une fonction ne dépend plus seulement de ses paramètres mais également des valeurs des variables globales qu'elle utilise. Ce sont des entrées déguisés, masquées. Modifier à un endroit la valeur d'une variable globale peut modifier le comportement de plusieurs appels au sous-programme.

6.1 Les paramètres

Paramètres formels : Lors de la déclaration d'un sous-programme on ne connait pas encore les données effectives. De plus la généralité d'un sous-programme est de pouvoir travailler sur un ensemble de données qui ont la même forme, la même structure ou les mêmes propriétés. Ces données sont alors référencées sous la forme de paramètres formels.

Une façon de lire l'entête d'un sous-programme est de dire : étant donnés les paramètres...

Paramètres effectifs : Les paramètres effectifs sont les données qui sont fournies lors de l'appel d'un sous-programme et qui donc correspondent aux paramètres formels du sous-programme.

Les paramètres (formels, puis effectifs) représentent de l'infomation qui est échangée entre le programme et le sous-programme. Pour préciser les règles de cette communication, nous définissons des modes de passages de ces paramètres. Un mode de passage définit comment sont transmises les données au sous-programme et les effets des actions que le sous-programmes peut faire dessus.

Jusqu'à présent, nous n'avons vu qu'un seul mode de passage des paramètres : le mode en entrée noté **`in`**. Dans ce paragraphe, nous commençons par justifier la nécessité d'avoir plusieurs

modes de passage des paramètres et nous présentons ensuite ces différents modes. Nous terminons en décrivant les modes de passage de paramètres en Pascal.

6.1.1 Pourquoi des modes de passage ?

Lors de la présentation du langage langage algorithmique, nous avons vu des opérations élémentaires d'entrée/sortie qui fonctionnent sur les types élémentaires. En fait, ces opérations élémentaires **Lire** et **Écrire** sont tout simplement des procédures prédéfinies par le langage et stockées dans une bibliothèque standard[1]. D'après ce que nous savons des procédures, la définition de **Lire** et **Écrire** pour les entiers pourrait être la suivante :

```
1  Procédure Écrire(n : in Entier) Est
2          -- Afficher à l'écran l'entier n dans sa représentation en base 10
3      Début
4          -- Les instructions nécessaires
5      Fin
6
7  Procédure Lire(n : out Entier) Est
8          -- Lire au clavier les caractères qui doivent correspondre à la
9          -- représentation en base 10 d'un entier, reconstituer l'entier et le
10         -- ranger dans la variable n pour le transmettre au programme appelant.
11     Début
12         -- Les instructions nécessaires
13     Fin
```

Écrire reçoit un entier du programme appelant et l'affiche à l'écran. La valeur de l'entier est une information en entrée du sous-programme : elle est fournie par le programme appelant et utilisée par le sous-programme. La paramètre est dit en entrée (pour le sous-programme).

À l'inverse, **Lire** reconstitue un entier à partir des caractères tapés au clavier et transmet l'entier qu'ils représentent au programme appelant. L'information, la valeur de l'entier lu, est donc fournie par le sous-programme au programme appelant.

Si on essaie d'exécuter la procédure **Lire**, on se rend alors compte que la valeur lue ne peut être transmise de la sorte au programme appelant. La valeur de l'entier lu au clavier est en effet rangée dans la zone réservée pour stocker le paramètre formel n de la procédure. Or, cette zone n'existe plus après l'exécution de la procédure. Le paramètre est dit en sortie (pour le sous-programme).

6.1.2 Quels modes de passage des paramètres ?

Lorsque le programmeur définit une procédure, c'est pour pouvoir ensuite l'utiliser comme une instruction élémentaire.

Nécessairement, il y a un échange de données entre le sous-programme et le programme appelant (programme principal ou sous-programme qui contient un appel au sous-programme considéré). En effet, les calculs effectués par le sous-programme doivent être faits à partir des données du programme appelant. Inversement, des informations calculées dans le sous-programme

[1] Une bibliothèque regroupe un ensemble de sous-programmes.

pourront ensuite être exploitées par le programme appelant. Pour cette raison, on va caractériser les échanges d'informations en identifiant dans quel sens l'information circule.

Prenons quelques exemples. Lorsque le programmeur utilise la procédure **Écrire** sur les entiers, il lui donne la valeur de l'entier à afficher. On parle alors de paramètre en entrée de la procédure. En revanche, lorsque le programmeur utilise la procédure **Lire** sur les entiers, c'est pour que la procédure lui fournisse une valeur pour la variable qu'il aura passée en paramètre. On parle alors de paramètre en sortie de la procédure. Enfin, une donnée peut être à la fois en entrée et en sortie. On parle alors de paramètre en entrée/sortie.

Pour illustrer l'échange de données entre le sous-programme et le programme appelant, on peut faire une analogie entre un client et un fournisseur (figure 1). Le fournisseur est le sous-programme : il réalise un traitement à la demande du client (le programme appelant). Prenons un exemple concret : vous êtes le client, le fournisseur est un bijoutier. Pour acheter une montre, vous indiquez vos goûts et votre budget (entrées) au bijoutier et il vous vend une montre (sortie). La montre tombe en panne. Vous souhaitez la faire réparer. Vous indiquez au bijoutier les symptômes de la panne (entrée) et vous lui donnez la montre qu'il vous rend ensuite réparée (entrée/sortie).

En algorithmique, nous utiliserons donc ces trois modes : en *entrée*, en *sortie* et en *entrée/-sortie*. Nous les détaillons dans les paragraphes suivants.

6.1.3 Paramètres en entrée

Définition : Un paramètre en entrée correspond à une donnée qui est fournie par le programme appelant au sous-programme. Le sous-programme n'a pas le droit de changer la valeur du paramètre formel, il ne peut qu'accéder à sa valeur.

Paramètre effectif : Le paramètre effectif correspondant est n'importe quelle expression dont le type est identique au type du paramètre formel.

Notation : Un paramètre formel en entrée est déclaré en utilisant le mot-clé **in** :

```
1 <nom> : in <Type>        -- paramètre formel correspondant à une donnée en entrée
```

Exemple : La procédure `afficher_mois` affiche le nom correspondant au mois passé en paramètre. L'appelant donne le mois mais n'attend pas de modification en retour. Le mois est donc un paramètre en entrée. (Mois est un type énuméré.)

```
1  Procédure afficher_mois(un_mois : in Mois) Est
2          -- afficher le nom du mois
3      Début
4          Selon un_mois Dans
5              janvier     : Écrire("janvier")
6              février     : Écrire("février")
7              mars        : Écrire("mars")
8              avril       : Écrire("avril")
9              mai         : Écrire("mai")
10             juin        : Écrire("juin")
```

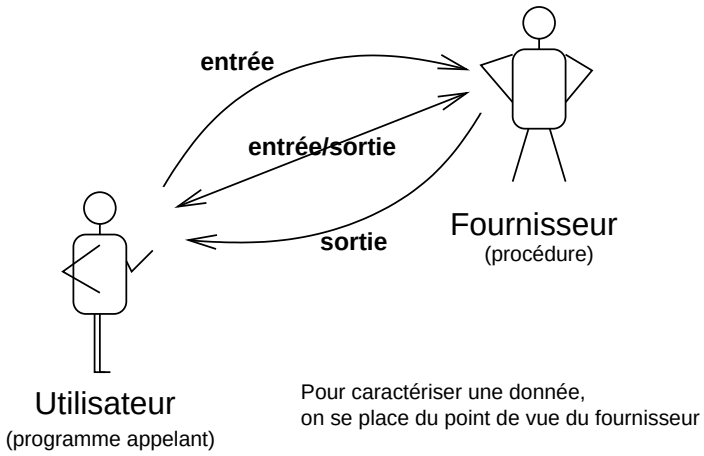

FIG. 1 – Les modes de passage des paramètres

```
11              juillet     : Écrire("juillet")
12              août        : Écrire("août")
13              septembre   : Écrire("septembre")
14              octobre     : Écrire("octobre")
15              novembre    : Écrire("novembre")
16              décembre    : Écrire("décembre")
17          FinCas
18      Fin
```

6.1.4 Paramètres en sortie

Définition : Un paramètre en sortie correspond à une information qui est demandée au sous-programme par le programme appelant.

En conséquence, le sous-programme ne doit pas dépendre (et donc utiliser) la valeur du paramètre formel. De plus, le sous-programme doit impérativement donner une valeur au paramètre effectif.

Paramètre effectif : Le paramètre effectif correspondant est nécessairement une variable (ou assimilé : élément d'un tableau ou champ d'un enregistrement) dont le type est identique au type du paramètre formel.

Notation : Un paramètre formel en sortie est déclaré en utilisant le mot-clé **out** :

```
1  <nom> : out <Type>      -- paramètre formel correspondant à une donnée en sortie
```

Exemple : La procédure saisir_mois lit au clavier un mois. Le paramètre mois est donc en sortie. L'utilisateur saisit le mois par son numéro dans l'année. (Mois est un type énuméré.)

```
1  Procédure saisir_mois(un_mois : out Mois) Est
2          -- saisir un mois au clavier.  L'utilisateur entre le mois sous la
3          -- forme d'un entier compris entre 1 et 12.
4      Variables
```

```
5           numéro : Entier -- numéro du mois saisi au clavier
6       Début
7           -- Saisir le numéro du mois
8           Lire(numéro);
9
10          -- Traiter les erreurs de saisie
11          TantQue(numéro < 1 Ou numéro > 12) Faire        -- Saisie incorrecte
12              -- Signaler erreur
13              Écrire("Erreur de saisie.")
14              -- Saisir un nouveau numéro
15              Écrire("Entrez un entier compris entre 1 (janvier) et 12 (décembre) : ")
16              Lire(numéro)
17          FinTQ
18
19          -- Convertir le numéro en mois
20          Selon numéro Dans
21                  1 : un_mois ← janvier
22                  2 : un_mois ← février
23                  3 : un_mois ← mars
24                  4 : un_mois ← avril
25                  5 : un_mois ← mai
26                  6 : un_mois ← juin
27                  7 : un_mois ← juillet
28                  8 : un_mois ← août
29                  9 : un_mois ← septembre
30              10 : un_mois ← octobre
31              11 : un_mois ← novembre
32              12 : un_mois ← décembre
33          FinCas
34      Fin
```

Attention : Ce qui peut choquer est que, lors d'un appel à la procédure, il faut fournir un paramètre. On pourrait donc se dire que c'est aussi une entrée. Ce n'est pas le cas car, la valeur de cette variable n'est pas utilisée par la procédure. La variable est simplement un contenant qui sert à transmettre une information de la procédure vers l'appelant. C'est comme un client qui va acheter du vin avec un cubitainer. Le cubitainer n'est pas une entrée pour le détaillant, mais seulement le moyen de transmettre le vin au client. Le cubitainer est la variable, le vin (son contenu) est le paramètre en sortie.

6.1.5 Paramètres en entrée/sortie

Définition : Un paramètre en entrée/sortie correspond à une information à la fois en entrée et en sortie. La procédure doit donc tenir compte de la valeur du paramètre effectif (et d'autres informations éventuellement) pour donner une nouvelle valeur à ce paramètre.
Paramètre effectif : Le paramètre effectif correspondant est nécessairement une variable (ou assimilé : élément d'un tableau ou champ d'un enregistrement) dont le type est identique au type du paramètre formel.
Notation : Un paramètre formel en entrée/sortie est déclaré en utilisant les mots-clés **`in out`** :

```
<nom> : in out <Type>    -- paramètre formel correspondant à une donnée
                         -- en entrée-sortie
```

Exemple : La procédure `permuter` est un exemple de procédure qui a des paramètres en entrée-sortie. La valeur de chaque paramètre est utilisée pour modifier la valeur de l'autre paramètre.

```
Procédure permuter(a : in out Entier ; b : in out Entier) Est
        -- échanger les valeurs de deux valeurs entières
    Variables
        tmp : Entier    -- utilisé pour faire la permutation
    Début
        tmp ← a
        a ← b
        b ← tmp
    Fin
```

```
Variable
    i, j : Entier;
Début
    Lire(i)
    Lire(j)
    permuter(i, j)      -- OK
    permuter(i, 5)      -- NON : le deuxième paramètre n'est pas une variable
Fin.
```

Sur l'exemple de cette procédure, on constate bien que le sous-programme commence par lire la valeur du paramètre formel (récupère l'information transmise par l'appelant) avant d'en changer la valeur (modification qui sera visible de l'appelant).

Exemple : Sous-programme qui incrémente un numéro de mois.

```
Procédure incrémenter_mois(num_mois: in 1..12) Est
        -- incrémenter le numéro du mois
        --
        -- Nécessite
        --              num_mois >= 1 Et num_mois <= 12
    Début
        Si num_mois < 12 Alors
            num_mois ← num_mois + 1
        Sinon
            num_mois ← 1
        FinSi
    Fin
```

6.2 Les variables globales

Attention : L'utilisation des variables globale, même si elle constitue une possibilité pour faire communiquer des sous-programmes, **ne doit pas être utilisée**.

Nous pouvons apporter plusieurs jsutifications :

– les variables globales rendent le sous-programme dépendant de son contexte. Il ne peut donc plus être réutilisé librement ;

- l'utilisation de variables globales rend la compréhension du sous-programme plus difficile car il ne dépend pas seulement des informations qui lui sont transmises explicitement par les paramètres formels mais également de la valeur (de l'état) des variables globales dont les modifications sont difficiles à prévoir (il faut lire tout le programme) ;
- l'utilisation des variables globales peut conduire à des erreurs très difficiles à trouver (voir l'exemple donné à la section 4).

Il y a cependant des cas où les variables globales sont effectivement la bonne façon de traiter un problème. Des précautions doivent être prises, en particulier de telles variables doivent être isolées du reste de l'application et sont généralement définies à l'intérieur d'un module.

Un exemple où l'utilisation d'une variable globale est nécessaire, c'est lorsqu'une procédure doit conserver une information entre deux appels consécutifs. Par exemple, le nombre de fois où la procédure est appelée. Ceci peut être réalisé par une variable globale qui ne sera utilisée que par cette procédure.

Un autre exemple est lorsqu'on veut représenter une information qui est externe au programme (qui a donc la durée de vie du programme) telle que le nombre de lignes utilisées sur l'écran (pour gérer un affichage page par page). La encore, il est souhaitable que la variable globale ne soit accessible que par un nombre restreint de sous-programmes.

Règle : Même si la plupart des langages de programmation le permettent, une procédure ne doit pas utiliser de variables globales. Ceci est source d'erreurs difficiles à détecter et nuit à la compréhension et donc à la maintenance des programmes. Toutes les informations que manipule un sous-programme doivent donc lui être passées par l'intermédiaire de ses paramètres.

Si on a vraiment besoin de variables globales, alors il faut clairement le préciser dans le commentaire de la variable globale (lister tous les sous-programmes qui l'utilisent et/ou la modifient) et dans le commentaire de tous les sous-programmes qui l'utilisent. La variable globale est en fait un paramètre caché.

7 Exécution d'un appel à un sous-programme

7.1 Vérification statique

Avant d'être exécuté, un algorithme doit être vérifié. En ce qui concerne l'appel d'un sous-programme, il faut vérifier que :

1. l'appel du sous-programme apparaît à un *endroit autorisé* :
   - dans le cas d'une procédure, l'appel doit correspondre à une instruction ;
   - dans le cas d'une fonction, l'appel doit correspondre à une expression (dont le type est compatible avec le type de retour de la fonction) ;
2. le *nombre de paramètres effectifs* correspond au nombre de paramètres formels ;
3. le *type de chaque paramètre effectif* est identique au type du paramètre formel correspondant.
4. le paramètre effectif est compatible avec le *mode de passage du paramètre formel* :
   - pour un paramètre en entrée : toute expression (du bon type) est autorisée ;
   - pour un paramètre en sortie ou en entrée/sortie : le paramètre effectif doit nécessairement être une variable (ou un élément d'un tableau ou un champs d'un enregistrement) dont le type est identique à celui du paramètre formel.

7.2 Exécution d'un appel

Lorsque l'appel d'un sous-programme est statiquement correct, il peut être appelé. Un *bloc d'activation* est créé dans la pile. Il contient les informations suivantes :

- le numéro d'instruction à exécuter après le sous-programme (retour après le déroutement) ;
- les paramètres formels initialisés en fonction des paramètres effectifs ;
- les variables locales du sous-programmes.

Les étapes correspondant à l'exécution de l'appel d'une procédure sont les suivantes :

1. Évaluer les paramètres effectifs qui correspondent à un paramètre formel en entrée.
2. Conserver le numéro de l'instruction qui a provoqué l'appel au sous-programme.
3. Réserver de la place pour les paramètres formels.
4. Initialiser, en fonction de leur mode de passage, les paramètres formels à partir des paramètres effectifs.
   - paramètre en entrée : le paramètre formel est initialisé avec la valeur du paramètre effectif ;
   - paramètre en sortie : le paramètre formel est initialisé avec l'adresse du paramètre effectif. Pour indiquer que la valeur du paramètre effectif n'est pas utilisée, on peut l'initialiser avec un point d'interrogation. Tout accès au paramètre formel est équivalent à un accès au paramètre effectif ;
   - paramètre en entrée/sortie : le paramètre formel est initialisé avec l'adresse du paramètre effectif. Tout accès au paramètre formel est équivalent à un accès au paramètre effectif.

5. Réserver de la place pour les éventuelles variables locales. Elles ne sont pas initialisées. On l'indique en mettant un point d'interrogation.
6. Exécuter les instructions du corps de la procédure comprises entre les mots-clés **Début** et **Fin**.
7. Libérer (effacer) la place occupée par les variables locales et les paramètres formels lorsque l'exécution de la procédure est terminée (le mot-clé **Fin** est atteint).
8. Continuer l'exécution dans le programme appelant (fin du déroutement). L'instruction suivante à exécuter est celle qui suit l'instruction dont le numéro a été conservé lors de l'appel du sous-programme.

Remarque : Dans le cas d'une fonction, les paramètres sont forcément en entrée et on ajoute à la liste des variables locales une variable prédéfinie de nom **Résultat**. Sa valeur à la fin de l'exécution du sous-programme (juste avant qu'elle ne soit effacée) correspond à la valeur renvoyée par la fonction.

7.3 Illustration

Considérons la procédure suivante :

```
1  Procédure illustrer_exécution(  a : in Entier ;
2                                  b : out Entier ;
3                                  c : in out Entier) Est
4      Variable
5          i : Entier
6      Début
7          Pour i ← 1 JusquÀ i = a Faire
8              b ← b + a
9          FinPour
10         c ← b
11     Fin
```

et l'appel :

```
11 Variable
12     n1, n2: Entier
13 Début
14     n1 ← 1
15     n2 ← 2
16     illustrer_exécution(3, n1, n2)
17 Fin.
```

La figure 2 indique l'état de la pile correspondant à l'exécution de ce programme jusqu'à l'appel de la procédure (les étapes 1 à 5 ont été réalisées, l'instruction suivante à exécuter est donc celle qui porte le numéro 7, première instruction de la procédure). L'étape suivante est donc la 6 : exécuter les instructions du corps de la procédure.

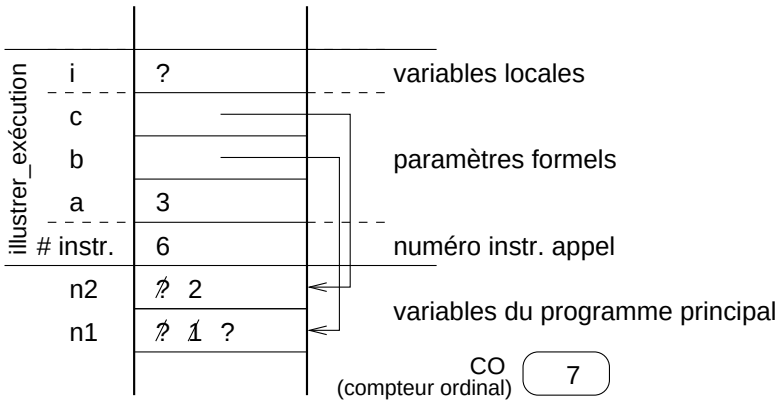

FIG. 2 – État de la mémoire avant l'exécution du corps de la procédure

8 Procédures

Dans les paragraphes précédents, nous avons introduit la notion de procédure en partant d'exemples. Dans ce paragraphe, nous donnons une définition plus précise d'une procédure. En particulier, nous insistons sur les notions d'interface et d'implantation d'une procédure. L'interface s'adresse à l'utilisateur de la procédure : elle décrit ce que fait la procédure et la manière de l'utiliser. L'implantation s'adresse au processeur : elle décrit l'enchaînement d'instructions correspondant au traitement réalisé par la procédure. Nous terminons ce paragraphe en décrivant comment définir une procédure.

8.1 Interface d'une procédure

L'interface s'adresse à l'utilisateur de la procédure (un programmeur). Elle doit donc préciser la syntaxe d'appel de la procédure : c'est la signature ou le prototype de la procédure. Cependant, la connaissance de la syntaxe d'appel n'est pas suffisante. Il est également nécessaire de savoir ce que fait la procédure : c'est la sémantique de la procédure qui, malheureusement, ne peut souvent qu'être décrite de manière informelle dans un commentaire[2].

8.1.1 Signature

Pour définir la syntaxe d'appel, il est nécessaire de définir :
- le **nom** de la procédure. Ce nom doit être significatif, c'est-à-dire qu'il doit suggérer le traitement que réalise la procédure. Puisqu'une procédure correspond à un traitement, une instruction de haut niveau, on lui donnera[3] pour nom un verbe à l'infinitif ;
- les **paramètres formels** de la procédure. Un paramètre formel décrit une information échangée entre le programme appelant (à l'origine de l'appel) et la procédure. Il est com-

[2] La programmation contractuelle telle qu'elle est proposée par B. Meyer est une solution partielle à se problème qui n'est malheureusement pas supportée par les langages de programmation traditionnels.

[3] pourra lui donner. Ceci est conseillé mais, bien entendu, pas obligatoire !

posé d'un nom (significatif bien sûr), d'un type (tout type est possible) et d'un mode de passage (en entrée, en sortie ou en entrée/sortie, voir paragraphe 6.1).
Une procédure peut avoir zéro, un ou plusieurs paramètres formels. Les différents paramètres formels sont séparés par des points virgules.
Conseil : Une procédure ne doit pas avoir trop de paramètres. Si le nombre de paramètres est trop important, c'est peut-être parce que la procédure fait trop de choses, il faut alors la décomposer, peut-être parce qu'elle est trop élémentaire, peut-être parce que les données manquent de structure, etc.

8.1.2 Sémantique

La sémantique d'une procédure est décrite dans un commentaire. Ce n'est pas parce que le compilateur ne peut pas exploiter les commentaires qu'il faut les négliger. En effet, les programmes sont avant tout lus lors du développement initial mais surtout pour les phases de maintenance (correction et évolution) par des êtres humains (donc des êtres capables de lire et comprendre les commentaires).

Le commentaire doit faire apparaître les informations suivantes :

- l'**objectif** de la procédure. Il doit décrire ce que fait la procédure et non comment elle le fait. Cette description doit être précise et succincte ;
- les **préconditions** sur les paramètres formels en entrée. Elles décrivent les propriétés que doivent respecter les paramètres en entrée pour que la procédure puisse être exécutée ;
- les **postconditions** sur les paramètres en sortie : elles décrivent les propriétés qui caractérisent les paramètres en sortie après l'exécution de la procédure.

Les rubriques ci-dessus sont essentielles. Les renseigner systématiquement aide à bien définir ce que fait la procédure.

Il est possible de rajouter d'autres rubriques comme par exemple la complexité de la procédure (un ordre de grandeur du temps d'exécution de la procédure en fonction de la valeur des paramètres en entrée). On peut également indiquer la solution retenue pour implanter la procédure, mais sa description doit être très succincte.

8.2 Implantation d'une procédure

L'interface décrit ce que fait la procédure. L'implantation décrit comment elle le fait. Elle s'adresse donc au processeur. En d'autres termes, l'interface décrit le problème à résoudre, l'implantation décrit une solution (un algorithme) pour le résoudre. Comme tout algorithme, il est donc composé de **variables locales** et d'**instructions**.

8.3 Comment définir une procédure ?

Ici, nous proposons une manière de définir une procédure. Ce n'est bien entendu pas la seule manière de faire. Il est important de commencer par définir l'interface (savoir ce que fait la procédure) avant l'implantation (savoir comment elle le fait). Nous conseillons donc de suivre les étapes suivantes :

1. **Définir l'interface de la procédure :**
   (a) Définir l'*objectif* de la procédure. Ceci a pour objectif de bien comprendre ce que doit faire la procédure (cf R0 des raffinages).
   (b) Identifier les *paramètres formels* et leur mode de passage (entrée et/ou sortie).
   (c) En déduire un *nom significatif* pour la procédure.
   (d) Identifier les *préconditions* et les *postconditions* sur ces paramètres (et l'état du système).
   (e) Rédiger l'interface de la procédure à partir des informations ci-dessus.
2. **Définir l'implantation de la procédure :** Une fois l'interface définie, on peut alors s'intéresser à son **implantation**. Une méthode possible est celle par raffinages successifs, le R0 étant défini par l'interface de la procédure. On a en effet ce que doit faire la procédure (objectifs et postconditions), les données manipulées (paramètres formels) et les hypothèses sur ces paramètres (préconditions).

Remarque : Une fois l'implantation terminée, on peut compléter l'interface en indiquant la complexité de la solution retenue, le principe de la solution, etc.

9 Fonctions

Tout comme une procédure, une fonction est un sous-programme. La différence est qu'une fonction renvoie une information et donc peut être assimilée à une expression, alors qu'une procédure effectue un traitement et correspond donc à une instruction. Comme une procédure, une fonction a une interface et une implantation que nous décrivons ci-dessous.

9.1 Interface d'une fonction

L'interface d'une fonction est composée d'une partie syntaxique correspondant à sa signature et d'une partie sémantique.

9.1.1 Signature

La signature d'une procédure est composée :
- d'un **nom** (significatif) qui caractérise l'information renvoyée par la fonction et est donc soit un nom commun, soit un adjectif[4] ;
- des **paramètres formels**. Ils sont tous en entrée. Une fonction ne peut pas avoir de paramètres en sortie ou en entrée/sortie. Une fonction doit toujours avoir au moins un paramètre formel.
  Justification : Si une fonction n'a pas de paramètres, c'est une constante ! Sinon c'est qu'elle utilise une variable globale, ce qui est à éviter, ou qu'elle réalise des entrées/sorties. Dans ce dernier cas, il faudrait en faire une procédure, comme `Lire` ;
- un **type de retour** qui est le type de l'information renvoyée.
  Attention : Le type de retour d'une fonction est soit un type élémentaire, soit un type énuméré, soit un type intervalle, soit un type chaîne de caractères. Ce ne peut être ni un enregistrement, ni un tableau.[5]
  Question : Comment fait-on pour renvoyer une information d'un tel type ?
  Solution : On utilise une procédure avec un paramètre formel en sortie !

9.1.2 Sémantique

La sémantique d'une fonction est précisée dans un commentaire qui contient les rubriques suivantes :
- l'**objectif** de la fonction qui décrit ce que calcule la fonction ;
- les **préconditions** sur les paramètres formels ;
- les **(post)conditions** sur le résultat de la procédure.

Comme dans le cas d'une procédure, d'autres rubriques peuvent être ajoutées.

[4] Ceci est conseillé mais, bien entendu, pas obligatoire !

[5] Ceci est en fait dépendant du langage. Ces contraintes sont celles du langage Pascal. D'autres langages autorisent tout type comme type de retour.

9.2 Implantation d'une fonction

L'implantation d'une fonction décrit comment le calcul est réalisé. Elle est similaire à celle d'une procédure à la différence qu'une variable locale à la fonction est automatiquement déclarée. Son nom est **`Résultat`** et son type est le type de retour de la fonction. Elle peut (doit) être utilisée pour réaliser les calculs. La valeur renvoyée par la fonction est la valeur de la variable **`Résultat`** à la fin de l'exécution des instructions de la fonction.

10 Procédure ou fonction ?

Une fonction peut être vue comme un cas particulier de procédure. Dans les paragraphes suivants, nous expliquons comment réécrire une fonction sous la forme d'une procédure. Nous en déduisons l'intérêt de définir des fonctions plutôt que des procédures. Enfin, nous définissons des critères permettant de décider si un sous-programme doit être décrit comme une fonction ou une procédure.

10.1 La fonction comme procédure

Toute fonction peut s'écrire sous la forme d'une procédure. La procédure correspondante prend un paramètre formel de plus que la fonction. Donnons un nom à ce paramètre supplémentaire... au hasard... résultat. Son type est le type de retour de la fonction et son mode de passage est en sortie. Le nom de la procédure est le nom de la fonction précédée de « `calculer_` ».

Prenons un exemple concret : la fonction max qui renvoie le plus grand de deux entiers.

```
1 Fonction max(a : in Entier ; b : in Entier) : Entier Est
2         -- le plus grand de deux nombres entiers
3     Début
4         Si a > b Alors
5             Résultat ← a
6         Sinon
7             Résultat ← b
8         FinSi
9     Fin
```

La procédure correspondante s'écrit alors :

```
1 Procédure calculer_max(résultat: out Entier ; a : in Entier ; b : in Entier) Est
2         -- calculer dans résultat le plus grand des deux nombres entiers a et b
3     Début
4         Si a > b Alors
5             résultat ← a
6         Sinon
7             résultat ← b
8         FinSi
9     Fin
```

Remarque : Dans l'optique d'un sous-programme le paramètre en sortie que nous avons appelé `résultat` car il correspond au **`Résultat`** de la fonction devrait s'appeler `max`, comme la fonction. Ce n'est pas un hasard !

10.2 Intérêt des fonctions par rapport aux procédures

Si la fonction `max` et la procédure `calculer_max` sont strictement équivalentes par rapport à ce qu'elles calculent, la fonction est d'un emploi beaucoup plus souple. Prenons un exemple. On veut calculer la plus grande des valeurs de 3 variables entières (n1, n2 et n3) et de la constante entière 15 et mettre le résultat dans la variable `plus_grand`. On suppose les différentes variables déjà déclarées. En utilisant la fonction `max`, on peut écrire :

```
1 plus_grand ← max(max(n1, n2), max(n3, 15))
```

alors qu'avec la procédure on est obligé de décomposer les calculs en écrivant (on suppose pg1 et pg2 deux variables entières déjà déclarées) :

```
1 calculer_max(pg1, n1, n2)
2 calculer_max(pg2, n3, 15)
3 calculer_max(plus_grand, pg1, pg2)
```

Ce qui peut également s'écrire :

```
1 calculer_max(plus_grand, n1, n2)
2 calculer_max(plus_grand, plus_grand, n3)
3 calculer_max(plus_grand, plus_grand, 15)
```

On constate que l'écriture avec la fonction est plus naturelle et que le calcul peut s'effectuer en une ligne grâce à une expression composée. Bien entendu, il faut éviter d'écrire des expressions trop compliquées qui deviendraient difficiles à lire et à comprendre.

L'intérêt des fonctions par rapport aux procédures est donc d'ordre syntaxique : elles sont plus facile à utiliser car elles peuvent apparaître dans une expression (et si on veut les utiliser pour initialiser une variable, il suffit de les utiliser avec l'affectation).

10.3 Choisir entre fonction et procédure

Puisqu'une fonction est plus souple d'utilisation, il est préférable d'écrire une fonction plutôt qu'une procédure. Cependant, ceci n'est pas toujours possible. Des contraintes présentées sur le type de retour et les paramètres d'une fonction (section 9), on peut en déduire la règle suivante.
Règle pour pouvoir écrire une fonction : Si on a identifié un sous-programme qui a les propriétés suivantes :

- elle a un seul paramètre formel en sortie,
- le type du paramètre en sortie est ni un enregistrement, ni un tableau,
- tous ses autres paramètres formels (il y en a au moins un) sont en entrée,

alors il est préférable de l'écrire sous la forme d'une fonction.
Conseil : Il faut donc, dans un premier temps, raisonner en terme de sous-programme. C'est lorsque vous aurez identifié les paramètres formels (et leur mode de passage) que vous pourrez choisir de décrire le sous-programme comme une fonction ou une procédure (suivant que la règle ci-dessus s'applique ou non).

11 Liens entre raffinages et sous-programmes

Chaque niveau intermédiaire d'un raffinage est un candidat à être une procédure (ou une fonction pour les expressions). Ce n'est cependant pas une condition nécessaire. Tout niveau intermédiaire ne doit pas être transformé en sous-programme.

Les critères qui font qu'un niveau de raffinage est transformé en sous-programme sont :

- le *sous-programme est réutilisable* : est-ce que le traitement est général et pourra être utilisé à plusieurs endroits dans l'algorithme ou éventuellement dans d'autres algorithmes ?
- l'*amélioration de la lisibilité* : le sous-programme représente quelques lignes de code qui, si elle étaient laissées dans le programme appelant, le rendrait difficile à lire (trop long, trop détaillé, etc.) ;
- le *sous-programme correspond à un traitement bien défini*, relativement indépendant (pas trop spécifique) des autres étapes de l'algorithme.
  En particulier, le nombre de paramètres de la procédure doit être limité. Si une procédure a un grand nombre de paramètres, c'est qu'elle est trop liée à son contexte d'appel. En faire une procédure ne permettra certainement pas de gagner en clarté.

12 Imbrication de sous-programmes

Chaque niveau de raffinage intermédiaire peut correspondre à un sous-programme (paragraphe 11). Par ailleurs, chaque sous-programme constitue un sous-problème qui pour être résolu doit être raffiné (paragraphe 8). Ainsi, le raffinage d'une procédure peut faire intervenir des étapes de haut niveaux qui, à leur tour, peuvent correspondre à des sous-programmes. Puisque ces sous-programmes ont été identifiés pendant le raffinage d'un sous-programme, ils peuvent être considérés comme spécifiques, locaux à ce sous-programme. On parle alors de **sous-programmes imbriqués**.

Tout comme une variable locale d'un sous-programme, un sous-programme local à un sous-programme ne sera visible (et donc utilisable) que depuis ce sous-programme. Si son nom correspond à celui d'une entité définie à un niveau supérieur, il masque cette entité.

En algorithmique, nous déclarerons les sous-programmes locaux à un sous-programme avant la déclaration des variables locales du sous-programme englobant. Tout comme dans le cas de l'algorithme principal, ceci évite que les sous-programmes utilisent par inadvertance ces variables locales. Attention cependant, les paramètres formels du sous-programme englobant sont des **données globales** pour les sous-programmes locaux. Il faut donc bien faire attention à ce qu'on fait.

12.1 Tolérance sur les sous-programmes

Nous avons dit dans le paragraphe 8, qu'un sous-programme ne devait manipuler que les données qui lui sont passées en paramètres. En particulier, il ne doit pas utiliser de variables globales. Cependant, dans le cas de sous-programmes imbriqués, il est toléré qu'ils puissent utiliser les paramètres formels (et même des variables !) du sous-programme englobant. Ceci est justifié par le caractère local et donc spécifique du sous-programme.

Conseil : Je pense toutefois que ce genre de pratique est à éviter car il conduit à des erreurs difficiles à détecter (voir l'exemple de la section 4) et que d'autre part les sous-programmes sont plus difficiles à comprendre et à maintenir car ils dépendent de leur contexte (du sous-programme englobant et de ses données).

12.2 Sous-programmes d'intérêt général

Si, lors du raffinage d'un sous-programme, vous identifiez un sous-programme qui est suffisamment général pour pouvoir être utilisé à d'autres endroits de l'algorithme ou dans de futurs algorithmes, il est souhaitable de ne pas en faire un sous-programme local, mais de le faire apparaître au premier niveau.

Ainsi, s'il y a effectivement un lien entre raffinage et sous-programme, on constate toutefois que la structure hiérarchique du raffinage (déduite de l'analyse et de la résolution du problème posé) peut ne pas correspondre à la structure hiérarchique d'imbrication des sous-programmes (des sous-programmes apparus relativement tard dans le raffinage peuvent apparaître au premier niveau dans l'algorithme final).

Reprenons l'exemple du paragraphe 2.5. Pour implanter la procédure `reculer` qui fait reculer le robot de n cases, nous identifions le raffinage composé des étapes suivantes :

```
1 faire demi-tour
2 progresser de n cases
3 faire demi-tour
```

Chacune de ces étapes est elle-même complexe et peut donc donner lieu à une procédure. Comme elles sont identifiées au cours du raffinage de la procédure `reculer`, elles peuvent être définies comme procédures locales. Voici l'algorithme correspondant :

```
1  Procédure reculer(n : in Entier) Est
2          -- faire avancer le robot de n cases vers l'arrière, n étant
3          -- strictement positif.  La direction du robot n'est pas modifiée.
4          --
5          -- Algorithme : avancer dans la direction opposée.
6
7          Procédure faire_demi_tour Est
8                  -- faire faire demi tour au robot
9              Début
10                 PIVOTER
11                 PIVOTER
12             Fin
13
14         Procédure progresser(n : in Entier) Est
15                 -- Faire avancer le robot de n cases dans la direction courante.
16                 -- N doit être un entier strictement positif
17             Variable
18                 i : Entier
19             Début
20                 Pour i ← 1 JusquÀ i = N Faire
21                     AVANCER
22                 FinPour
23             Fin
24
25     Début                -- procédure reculer
26         faire_demi_tour
27         progresser(n)                    -- n est le paramètre effectif de progresser
28         faire_demi_tour
29     Fin
```

Cependant les procédures `faire_demi_tour` et `progresser` sont suffisamment générales pour être utilisées dans d'autres algorithmes. Aussi, au lieu de les définir localement, il est préférable de les définir au premier niveau. Elles pourront ainsi être réutilisées.

Noter que d'imbriquer les sous-programmes n'améliore pas la lecture du code. En conséquence, il faut éviter d'avoir plusieurs niveaux d'imbrication et/ou trop de sous-programmes locaux.

12.2.1 Sous-programmes et langages de programmation

Si l'imbrication est possible dans certains langages (Pascal, Modula2, Ada, Lisp, Caml...), elle est interdite dans d'autres (C, C++, Eiffel...).

13 Les sous-programmes en C

13.1 Tout est fonction

En C, les notions de « procédures » et « fonctions » existent mais l'usage peut qu'on les appelle toutes fonctions. Nous garderons cependant la terminologie *fonction* et *procédure* avec leur sens algorithmique.

Comme en algorithmique, une fonction est caractérisée par :
- un identifiant (le nom de la fonction) ;
- une liste de paramètres formels (nombre, noms et types) ;
- un type de retour ;
- un corps composé d'instructions.

Cependant, il n'y a pas de variable **`Résultat`**. On utilise l'opérateur **`return`** `expression` qui arrête l'exécution de la fonction et renvoie comme résultat de la fonction la valeur de `expression`. Nous imposerons qu'il n'y ait qu'un seul **`return`** par fonction, nécessairement placé avant l'accolade finale.[6]

Remarque : Une **procédure** est une « fonction » qui ne renvoie pas de résultat. Son type de retour est donc **`void`**. Elle ne comporte pas de **`return`**.[7]

La syntaxe générale d'une fonction est :

```
1 /* Description de l'objectif du sous-programme avec les éventuelles
2  * préconditions et postconditions.
3  */
4 <type_retour> <nom_fonction> (<paramètres_formels_typés>)
5 {
6     /* corps de la fonction */
7     return <expression>;
8 }
```

En C, il n'y a pas de contrainte sur la valeur de retour d'une fonction. On peut donc renvoyer un enregistrement et un tableau (même si le cas des tableaux est un peu particulier, voir section 13.4)

Exemples : Voici un sous-programme appelé `max2`, en fait une fonction, qui calcule le plus grand des deux entiers passés en paramètre.

```
1 /* Le plus grand des deux entiers a et b. */
2 int max2(int a, int b) {
3     int resultat;
```

[6] En C, il est possible de mettre plusieurs **`return`** mais ceci contredit le principe qu'une fonction, un sous-programme en général, doit avoir un seul point d'entrée et un seul point de sortie.

[7] En C, il est possible d'avoir un **`return`** sans expression mais nous l'interdisons.

```
4       if (a > b) {
5           resultat = a;
6       }
7       else {
8           resultat = b;
9       }
10      return resultat;
11  }
```

Voici une fonction max3 qui calcule le plus grand des trois entiers passés en paramètre en s'appuyant sur la fonction précédente max2. Notons qu'en C, on ne peut utiliser une fonction que si elle est déjà connue du compilateur donc si elle a été déclarée avant.

```
1  /* Le plus grand des trois entiers n1, n2 et n3. */
2  int max3(int n1, int n2, int n3) {
3      int max12;  /* le plus grand de n1 et n2 */
4      max12 = max2(n1, n2);
5      return max2(max12, n3);
6
7          /* Remarque : on aurait pu ne pas utiliser la
8           * variable locale max12 en faisant :
9           *
10          *      return max2(max2(n1, n2), n3);
11          */
12 }
```

Enfin, voici un programme de test qui utilise les fonctions max2 et max3.

```
1  /* Programme de test de la fonction max2() */
2  void tester_max()
3  {
4      int a = 2;
5      int b = 5;
6      int i, j;
7
8      printf("a = %d et b = %d\n", a, b);
9      i = max2(a, b);     /* les deux paramètres sont des variables */
10     printf("i = max2(a, b) = %d\n", i);
11     j = max2(8, 2*i);   /* mais peuvent être des expressions quelconques */
12     printf("j = max2(8, 2*i) = %d\n", j);
13     printf("max3(9, i, j) = %d", max3(9, i, j));
14 }
```

Le résultat de l'exécution donne :

```
1  a = 2 et b = 5
2  i = max2(a, b) = 5
3  j = max2(8, 2*i) = 10
4  max3(9, i, j) = 10
```

13.2 Mode de passage des paramètres

En C, il n'existe qu'un seul mode de passage des paramètres, le passage par valeur. La question est alors comment faire pour écrire un sous-programme qui doit changer la valeur des paramètres effectifs (comme par exemple permuter) ? La solution est de s'appuyer sur la notion de pointeur pour faire un passage « par adresse » qui est un fait un passage par valeur de l'adresse d'une variable. Ça paraît compliqué... mais c'est développé dans les points suivants.

13.2.1 Le passage par valeur

Le seul mode de passage de paramètres en C est donc le passage par valeur. Ceci signifie que les paramètres formels d'un sous-programme sont assimilables à des variables locales qui sont initialisées avec la valeur du paramètre effectif correspondant (fourni par le programme appelant).

Il y a donc copie dans la pile d'exécution de la valeur des paramètres effectifs. Cette copie est accessible sous le nom du paramètre formel. Les conséquences sont donc qu'en C on peut :

- affecter un paramètre formel et plus généralement changer sa valeur ;
- ses changements étant faits sur la copie dans la pile d'exécution ils ne sont pas visibles du programme appelant.

Considérons l'exemple suivant.

```
1  /*****************************************************************************
2   *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
3   *  Version  : 1.1
4   *  Objectif : Exemple illustrant le passage par valeur en C.
5   *****************************************************************************/
6
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 void proc(int a, double b)
11 {
12     printf("proc début : a = %d et b = %f\n", a, b);
13     a++;        /* On incrémente la copie du paramètre effectif */
14     b = b * 2;  /* Idem */
15     printf("proc fin   : a = %d et b = %f\n", a, b);
16 }
17
18 int main()
19 {
20     int i = 2;
21     double x = 3.14;
22     printf("main début : i = %d et x = %f\n", i, x);
23     proc(i, x);
24     printf("main fin   : i = %d et x = %f\n", i, x);
25
26     return EXIT_SUCCESS;
27 }
```

Les valeurs du programme principal (`main`) sont i et x qui ont respectivement les valeurs 2 et 3.14. Lors de l'appel à la procédure `proc`,[8] de la place est réservée dans le pile d'exécution pour stocker la valeur de deux variables a et b de types respectifs **`int`** et **`double`**. Ces deux variables correspondent aux paramètres formels qui sont initialisés avec la valeur des paramètres effectifs (en l'occurrence 2 et 3.14, les valeurs de i et x). Les valeurs des paramètres formels sont modifiées dans le sous-programmes, c'est donc la zone réservée dans la pile d'exécution qui est modifiée, pas la zone mémoire occupée par les variables i et x du programme principal. Aussi quand la procédure `proc` se termine, les valeurs de i et x n'ont pas changées.

Voici le résultat de l'exécution de ce programme.

```
1 main début : i = 2 et x = 3.140000
2 proc début : a = 2 et b = 3.140000
3 proc fin   : a = 3 et b = 6.280000
4 main fin   : i = 2 et x = 3.140000
```

Le mode de passage par valeur peut être utilisé pour le mode **`in`** identifié un algorithmique.

Remarque : Pour avoir réellement un mode **`in`**, on peut ajouter un **`const`** devant le type du paramètre. Le compilateur émet alors au moins un message d'avertissement si on essaie de changer la valeur de ce paramètre. Voici une fonction qui calcule la distance entre deux réels en essayant de changer la valeur des paramètres (ce n'est qu'un exemple illustratif, il serait beaucoup plus naturel – et plus lisible – de renvoyer la valeur absolue de la différence !).

```
1  /*****************************************************************************
2   *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
3   *  Version  : 1.1
4   *  Objectif : Illustrer le const pour réaliser une passage en mode « in ».
5   *****************************************************************************/
6
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 /* distance entre les réels a et b.  */
11 double distance(const double a, const double b)
12 {
13     a = a - b;
14     if (a < 0) {        /* rendre a > 0 */
15         a = - a;
16     }
17     return a;
18 }
19
20 int main()
21 {
22     printf("distance(1, 3) = %f\n", distance(1, 3));
23     printf("distance(3, 1) = %f\n", distance(3, 1));
24
25     return EXIT_SUCCESS;
26 }
```

[8] Ce nom n'est pas significatif et le sous-programme n'a pas de commentaire car il a pour seul but d'illustrer le mode de passage de paramètres par valeur.

Le compilateur gcc emet les messages d'avertissement suivants :

```
exemple-distance-in.c: In function 'distance':
exemple-distance-in.c:13: warning: assignment of read-only location
exemple-distance-in.c:15: warning: assignment of read-only location
```

13.2.2 Les pointeurs

Définition : Un pointeur correspond à l'adresse en mémoire d'une donnée typée.

Par extension, on appelle également pointeur une variable dont la valeur est un pointeur et le type de cette variable.

Notation : Une variable de type pointeur est déclarée avec un * entre le type et l'identifiant de la variable.

```
int *pi;        /* pi est un pointeur sur un int */
double *pr;     /* pr est un pointeur sur un double */
```

Attention : Lorsque l'on parle de pointeur, il faut toujours préciser le type de l'information qui est pointée (qui se trouve à l'adresse considérée en mémoire). Ainsi, un pointeur sur un **int** et un pointeur sur un **double**, même si ce sont deux pointeurs, ne sont pas considérés comme des types compatibles.

Exemple simple : Commençons par un exemple simple qui montre le principe des pointeurs.

```
int i;          /* un entier i */
int j;          /* un autre entier j */
int *pi;        /* un pointeur sur entier pi */

i = 5;          /* i est initialisé à 5 */
pi = &i;        /* pi est initialisé avec l'adresse en mémoire de i */
j = *pi;        /* *pi est la valeur contenue dans la zone mémoire
                 * pointée par pi.  * est l'opérateur de déréférencement
                 */
printf("j = %d\n", j);  /* j = 5 */
*pi = 0;        /* changer la valeur de la zone mémoire pointée par pi */
printf("i = %d\n", i);  /* i = 0 */
    /* Conclusion : puisque pi est initialisé avec l'adresse de i, i
     * et *pi sont deux moyens différents d'accéder à la même zone
     * mémoire.
     */
```

Opérateur d'adressage : L'opérateur d'adressage, noté &, permet d'obtenir l'adresse en mémoire d'une variable. Cette adresse est compatible avec le type pointeur correspondant au type de la variable.

Opérateurs sur les pointeurs : Plusieurs opérateurs sont disponibles sur les pointeurs :

- le *déréférencement* : noté *, il fait référence à la zone mémoire pointée. Par exemple, si pi est un pointeur sur un entier, *pi est l'entier pointé.
  On ne peut utiliser l'opérateur de déréférencement que si le pointeur contient l'adresse d'une zone mémoire valide (c'est-à-dire que le programmeur a réservée). Dans le cas contraire, c'est une erreur dans le programme qui n'est pas détectée à la compilation et ne provoque pas toujours (malheureusement !) d'erreurs à l'exécution.

- l'*affectation* : ceci permet d'initialiser un pointeur. Un pointeur peut être initialisé avec :
  - l'adresse en mémoire d'une donnée de même type en utilisant l'opérateur d'adressage.

```
1 int i;
2 int *pi;
3 pi = &i;        /* pi est initialisé avec l'adresse en mémoire de i */
4     /* Remarque : i et *pi sont deux moyens différents pour accéder à
5      * le même zone en mémoire.
6      */
```

  - la valeur d'un pointeur. Les deux pointeurs pointent alors tous les deux sur la même zone mémoire ;

```
1 int i;          /* un entier i */
2 int *pi;        /* un pointeur sur entier pi */
3 int *p2;        /* un autre pointeur sur entier */
4 pi = &i;
5 p2 = pi;        /* p2 et pi contiennent tous les deux l'adresse de i */
```

  - la constante NULL (définie dans `stdlib.h`) qui signifie que le pointeur ne pointe sur aucune zone mémoire. Il ne peut pas être déréférencé.

```
1 int *pi;
2 pi = NULL;
3     /* Remarque : *pi n'a alors pas de sens.  Essayer de le faire
4      * devrait alors provoquer une erreur à l'exécution (ceci est
5      * dépendant du compilateur) !
6      */
```

- les opérateurs d'égalité : il est possible d'utiliser `==` et `!=` pour tester l'égalité et la différence.
- les opérateurs relationnels : on peut également utiliser les autres opérateurs de comparaison `>`, `<`, `<=`, `>=` qui consiste à comparer la valeur des adresses (voir section 13.4).
- les opérateurs arithmétiques : on peut incrémenter un pointeur. Ceci revient à avancer dans la mémoire d'un nombre d'octets égal à celui nécessaire pour représenter en mémoire la donnée pointée. On passe donc à la donnée « suivante » en mémoire. Ceci n'a de sens que si les données sont contiguës en mémoire, donc s'il s'agit de tableau (voir section 13.4)

Attention : Les opérateurs de comparaison et les opérateurs arithmétiques font partie de ce que l'on appelle l'arithmétique sur les pointeurs et que nous n'utiliserons pas !

Remarque : Les pointeurs servent également lorsque l'on fait de l'allocation dynamique de mémoire mais ce n'est pas l'objet de ce chapitre !

13.2.3 Le passage par adresse

Pour pouvoir modifier la valeur d'un paramètre en C, l'astuce consiste à ne pas passer en paramètre une variable, mais l'adresse de cette variable. Le sous-programme a alors directement accès à la zone mémoire du programme appelant et peut donc la modifier.

Je qualifie ceci d'une astuce car il s'agit bien d'un passage par valeur de l'adresse de la variable, et non d'un réel passage en mode **`out`** ou **`in out`**. Ainsi, dans le sous-programme, il

faudra penser à déréférencer le paramètre formel et lors de l'appel il faudra penser à donner l'adresse de la variable.

Voici comment écrire en C la fonction permuter.

```
1 /* Permuter la valeur des deux caractères c1 et c2 */
2 voir permuter(char *c1, char *c2)
3 {
4     char tmp;   /* pour conserver la valeur de *c1 */
5     tmp = *c1;
6     *c1 = *c2;
7     *c2 = tmp;
8 }
```

Voici comment on peut (doit l'utiliser).

```
1 char lettre = 'A';
2 char chiffre = '5'
3
4 printf("lettre = %c et chiffre = %c\n", lettre, chiffre);
5 permuter(&lettre, &chiffre);
6 printf("lettre = %c et chiffre = %c\n", lettre, chiffre);
```

Remarque : C'est ce que nous avons utilisé pour scanf...

13.3 Les préconditions et postconditions de la programmation par contrat

En C, il n'est pas possible d'exploiter automatiquement les préconditions et postconditions qui ont été identifiées lors de la spécification d'un sous-programme.

Cependant, il existe un module appelé <assert.h> qui offre la macro[9] `assert(condition)`. Si la condition est vraie alors l'exécution du programme continue normalement sinon un message d'erreur est affiché et l'exécution stoppée. La macro `assert` peut donc être utilisée dans le code pour vérifier qu'une propriété identifiée lors de l'écriture du programme est bien vérifiée lors de son exécution. Si ce n'est pas le cas, l'arrêt du programme et le message d'erreur nous permettrons de savoir qu'il y a un problème.

Voici comment écrire en C la fonction pgcd en tenant compte des préconditions et postconditions.

```
1 /*****************************************************************************
2  *  Auteur   : Xavier Crégut <cregut@enseeiht.fr>
3  *  Version  : 1.1
4  *  Objectif : Programme du pgcd et de son test
5  *****************************************************************************/
6
7 #include <stdio.h>
8 #include <stdlib.h>
9
10 #include <assert.h>
11
12 /* calculer le pgcd de deux entiers strictement positifs a et b.
```

[9] Une macro ressemble à une fonction C mais est traitée non pas par le compilateur C mais par le préprocesseur.

```
 *
 * Nécessite
 *      a > 0;
 *      b > 0;
 */
int pgcd (int a, int b)
{
    assert(a > 0);      /* utilisation de assert pour les préconditions */
    assert(b > 0);

    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return b;
}

/* Tester la fonction pgcd */
void tester_pgcd()
{
    printf("pgcd(4, 5) = %d (doit être 1)\n", pgcd(4, 5));
    printf("pgcd(10, 15) = %d (doit être 5)\n", pgcd(10, 15));
    printf("pgcd(0, 15) = %d (doit être 5)\n", pgcd(0, 15));
}

int main()
{
    tester_pgcd();
    return EXIT_SUCCESS;
}
```

On peut alors se poser la question de ce que donne le programme de test. On obtient donc :

```
pgcd(4, 5) = 1 (doit être 1)
pgcd(10, 15) = 5 (doit être 5)
exemple-pgcd: exemple-pgcd.c:20: pgcd: Assertion 'a > 0' failed.
```

Si on n'avait pas mis de `assert` pour vérifier les préconditions, on aurait eu un programme qui bouclait sans fin !

Remarque : Dans le version finale du programme, lorsqu'il a été bien testé et qu'on pense qu'il ne contient plus d'erreur, on peut désactiver les vérifications faites par `assert` en ajoutant l'option `-DNDEBUG` au compilateur C. Il n'est donc pas nécessaire de supprimer physiquement les appels à `assert` dans le programme.

13.4 Cas particulier des tableaux

Les tableaux sont un cas particulier en C. Les tableaux n'existent pas réellement et sont représentés comme des pointeurs.

Normalement, lorsque l'on parle d'un tableau d'entiers par exemple, on désigne l'ensemble des cases du tableau et les entiers qu'elles contiennent. On pourrait donc s'attendre à ce que lors d'un passage par valeur d'un tableau, tous les éléments du tableau soient copiés dans la pile. Ainsi, si le sous-programme modifie un élément du tableau, cette modification ne devrait pas être visible du programme appelant. Et bien il n'en est rien !

Prenons l'exemple de deux sous-programmes. Le premier permet de trier un tableau.

```
1  /* Trier le tableau tab qui contient nb éléments.
2   * @param tab   le tableau à trier
3   * @param nb    le nombre d'éléments de tab
4   */
5  void trier_insertion(int tab[], int nb)
6  {
7      int indice;         /* tab[0..indice] est trié */
8      int position;       /* position d'insertion de tab[indice] */
9      int i;              /* variable de boucle */
10     int memoire;        /* conserver la valeur de tab[indice]   */
11
12     for (indice = 1; indice < nb; indice++) {
13         /* Insérer tab[indice] dans tab(0..indice-1) */
14
15         /* Conserver la valeur de tab[indice] */
16         memoire = tab[indice];
17
18         /* déterminer la position théorique de tab[indice] */
19         position = 0;
20         while (position < indice && memoire >= tab[position]) {
21             position++;
22         }
23
24         /* décaler les éléments compris entre position et indice-1 */
25         for (i = indice-1; i >= position; i--) {
26             tab[i+1] = tab[i];
27         }
28
29         /* ranger l'élément */
30         tab[position] = memoire;
31     }
32 }
```

Le second permet d'afficher les éléments d'un tableau.

```
1  /* Afficher le tableau tab qui contient nb éléments.
2   * @param tab   le tableau à afficher
3   * @param nb    le nombre d'éléments à afficher
4   */
5  void afficher_tableau(int tab[], int nb)
6  {
7      /* Afficher le tableau */
8      printf("[");
9      if (nb > 0) {       /*{ le tableau n'est pas vide }*/
10         int i;  /* pour parcourir les indices du tableau */
```

```
11
12          /* afficher le premier élément */
13          printf("%i", tab[0]);
14
15          /* afficher les autres éléments */
16          for (i=1; i < nb; i++) {
17              printf("; %i", tab[i]);
18          }
19      }
20      printf("]");
21  }
```

On peut se demander ce que donne le programme de test suivant. Est-ce que le tableau est trié ou non, sachant que la tableau semble être passé par valeur ?

```
1  void tester1()
2  {
3      int tab1[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
4      afficher_tableau(tab1, 10); printf("\n");
5      trier_insertion(tab1, 10);
6      afficher_tableau(tab1, 10); printf("\n");
7  }
```

Le résultat de ce programme montre que le tableau est trié. Il ne s'agit donc pas d'un passage par valeur du tableau et de ses éléments !

```
1  [9; 8; 7; 6; 5; 4; 3; 2; 1; 0]
2  [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

En conséquence, le tableau n'est pas passé par valeur mais bien par adresse. C'est en fait l'adresse du premier élément qui est passé.

En fait les tableaux n'existent pas réellement en C et les crochets ne sont qu'une facilité syntaxique. Ainsi, noter `tab[i]` est identique à noter `*(tab+i)`. `tab+i` est donc l'adresse du i^e élément du tableau.

```
1  int tab[10] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
2  assert(tab == &tab[0]);                 /* tab == &tab[0] */
3  assert(tab[0] == *tab && *tab == 9);    /* tab[0] == *tab == 9 */
4  assert(tab[9] == *(tab+9));             /* tab[9] == *(tab+9) */
5  assert(*(tab + 9) == 0);                /*        == 0 */
```

Ceci permet des choses très condensés, peu lisibles mais parfois un peu efficaces. Nous n'utiliserons pas l'arithmétique sur les pointeurs et utiliserons la notation des tableaux avec les crochets.

Revenons sur l'exemple du tri d'un tableau pour voir comment tirer partie du fait que seule l'adresse du premier élément soit passée en paramètre. On peut le voir sur ce deuxième programme dans lequel on commence par trier la première moitié du tableau (on donne 5 pour le nombre d'éléments et non 10), puis la seconde moitié. Pour trier la seconde moitié, on donne comme premier paramètre `tab+5` qui est équivalent à `&tab[5]`, c'est-à-dire l'adresse du 5^e élément. On donne 5 pour la taille. On ne doit pas donner plus sinon on sort de la limite du tableau. Après exécution, on constate que les deux moitiés du tableau sont bien triées (le tableau n'étant bien sûr pas trié dans son ensemble).

```
1 void tester2()
2 {
3     int tab1[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 };
4     afficher_tableau(tab1, 10); printf("\n");
5     trier_insertion(tab1, 5);
6     afficher_tableau(tab1, 10); printf("\n");
7     trier_insertion(tab1+5, 5);
8     afficher_tableau(tab1, 10); printf("\n");
9 }
```

Le résultat de ce programme montre que le tableau est trié.

```
1 [9; 8; 7; 6; 5; 4; 3; 2; 1; 0]
2 [5; 6; 7; 8; 9; 4; 3; 2; 1; 0]
3 [5; 6; 7; 8; 9; 0; 1; 2; 3; 4]
```

13.5 Remarques diverses

On ne peut pas imbriquer les sous-programmes même si certains compilateurs (en particulier gcc/g++) le permettent !