Cours/UPMC/LI102/2

Programmation impérative et éléments d'algorithmique

Compléments de cours 2005-2006

Anne Brygoo
Titou Durand
Maryse Pelletier
Michèle Soria

PARACAMPLUS

Déjà paru :

UPMC/LI101 : annales corrigées

Prochainement(?) :

UPMC/LI102 : annales corrigées

UPMC/LI362 : annales corrigées

Missions de Paracamplus :

Paracamplus a pour vocation de procurer, à tous les étudiants en université, des ouvrages (annales corrigées, compléments de cours) et des environnements informatiques de travail, bref, des ressources pédagogiques propres à permettre, accélérer et entretenir l'apprentissage des connaissances enseignées.

Les ouvrages des éditions Paracamplus ont un format défini pour tenir commodément dans une poche et vous accompagner pendant vos déplacements en transports en commun. Ils bénéficient surtout d'une conception nous permettant de vous les proposer à des coûts très abordables.

Rafraîchir les annales tous les ans, procurer les compléments appropriés à chaque cours, vous proposer les meilleurs outils pour apprendre, tels sont les buts de ces ouvrages.

Découvrez-nous plus avant sur notre site **www.paracamplus.com**.

Il est interdit de reproduire intégralement ou partiellement la présente publication sans autorisation du Centre Français d'exploitation du droit de Copie (CFC) - 20 rue des Grands-Augustins - 75006 PARIS - Téléphone : 01 44 07 47 70, Fax : 01 46 34 67 19.

© 2006 Copyright retenu par les auteurs.

SARL Paracamplus
7, rue Viollet-le-Duc, 75009 Paris – France
ISBN 2-916466-01-0

Table des matières

Chapitre 1

Présentation du cours

Le cours *Programmation impérative et éléments d'algorithmique*, dispensé à l'université Pierre et Marie Curie sous le code de l'UE Li102, est une initiation à la programmation itérative. Dans le cursus MIME (Mathématiques, Informatique, Mécanique, Electronique) de l'UPMC, cet enseignement se situe à la suite d'un cours de *programmation fonctionnelle et récursive* en Scheme (Li101). Les connaissances du cours Li101 ne sont pas un prérequis absolu pour Li102, mais on insistera, particulièrement au début, sur les ressemblances et les différences entre ces deux approches de la programmation.

L'environnement de programmation choisi pour cette introduction à la programmation impérative est `Excel`®, avec un sous-ensemble de VBA (Visual Basic for Applications), son langage de macros. Une connaissance minimale des fonctionnalités d'un tableur et de la manipulation des formules est un prérequis nécessaire pour pouvoir écrire des programmes VBA sous Excel.

L'objectif premier du cours est de présenter les traits principaux de la programmation impérative. Pour exprimer les programmes, on se limite à un sous-ensemble restreint et structuré du langage VBA, tant au niveau des constructions syntaxiques qu'au niveau des propriétés et méthodes (une trentaine en tout). L'utilisation d'un environnement de programmation dédié permet de bénéficier de l'existant : représentation effective des objets en mémoire et primitives pour manipuler ces objets. Dans cet environnement, il est possible d'écrire très vite des programmes intéressants, réalistes ou ludiques. La feuille de calcul permet de traiter de nombreux problèmes modélisables par tableur, mais elle peut aussi servir de table graphique pour programmer des jeux.

Ce cours est aussi une initiation à l'algorithmique : on étudie tout d'abord le problème de la recherche d'un élément dans un ensemble et celui du tri d'un ensemble d'éléments, en comparant différents algorithmes de résolution. Enfin, pour terminer en revenant sur le tableur, on s'intéresse au problème du recalcul automatique des valeurs affi-

chées dans une feuille de calcul, en présentant la structuration des données et les algorithmes permettant un recalcul efficace lors de la modification d'une cellule.

Ce fascicule de compléments de cours s'organise en 9 chapitres, organisés en trois parties, d'inégale importance.

- La première partie est consacrée à l'étude de la programmation. Le chapitre 2 est une introduction, qui permet d'écrire les premières fonctions en VBA, en utilisant les structures de contrôle du langage. Le chapitre suivant (chapitre 3) est consacré à l'étude de la notion de boucle. On introduit ensuite les procédures et les macros, et les interactions entre programme et utilisateur (chapitre 4). Dans le chapitre 5, qui est une introduction à la programmation objet, on montre comment utiliser les classes de VBA sur l'application tableur. Le dernier chapitre (chapitre 6) de cette première partie illustre toutes les notions présentées antérieurement sur une étude de manipulation de matrices.
- La deuxième partie du cours est une initiation à l'algorithmique non numérique. En travaillant toujours sur la feuille de calcul du tableur, on présente le problème de la recherche d'un élément dans une séquence (chapitre 7) et celui du tri d'une séquence d'éléments (chapitre 8). On étudie différentes méthodes, différents algorithmes et différentes implantations, en les comparant du point de vue de leur efficacité.
- Enfin en dernière partie (chapitre 9), on étudie le problème de la cohérence des informations affichées sur une feuille de calcul : lorsque l'on modifie une formule, le tableur doit recalculer toutes les cellules impactées par cette modification. L'étude de cet algorithme de recalcul permet de mettre en évidence le graphe de dépendances qui structure la feuille de calcul et de travailler sur l'optimisation de l'ordre des calculs.

On trouvera de plus, en annexe, les éléments qu'il faut connaître sur le tableur `Excel`®(écriture des références, des formules. . .) pour pouvoir écrire des programmes dans cet environnement.

Dans le cadre de l'enseignement de LI102, plusieurs autres documents sont distribués aux étudiants :

- des informations pratiques sur les environnements de travail :
  - une fiche technique `Excel`®qui explique comment manipuler les classeurs, feuilles de calculs et cellules.
  - une fiche technique VBA expliquant comment utiliser l'environnement de programmation pour écrire, exécuter et déboguer des programmes.
- un manuel de référence pour ce cours, appelé mémento, qui présente la syntaxe de Visual Basic ainsi que la liste des propriétés et méthodes de VBA utilisées. (Ce mémento est le seul document autorisé à l'examen.)

Ces compléments de cours sont issus des transparents du cours de Li102 dispensé depuis quelques années à l'Université Pierre et Marie Curie. Nous tenons à remercier toute la hiérarchie de l'UPMC qui nous a toujours soutenus dans les diverses expériences, pédagogiques et logicielles, que nous avons tentées.

Un grand merci aussi à Séverine Dubuisson, qui a relu la version préliminaire avec beaucoup d'attention et nous a permis d'améliorer grandement la présentation.

Paris, le 2 Février 2006
Anne Brygoo,
Titou Durand,
Maryse Pelletier,
Michèle Soria.

Chapitre 2

Introduction à Visual Basic

L'objectif premier de ce cours est de présenter les traits principaux de la programmation impérative. L'environnement de programmation choisi est `Excel`, avec un sous-ensemble de VBA (Visual Basic for Applications). Ce chapitre présente les bases du langage et permet d'écrire des fonctions simples avec des affectations de variables et des alternatives.

Pour sérier les difficultés, nous avons choisi d'introduire les structures de contrôle en écrivant uniquement des fonctions (chapitres 2 et 3) et les procédures ne sont introduites qu'au chapitre 4.

2.1 Introduction

On présente d'abord quelques notions de base sur lesquelles s'appuie ce cours. Pour passer de la programmation fonctionnelle à la programmation impérative il faut comprendre quelques différences essentielles : environnement d'exécution, différence entre les entités élémentaires expression et instruction ; différence entre les mécanismes d'expression de la répétition, récursion et itération ; différence entre les deux types de sous-programmes, procédures et fonctions. On insiste aussi sur l'importance de la spécification d'un programme pour expliquer comment l'utiliser et ce qui en est attendu.

2.1.1 Environnement

Dans le cours de programmation récursive, on a décrit la notion d'environnement : en Scheme, l'environnement est constitué de définitions de fonctions et de valeurs de variables, ces dernières ayant une valeur constante tant qu'elles sont présentes dans l'environnement.

En programmation impérative, cette notion d'environnement est tout aussi, et même peut-être plus importante, avec une différence fonda-

mentale : la valeur d'une variable n'est plus constante, elle varie pendant l'exécution du programme. Ainsi, alors qu'en programmation fonctionnelle, on peut seulement *enrichir* l'environnement (par de nouvelles définitions), en programmation impérative on peut également *modifier* l'état présent de l'environnement - on parle alors d'environnement d'exécution.

2.1.2 Expressions et instructions

En style purement fonctionnel, on ne fait que composer des expressions et tout programme est décrit par une expression. En style impératif, un programme est un enchaînement d'instructions à exécuter.

On manipule ici deux types d'entités élémentaires : les expressions et les instructions

- une *expression* s'évalue en un résultat (d'un certain type).
  Par exemple `3+5-7` ou `factorielle(5)` sont des expressions entières et `X < 0` une expression booléenne.
- une *instruction* décrit une action de l'ordinateur. Par exemple l'appel de procédure `effacerTout` et l'affectation `X = X + 1` sont des instructions.

2.1.3 Récursion et itération

Les deux mécanismes permettant d'exprimer la répétition d'opérations dans un programme sont la récursion et l'itération (boucles).

- Une méthode récursive est fondée sur la décomposition du traitement initial en traitements sur des données strictement plus petites ; et il y a traitement direct des données "atomiques" de base. Par exemple la définition récursive de la fonction *factorielle* sur les entiers strictement positifs est donnée par une règle qui s'auto-applique,

$$fact(n) = \begin{cases} 1 & \text{si} \quad n = 0 \\ n \times fact(n) & \text{si} \quad n > 0 \end{cases}$$

- Dans une méthode itérative, on initialise un environnement d'exécution, puis on exécute un certain nombre de fois une suite d'instructions qui modifie un état de l'environnement d'exécution ; et l'exécution s'arrête lorsque cet état atteint un certain statut. Par exemple pour la définition itérative de la fonction *factorielle* sur les entiers strictement positifs, on peut écrire la boucle suivante, qui effectue le produit itéré des n premiers entiers naturels :

$$fact(n) = Produit(i\,;\, i \text{ allant de } 1 \text{ à } n)$$

2.1.4 Procédures et fonctions

Il existe deux sortes de (sous)-programmes : les fonctions et les procédures. En programmation purement fonctionnelle on n'utilise pas de procédures (et inversement le style de programmation impérative, qui utilise des procédures, est quelquefois appelé programmation procédurale).

Fonctions et procédures sont de natures différentes : une fonction renvoie une valeur (sans modifier l'environnement d'exécution), alors qu'une procédure ne renvoie rien mais modifie l'environnement. Par exemple une fonction qui calcule la somme de 2 et de 3 renvoie une valeur (5) - qui peut ensuite être utilisée (injectée) dans un autre calcul. En revanche une procédure qui affiche la valeur 5 modifie l'environnement (par la présentation d'une fenêtre contenant 5) mais cette valeur affichée ne peut en aucun cas être réutilisée.

Cette différence de nature se traduit au niveau de l'utilisation (appel) : un appel de fonction renvoie un résultat, qui est utilisé pour construire une expression, alors qu'un appel de procédure est une instruction.

2.1.5 Spécification des sous-programmes

La spécification d'un sous-programme définit les termes du contrat qui lie l'utilisateur et le réalisateur du sous-programme. La spécification d'un sous-programme contient deux parties :

- la *signature* (nom, liste et type des arguments, type du résultat pour une fonction), définit la syntaxe d'emploi ; elle est donnée dans l'en-tête du sous-programme.
- la *sémantique* décrit ce que fait le sous-programme ; elle est exprimée sous forme de commentaires, mais on s'attachera à respecter des règles systématiques : expliciter les hypothèses et les erreurs dans des clauses particulières, expliciter le résultat de l'exécution du sous-programme (renvoi d'une valeur pour une fonction et modifications de l'environnement d'exécution pour une procédure).

2.2 Visual Basic - Préliminaires

2.2.1 Types et expressions

Il y a trois types de base dans le domaine de programmation : numérique, logique et chaîne de caractères. Et les expressions s'écrivent en notation mathématique habituelle.

- Les types numériques sont les types `Integer`, `Long` (entiers longs), `Byte` (entiers compris entre 0 et 255), `Single` et `Double` (réels) avec les opérations habituelles (la division entière est notée `\`, et l'opérateur « modulo » est noté `mod`).

- Les constantes du type logique `Boolean` sont `False` et `True` et les opérations `Not`, `And`, `Or`.
- Le type `String` est le type des chaînes de caractères, les constantes étant notées entre guillemets anglais (") et l'opération de concaténation étant notée « & ».

Les opérateurs binaires de comparaison =, <> , <, >, <=, >= comparent deux valeurs numériques (ou deux chaînes de caractères) et produisent une valeur logique.

2.2.2 Aspects lexicaux

Visual Basic est un langage ligne : le passage à la ligne fait partie de la syntaxe du langage.

Commentaires

Les commentaires sont destinés au lecteur humain du programme, et ignorés lors de l'exécution. Il y a deux façons d'indiquer les commentaires en VBA : à l'aide du mot-clé `Rem` ou à l'aide du symbole apostrophe « ' ». Le reste de la ligne est alors ignoré. Un tel commentaire doit donc tenir sur une seule ligne (pour écrire des commentaires sur plusieurs lignes il faut mettre une apostrophe au début de chaque ligne).

```
' Ceci est un commentaire
' qui se poursuit sur la ligne suivante

Rem Ceci est un autre commentaire
```

Caractère de continuation

VBA étant un langage ligne, le changement de ligne est significatif : par exemple les en-têtes des sous-programmes doivent être écrites sur une seule ligne. Si une en-tête (ou toute autre instruction devant tenir sur une ligne) est trop longue pour tenir sur une seule ligne physique, on peut la prolonger en insérant le *caractère de continuation* blanc souligné (_), après une espace, à la fin de la première ligne, et ce qui est alors écrit sur la ligne suivante est considéré comme faisant partie de la même ligne.

```
Function somPremEnt(ByVal n As Integer) _
      As Integer
```

Le seul endroit où un caractère de continuation ne fonctionne pas de cette manière est à l'intérieur d'une chaîne de caractères (`string`), il est alors considéré comme du texte. Pour écrire une chaîne de caractères sur plusieurs lignes il faut utiliser l'opérateur de concaténation (&).

Structuration du texte

La structuration syntaxique des programmes VBA est assurée par un système de parenthésage : un mot particulier vient terminer chaque *instruction* ou bloc d'instructions.

En général le mot qui sert de parenthèse fermante est le même que celui qui sert de parenthèse ouvrante, précédé de `End`. Par exemple `End Function` vient terminer une fonction, commencée par `Function`, le mot `End Sub` vient terminer une procédure, commencée par `Sub`, le mot `End If` vient terminer une instruction d'alternative, commencée par `If`.

Mais il y a des exceptions à cette règle de formation. Par exemple `Next` vient terminer une boucle commencée par `For` et `Loop` vient terminer une boucle commencée par `Do`.

2.3 Fonctions en VBA

Il y a deux sortes de sous-programmes : les fonctions et les procédures. Une fonction définie par le programmeur renvoie un résultat, comme les fonctions prédéfinies du tableur. Une procédure a pour effet de modifier l'environnement d'exécution (par exemple colorier des cellules de la feuille de calcul).

Nous avons choisi d'écrire les premiers programmes uniquement sous forme fonctionnelle, l'écriture de procédures étant reportée au chapitre 4. Mais les règles et les conventions d'écriture - spécification, définition, utilisation - sont semblables pour les fonctions et les procédures.

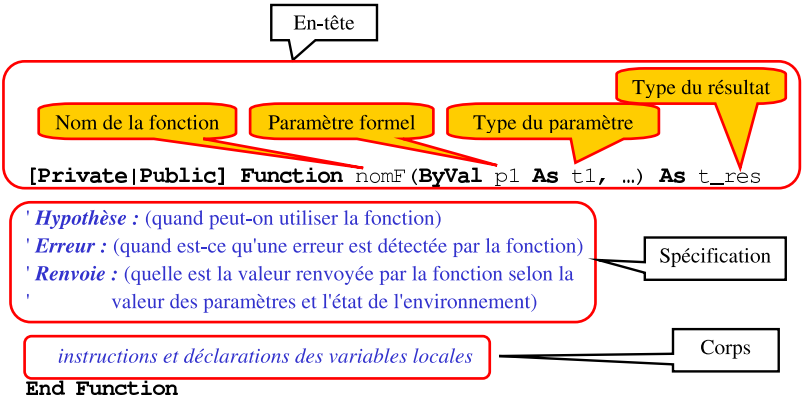
On explique tout d'abord dans cette section comment définir et utiliser une fonction. On étudiera ensuite, dans ce chapitre et le suivant, comment implanter les sous-programmes, c'est-à-dire construire le corps d'un sous-programme en enchaînant une suite d'instructions.

2.3.1 Définitions de fonctions

La syntaxe d'une définition de fonction décrit le nom, les arguments et le code formant le corps de cette fonction. On insiste aussi ici sur la spécification.

En-tête et spécification

Une définition de fonction s'écrit de la façon suivante :



L'en-tête de la fonction décrit la *signature* de la fonction : nom, type des paramètres et type du résultat :

- l'élément `Private` ou `Public` est facultatif [1] : une fonction privée n'est accessible qu'à l'intérieur du module où elle est définie, alors qu'une fonction publique est accessible de tous les modules (voir la notion de *module* dans la fiche technique VBA). La valeur par défaut est `Public`.
- le mot `Function` est suivi du nom de la fonction, ici `F`
- vient ensuite la liste des *paramètres formels* avec leurs types
  - chaque paramètre formel est précédé du mot `ByVal`, qui indique que l'on transmettra la *valeur* du paramètre d'appel, (il existe en VBA d'autres modes de transmission des paramètres, que l'on n'étudiera pas ici).
  - chaque paramètre formel a un nom, par exemple ici `p1`, et l'on indique aussi son type par `As` suivi du nom du type, ici `t1`.
  - les paramètres formels sont séparés par des virgules.
- enfin on indique le type du résultat de la fonction par `As` suivi du nom du type résultat, par exemple ici `t_res`.

Il reste à donner la partie *description* de la spécification de la fonction. Cette partie est présentée sous forme de commentaires à la suite de l'en-tête (ceci est une convention qui nous est propre).

- la ligne `Hypothèse` indique les conditions d'utilisation de la fonction
- la ligne `Erreur` signale les cas où une erreur est détectée par la fonction
- la ligne `Renvoie` décrit la valeur renvoyée par la fonction, selon la valeur des paramètres et l'état de l'environnement.

La ligne `Renvoie` est obligatoire dans toute description de fonction, mais les lignes `Hypothèse` et `Erreur` n'apparaissent que lorsqu'elles ont une

[1] par convention on note entre crochets les morceaux de code facultatifs

raison d'être.

Instruction de retour et sortie

Le corps de la fonction contient la déclaration des variables et les instructions. Parmi les instructions il doit **toujours** y avoir une *instruction de retour*, pour renvoyer le résultat de la fonction. Attention cette obligation n'est pas vérifiée par VBA. Pire, sans instruction de retour, la fonction retourne une valeur par défaut et il n'y a donc pas d'erreur détectée à l'exécution, mais en général cette valeur n'est pas la valeur attendue.

L'instruction de retour se note par le nom de la fonction, suivi du signe égal (=), suivi de l'expression dont la valeur doit être retournée en résultat : le nom de la fonction sert de variable pour contenir le résultat.

```
[Private|Public] Function F(ByVal p1 As t1) As t_res
            F = <expression> 'instruction de retour
End Function
```

La sortie d'une définition de fonction se fait en général en atteignant l'énoncé `End Function`. Le résultat renvoyé est alors le contenu de la variable du nom de la fonction.

On peut aussi sortir d'une fonction en atteignant l'instruction d'échappement `Exit Function`. Le résultat renvoyé est alors aussi le contenu de la variable du nom de la fonction.

On dissocie donc en VB la sortie de la fonction (par `Exit Function` ou en atteignant `End Function`) et la valeur renvoyée par la fonction (qui est la valeur contenue, au moment de la sortie, dans la variable nom de la fonction).

Exemple 1 : somme des n premiers entiers

On commence par écrire une fonction qui, étant donné un entier naturel n, renvoie la somme des n premiers entiers, en utilisant une formule de calcul. Le corps de la fonction est constitué d'une seule instruction : l'instruction de retour de la fonction, qui retourne la valeur de l'expression $n(n+1)/2$.

```
Function somPremEnt(ByVal n As Integer) As Integer
' Hypothèse : n >= 0
' Renvoie : la somme des n premiers entiers naturels
   somPremEnt = (n * (n + 1)) / 2
End Function
```

2.3.2 Appel de fonctions

Un appel de fonction est l'application d'une fonction à des *arguments* (on dit aussi *paramètres effectifs* ou *paramètres réels*) qui sont des expressions. Par exemple, l'appel `somPremEnt(3+2)` renvoie le résultat de

l'application de la fonction `somPremEnt` avec le paramètre effectif $3+2$ (dont la valeur est 5), c'est-à-dire que cet appel renvoie la valeur 15.

Un appel de fonction peut se faire soit à l'intérieur d'un programme VBA, soit directement dans une cellule de la feuille de calcul.

Dans un programme VBA

Un appel d'une fonction permet de former une expression dans le corps d'un programme. Dans l'exemple ci-dessous le corps de la fonction `sommeEnt` contient deux appels à la fonction `somPremEnt`.

Exemple 2 : somme des entiers de n à m

Pour écrire une fonction qui renvoie la somme $n+(n+1)+\ldots+m$ des entiers compris entre deux entiers naturels n et m, avec $n \leq m$, on calcule la différence entre la somme des m premiers entiers (résultat de l'appel le la fonction `somPremEnt` avec l'argument m) et la somme des $n-1$ premiers entiers (résultat de l'appel `somPremEnt(n-1)`).

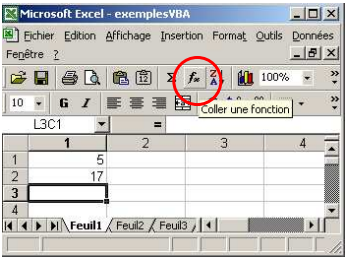
```
Function sommeEnt(ByVal n As Integer, ByVal m As Integer) _
                                                    As Integer
' Hypothèse : 0 < n <= m
' Renvoie : la somme des entiers compris entre n et m (inclus)
   sommeEnt = somPremEnt(m) - somPremEnt(n - 1)
End Function
```

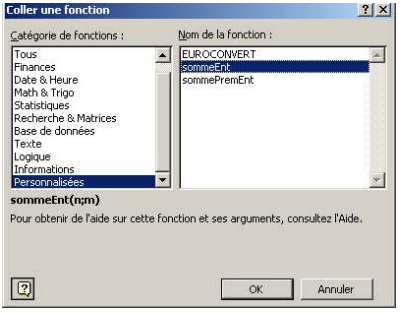
De même dans l'expression `2 * sommeEnt(10,20)/5`, l'appel de la fonction `sommeEnt` avec les paramètres effectifs 10 et 20 renvoie la valeur 165, et l'expression vaut donc finalement 66.

Dans l'exemple précédent, notez que les paramètres 10 et 20 sont séparés par une virgule : dans un appel de fonction à l'intérieur d'un programme VBA, les paramètres effectifs sont séparés par des virgules.

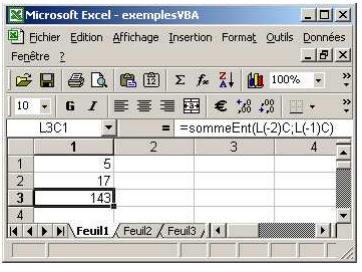
Dans la feuille de calcul

Pour tester les fonctions dans l'environnement `Excel` on peut appeler une fonction dans une case de la feuille de calcul. Cela peut se faire avec l'aide du « menu fonction » (entouré dans la figure de gauche ci-dessous). Dans la fenêtre de dialogue qui s'ouvre alors (figure de droite ci-dessous), on voit apparaître dans les « Fonctions personnalisées », la liste des fonctions définies par l'utilisateur, par exemple ici les fonctions `somPremEnt` et `sommeEnt`.





Lorque l'on sélectionne la fonction `sommeEnt`, il apparaît successivement plusieurs fenêtres permettant de donner les paramètres effectifs (ici on a sélectionné successivement la première et la deuxième case de la première colonne de la feuille de calcul, et ce sont les valeurs contenues dans ces cases qui constituent les paramètres effectifs). Finalement la formule correspondant à l'appel de fonction apparaît dans la barre de formule `=sommeEnt(L(-2)C;L(-1)C)`. Les paramètres effectifs sont notés par les références relatives des cellules qui les contiennent : le premier est situé sur la même colonne, deux lignes au dessus de la case qui contient le résultat de l'appel et le second sur la même colonne, une ligne au dessus.



Remarque : il faut noter que lorsqu'une fonction est appelée dans la feuille de calcul, les paramètres sont séparés par des points-virgules (et non des virgules). La raison de ce changement de notation est que dans le tableur on note les nombres décimaux à la française, avec des virgules (alors que c'est le point décimal anglo-saxon qui est utilisé en Visual Basic) et il faut donc différencier le séparateur de paramètres en le notant avec un point-virgule.

2.4 Constantes et variables

Constantes et variables sont des éléments permettant de stocker et de manipuler des données. Constantes et variables ont un nom, un type

et une valeur. Parmi les types manipulés par VBA, on retiendra dans un premier temps les nombres (`integer`, `long`, `double`, ...) , les chaînes de caractères (`string`) et les booléens (`boolean`).

Constantes et variables sont déclarées d'un certain type, ce qui permet de leur attribuer les caractéristiques pour stocker des données de ce type.

Une constante conserve une même valeur pendant toute l'exécution d'un programme, alors qu'une variable désigne un emplacement de stockage contenant des données pouvant être modifiées au cours de l'exécution du programme. L'instruction permettant de modifier la valeur d'une variable est l'affectation.

2.4.1 Notion de constante

Nature

Une constante est un nom pour une valeur calculée lors de l'analyse du programme en VBA. Noter qu'*on ne peut pas modifier la valeur d'une constante* : la valeur qui lui est affectée lors de sa déclaration reste la même pendant toute l'exécution du programme.

Déclaration des constantes en VBA

L'instruction `Const` permet de déclarer les constantes

– pour une constante locale à un sous-programme

```
Const nomCons As type  = <expr>
```

la déclaration explicite le nom `nomCons` à utiliser pour la valeur `expr`. En VBA décrire le type est facultatif (lorsque le type n'est pas déclaré explicitement, la constante se voit attribuer le type de données qui convient le mieux à la valeur `expr`), mais dans ce cours nous rendrons **obligatoires** toutes les déclarations de type.

– au niveau d'un module

```
Public Const nomCons As type = <expr>
[Private] Const nomCons As type = <expr>
```

Par défaut, les constantes sont privées.

Au niveau d'un sous-programme ou d'un module, on peut combiner plusieurs déclarations de constantes sur une même ligne, en séparant chaque déclaration par une virgule.

2.4.2 Notion de variable

Nature

Une variable associe une valeur à un nom. Dans un langage impératif, la valeur associée au nom peut évoluer au cours de l'exécution. Ceci est à opposer à ce qui se passe dans un langage purement fonctionnel où la valeur associée au nom est immuable.

Déclaration obligée

L'instruction `Option Explicit` est utilisée au niveau du module pour imposer la déclaration explicite (par l'instruction `Dim`) de toutes les variables de ce module.

Convention pédagogique : on mettra systématiquement cette instruction en première ligne de chaque module. Ainsi si l'on réfère un nom de variable non déclarée, une erreur se produit à la compilation.

Déclaration des variables

L'instruction `Dim` déclare des variables et attribue de l'espace de stockage.

- Locale à un sous-programme
  ```
  Dim nomVar As typeVar
  ```
- Au niveau du module
  ```
  [Private|Public] nomVar As typeVar
  ```
  Les variables déclarées au niveau module sont disponibles pour tous les sous-programmes du module.

2.4.3 Affectation des variables

Cette instruction permet d'affecter une valeur à une variable (et donc de modifier la valeur d'une variable).

Syntaxe

L'affectation se note par le symbole `=`, que l'on prononcera plutôt *reçoit* : la variable reçoit la valeur de l'expression.

```
nomVar = <expr>
```

Signification (sémantique)

Évaluer l'expression `<expr>` puis associer cette valeur à la variable `nomVar`.

Exemples

Considérons les exemples suivants

```
i = 1
'la variable entière i reçoit la valeur 1

b = 6 = 3 * 2
'la variable booléenne b reçoit la valeur vrai,
' résultat de l'évaluation de l'expression 6 = 3 * 2
```

Remarque : l'exemple précédent illustre différents sens du signe « = » symbole d'affectation ou symbole d'égalité dans les tests. (Et l'on a aussi vu un troisième sens du signe « = », instruction de retour de fonction.)

Différents sens d'une variable

Une variable associe une valeur (contenu) à un nom (contenant). Cependant on utilise indistinctement la variable pour désigner le contenu ou le contenant. Par exemple considérons l'affectation `i = i + 1`. Le symbole `i` à droite de l'affectation représente le contenu (pour évaluer l'expression `i + 1`, on ajoute `1` au contenu de la variable `i`), alors que le symbole `i` à gauche de l'affectation représente le contenant (on affecte au contenant nommé `i` le résultat de l'évaluation de l'expression précédente).

2.5 Structures de contrôle : séquence

Une instruction est un énoncé qui décrit une action ou un enchaînement d'actions. Les actions élémentaires sont l'évaluation d'un appel de procédure et l'affectation de variable. Les enchaînements d'instructions sont construits par les structures de contrôle : séquence, alternative et itération.

Les programmes sont *séquentiels* : les actions se suivent dans le temps et ne s'exécutent pas simultanément. Le *choix* permet de décider de l'action à exécuter en fonction de l'état de l'environnement d'éxécution ; et lorsque le choix est entre *deux* actions, on parle d'*alternative*. L'*itération* indique qu'il faut répéter un certain nombre de fois une suite d'actions ; le nombre de répétitions dépend de l'état de l'environnement d'éxécution, au travers d'une condition (condition de sortie dans les *boucles*).

Un programme est un morceau de texte qui décrit une suite d'actions, mais l'*ordre statique* de la description textuelle est en général différent de l'*ordre dynamique* dans lequel sont exécutées les actions, à cause des schémas de contrôle de choix et d'itérations. L'enchaînement dynamique est appelé *déroulement* du programme ou *processus*.

2.5.1 Syntaxe VBA

Le passage à la ligne sépare les instructions d'une séquence

```
Instruction_1
Instruction_2
```

Mais on peut aussi utiliser le symbole deux-points « : » comme séparateur d'instructions, la séquence précédente peut s'écrire de façon équivalente

```
Instruction_1 : Instruction_2
```

2.5.2 Sémantique

La séquence précédente signifie : exécuter l'instruction 1 puis exécuter l'instruction 2. Par exemple à l'issue de la séquence d'instructions

suvante, la variable m contient 5, p contient 8 et n contient 2.

```
n = 3 : m = 5
p = n + m : n = m - n
```

Attention à l'ordre des évaluations en séquence : si l'on modifie la séquence en

```
n = 3 : m = 5
n = m - n : p = n + m
```

à l'issue des affectations, m contient 5, n contient 2 et p contient 7.

2.5.3 Exemple : aire d'un triangle

Un théorème de la Grèce ancienne, que l'on attribue à Héron d'Alexandrie, mathématicien du premier siècle de notre ère, nous dit que l'aire d'un triangle de côtés a, b et c vaut $\sqrt{s(s-a)(s-b)(s-c)}$, avec $s=(a+b+c)/2$. Voici la fonction VBA correspondante :

```
Function aireTriangle(ByVal a As Double, ByVal b As Double, _
                                         ByVal c As Double) As Double
' Hypothèse : a, b et c sont positifs
' Renvoie : l'aire d'un triangle de cotés a, b et c
  Dim s As Double
  s = (a + b + c) / 2
  aireTriangle = Sqr(s * (s-a) * (s-b) * (s-c))
End Function
```

2.6 Structures de contrôle : alternative

2.6.1 Syntaxe VBA

La forme standard de l'alternative est

```
If exprbool Then
      instructions_1
Else
      instructions_2
End If
```

Mais on peut aussi la trouver sous la forme restreinte

```
If exprbool Then
      instructions
End If
```

ou sous la forme générale, avec imbrication sur l'alternant et un unique `End If` à la fin :

```
If exprbool_1 Then
      instructions_1
ElseIf exprbool_2 Then
      instructions_2
ElseIf ....
...
Else
      instructions_3
End If
```

2.6.2 Sémantique

La forme standard est un branchement à deux voies. Elle signifie :
- évaluer la condition (expression booléenne) `exprbool`,
- si `exprbool` vaut `vrai` exécuter `instructions_1` et sortir de l'alternative (c'est à dire continuer le programme en séquence, après le `End If`
- si `exprbool` vaut `faux` exécuter `instructions_2` et sortir de l'alternative.

Dans la forme restreinte, si `exprbool` vaut `faux` il n'y a aucune action à faire.

La forme générale correspond à un branchement selon un nombre de cas supérieur ou égal à trois : on examine successivement les expressions booléennes `exprbool_1`, `exprbool_2`, `exprbool_3`, ... au premier `exprbool_i` valant `vrai` on exécute `instructions_i` et on sort de l'alternative ; si aucune condition ne vaut `vrai` on exécute l'instruction qui suit le `Else` et on sort de l'alternative.

Attention, l'alternative est un schéma de contrôle et non une forme fonctionnelle : son évaluation modifie l'environnement de travail (par l'exécution d'instructions qui, par exemple, modifient l'affectation de certaines variables) mais ne renvoie pas une valeur.

2.6.3 Illustrations simples

Le morceau de programme suivant met dans *b* la valeur absolue de *a* :

```
If a < 0 Then
   b = - a
Else
   b = a
End If
```

Le suivant met dans *a* la valeur absolue de *a* :

```
If a < 0 Then
   a = - a
End If
```

Le dernier exemple met dans *b* la racine carrée de la valeur absolue de *a* (noter le commentaire sur la ligne du `Else`) :

```
If a < 0 Then
     b = sqrt(- a)
Elseif a > 0 Then
     b = sqrt(a)
Else 'si l'on passe dans cette branche de l'alternative, a vaut  0
     b = 0
End If
```

Remarque : en VBA la fin de ligne peut tenir lieu de fin d'instruction. Ainsi on peut écrire chacun des deux premiers exemples ci-dessus sur une seule ligne, et le mot `End If` peut alors être omis :

```
If a < 0 Then b = - a  Else b = a
```

```
If a < 0 Then a = - a
```

Cette écriture est valide mais, sauf pour un gain de place absolument nécessaire, elle est à éviter car elle risque de masquer la structuration.

2.6.4 Exemple : nombre de solutions d'une équation du second degré

On se propose d'écrire une fonction qui calcule le nombre de solutions réelles (on dit aussi *nombre de racines réelles*) d'une équation du second degré $ax^2+bx+c=0$. La méthode utilisée est classique : on calcule le discriminant $\Delta = b^2-4ac$ et trois cas peuvent se présenter :

- $\Delta < 0$: il n'y a pas de solution réelle
- $\Delta = 0$: il y a une solution réelle
- $\Delta < 0$: il y a deux solutions réelles

La fonction suivante implante cette méthode en examinant les 3 cas :

```
Function nbreSolutions(ByVal a As Double, ByVal b As Double, _
                       ByVal c As Double) As Integer
' Hypothèse : a non nul
' Renvoie : le nombre de solutions de l'équation ax^2 + bx + c = 0
   Dim delta As Double
   delta = b * b - 4 * a * c
   If delta < 0 Then
      nbreSolutions = 0
   ElseIf delta = 0 Then
      nbreSolutions = 1
   Else
      nbreSolutions = 2
   End If
End Function
```

Noter ici l'importance de la ligne d'hypothèse : si l'hypothèse n'est pas respectée, la fonction renvoie un résultat (1 pour $a=0$ et $b=0$, ce qui est incorrect, et 2 pour $a=0$ et $b\neq 0$, ce qui est aussi incorrect.

Deuxième implantation : traitement du cas a=0

Dans la fonction précédente on a pris comme hypothèse que a est non nul. On peut aussi traiter les cas où a est nul en se ramenant au nombre de solutions d'une équation du premier degré. On écrit d'abord la fonction `nbreSolutions1` concernant une équation linéaire $bx+c=0$: on doit considérer deux cas : $b=0$ et $b\neq 0$, et à l'intérieur du premier cas il faut aussi considérer deux cas : $c=0$ et $c\neq 0$.

```
Function nbreSolutions1(ByVal b As Double, ByVal c As Double) _
                                                    As Integer
' Renvoie : le nombre de solutions de l'équation bx + c = 0
' (renvoie -1 lorsque b=c=0, infinité de solutions)
   If b = 0 Then
```

```
        If c = 0 Then
           nbreSolutions1 = - 1
        Else 'ici c est différent de 0
           nbreSolutions1 = 0
        End If
     Else 'ici b est différent de 0
        nbreSolutions1 = 1
     End If
  End Function
```

Pour calculer le nombre de solutions d'une équation du second degré, on appelle alors la fonction `nbreSolutions1` si $a = 0$, et sinon on appelle la fonction initiale `nbreSolutions`, qui a pour hypothèse $a \neq 0$.

```
  Function nbreSolutions2(ByVal a As Double, ByVal b As Double, _
                          ByVal c As Double) As Integer
  ' Renvoie : le nombre de solutions de l'équation ax^2 + bx + c = 0
  ' (renvoie -1 lorsque a=b=c=0)
     If a = 0 Then
        nbreSolutions2 = nbreSolutions1(b, c)
     Else
        nbreSolutions2 = nbreSolutions(a, b, c)
     End If
  End Function
```

Noter que l'on a indiqué, dans la spécification de `nbreSolutions2`, que la fonction renvoie -1 lorsque les 3 paramètres sont nuls. Cette indication est importante pour une bonne utilisation de la fonction, elle doit donc apparaître dans sa spécification, même si ce résultat vient de l'appel de la fonction `nbreSolutions1`.

Troisième implantation : sortie par Exit

Dans la version suivante l'alternative ne contient pas de `Else` : dans le cas où $a = 0$ on appelle la fonction `nbreSolutions1` qui renvoie son résultat comme résultat de `nbreSolutions1` et on sort de la fonction. Si a ne vaut pas 0 on continue en séquence après le `End If` pour appeler la fonction `nbreSolutions`.

```
  Function nbreSolutions3(ByVal a As Double, ByVal b As Double, _
                          ByVal c As Double) As Integer
  ' Renvoie : le nombre de solutions de l'équation ax^2 + bx + c = 0
  ' (renvoie -1 lorsque a=b=c=0)
     If a = 0 Then
        nbreSolutions3 = nbreSolutions1(b, c)
        Exit Function
     End If
     nbreSolutions3 = nbreSolutions(a, b, c)
  End Function
```

Chapitre 3

Récursion et boucles

Le but de ce chapitre est d'étudier différentes façons de procéder pour répéter des opérations dans un programme.

3.1 Récursion

Avec la définition de fonction et l'alternative, que nous avons vues dans le chapitre précédent, il est, d'ores et déjà, possible d'écrire des fonctions récursives. La récursion est un mécanisme puissant qui permet d'exprimer la répétition d'opérations dans un programme. Nous nous contenterons ici de quelques exemples (pour étudier cette notion plus à fond, on pourra se reporter à l'ouvrage "Programmation récursive (en SCHEME)", *A. Brygoo, T. Durand, M. Pelletier, C. Queinnec, M. Soria*, Ed Dunod.).

3.1.1 Exemple 1 : définition récursive du reste

Soit la fonction `reste` qui renvoie le reste de la division d'un entier p, supérieur ou égal à zéro, par un entier q strictement positif. La relation de récurrence est :

- pour le cas où $p >= q$, $reste(p,q) = reste(p-q,q)$
- pour le cas où $p < q$ (dit « cas de base »), $reste(p,q) = p$.

De cette relation de récurrence on déduit une implantation de la fonction en la définition récursive suivante (l'implantation est dite « récursive » car dans la définition de la fonction il y a une référence, un appel, à elle-même).

Nous appelons cette fonction `resteRec`, et non `reste`, uniquement pour distinguer son implantation récursive d'une autre implantation, itérative `resteIter`, que nous verrons un peu plus loin. Les deux fonctions, `resteRec` et `resteIter` ont exactement la même spécification,

c'est-à-dire qu'elles renvoient le même résultat mais ne le calculent pas de la même manière (implantations différentes).

```
Function resteRec(ByVal p As Integer, _
                  ByVal q As Integer) As Integer
' Hypothèse : p >= 0 et q > 0
' Renvoie : le reste de la division de p par q
   If p >= q Then
      resteRec = resteRec(p - q, q)
   Else
      resteRec = p
   End If
End Function
```

3.1.2 Exemple 2 : définition récursive du quotient

Soit la fonction `quotient` qui renvoie le quotient de la division d'un entier p, supérieur ou égal à zéro, par un entier q strictement positif. La relation de récurrence est :

- pour le cas où $p >= q$, $quotient(p,q) = 1 + quotient(p-q,q)$
- pour le cas de base où $p < q$, $quotient(p,q) = 0$

On en déduit une implantation de la fonction `quotientRec` en la définition récursive suivante :

```
Function quotientRec(ByVal p As Integer, _
                     ByVal q As Integer) As Integer
' Hypothèse : p >= 0 et q > 0
' Renvoie : le quotient de la division de p par q
   If p >= q Then
      quotientRec = 1 + quotientRec(p - q, q)
   Else
      quotientRec = 0
   End If
End Function
```

3.2 Itération : boucle `For`

Pour itérer plusieurs fois la même séquence d'instructions (on dit aussi faire des itérations ou des boucles), Visual Basic (VB) offre plusieurs possibilités que l'on peut classer en boucle `For` et en boucle `Do`.

3.2.1 Syntaxe de la boucle `For`

Voici la syntaxe de la boucle `For` où les mots `For`, `To`, `Step` et `Next` sont quatre mots clefs, où `pas` est une constante entière et où `var` est une variable :

```
For var = expValDeb To expValFin [Step pas]
   Instructions
Next var
```

Par défaut, lorsque le mot clef `Step` n'apparaît pas, la valeur de *pas* est de 1.

3.2.2 Sémantique de la boucle For

Voici, lorsque Step est omis, la sémantique de la boucle For c'est-à-dire le sens que l'on attribue à son écriture ou encore la façon dont on doit lire cette instruction :
« Pour chaque valeur de la variable var, en partant de la valeur de début donnée par l'expression expValDeb jusqu'à la valeur de fin (comprise) donnée par l'expression expValFin, refaire les instructions qui se situent entre les mots clefs For et Next. La variable var est incrémentée (augmentée de 1) automatiquement à chaque passage de Next var ».

La variable var doit être déclarée avant la boucle et ne doit pas être modifiée par une des instructions situées entre les mots clefs For et Next car son incrémentation se fait automatiquement. Le « ne doit pas être modifiée » est à considérer comme une recommandation méthodologique et non comme une impossibilité technique, VBA ne vérifiant malheureusement pas cette obligation.

Si expValDeb vaut d et expValFin vaut f, les instructions sont exécutées pour var valant d, puis $d+1\ldots$, et enfin f. Elles sont donc effectuées $f-d+1$ fois. Les instructions ne sont pas exécutées lorsque $d > f$.

La sortie d'une boucle For se fait normalement quand la variable qui gère la boucle a dépassé la valeur de fin donnée par l'expression expValFin et, dans ce cas, l'instruction qui est ensuite effectuée est la première instruction qui suit Next var, on dit alors que « l'on sort de la boucle ». Il est possible aussi de sortir de la boucle sans que tous les tours de boucles initialement prévus se fassent, ce que nous verrons un peu plus loin dans ce chapitre.

Remarque 1 : lorsque, avant l'exécution d'une boucle For (sans partie Step), $f < d$, aucun tour de boucle n'est exécutée.

Remarque 2 : expValDeb et expValFin ne sont calculées qu'**une** fois.

3.2.3 Exemple 1 : boucle For pour la somme d'entiers

Voici un premier exemple d'utilisation de la boucle For dans une définition de la fonction sommeEnt (dont une autre définition, plus performante car nécessitant moins d'instructions, a déjà été donnée précédemment).

```
Function sommeEntBis(ByVal n As Integer, _
                     ByVal m As Integer) As Integer
' Renvoie : la somme des entiers compris entre n et m (inclus)
' (renvoie 0 lorsque n > m)
   Dim i As Integer
   Dim res As Integer
   res = 0
   For i = n To m
      res = res + i
```

```
    Next i
    sommeEntBis = res
End Function
```

3.2.4 Exemple 2 : boucle `For` pour le prédicat `estParfait`

Voici un deuxième exemple d'une boucle `For` utilisée dans un prédicat qui permet de savoir si un naturel est, ou non, un nombre parfait, c'est-à-dire s'il est égal, ou non, à la somme de ses diviseurs qui lui sont strictement inférieurs.

Par exemple 6 est parfait car $1+2+3=6$, de même que 28 car $1+2+4+7+14=28$.

Pour savoir si un naturel n est, ou non, un nombre parfait, nous allons, pour chaque entier (que l'on nomme par exemple i) compris entre 1 et la moitié de n, vérifier si cet entier i est un diviseur de n et, si c'est le cas, l'additionner à une variable *somme* qui est la somme de tous les diviseurs de n déjà rencontrés (et qui doit donc être initialisée à 0).

```
Function estParfait(ByVal n As Integer) As Boolean
' Hypothèse : n > 0
' Renvoie : True ssi n est un nombre parfait (i.e. s'il est égal
' à la somme de ses diviseurs qui lui sont strictement inférieurs)
    Dim i As Integer
    Dim somme As Integer
    somme = 0
    For i = 1 To n \ 2
        If n Mod i = 0 Then
            somme = somme + i
        End If
    Next i
    estParfait = somme = n
End Function
```

Remarque 1 : dans le code précédent le signe « \ » est l'opérateur de la division entière (alors que le signe « / » est l'opérateur de la division de nombres réels).

Remarque 2 : dans le code précédent, `n Mod i` permet d'avoir le reste de la division de `n` par `i`. Si ce reste est égal à zéro cela indique que `i` est diviseur de `n`.

Remarque 3 : dans le code précédent le signe « = » de « `somme = n` » permet de tester l'égalité entre `somme` et `n` et, le résultat de ce test est un booléen qui est affecté au résultat de la fonction par le « = » de « `estParfait =` ».

3.2.5 Boucle `For` avec un `Step`

Dans les exemples précédents de boucle `For`, sans que cela soit dit explicitement, à chaque tour de boucle, la variable qui gère la boucle augmente de 1. Mais il est possible d'indiquer explicitement la valeur de ce pas grâce au mot clef `Step` et à une expression qui donne le pas :

```
For var = expValDeb To expValFin Step pas
   Instructions
Next var
```

À chaque tour de boucle, la variable `var` prend la valeur `var + pas` et, lorsque `pas` est positif, la boucle s'arrête quand `var > expValFin`.

La constante `pas` peut être négative et dans ce cas, pour que les instructions soient exécutées, il faut que la valeur de `expValDeb` soit supérieure à la valeur de `expValFin`, sinon on n'entre pas dans la boucle.

L'exemple suivant donne une illustration de l'utilisation explicite d'un pas dans la boucle `For`.

```
Function sommeImpairs(ByVal n As Integer) As Integer
' Hypothèse : n > 0
' Renvoie : la somme des entiers impairs compris entre 1 et n (inclus)
   Dim i As Integer
   Dim res As Integer
   res = 0
   For i = 1 To n Step 2
      res = res + i
   Next i
   sommeImpairs = res
End Function
```

3.2.6 Sortie d'une boucle `For` par `Exit`

Dans le cas de la boucle `For`, on connaît, *a priori* le nombre de tours de boucle à effectuer. La sortie d'une boucle `For` se fait donc normalement quand le nombre de tours prévus a été effectué.

Mais, il est possible aussi de sortir de la boucle sans que tous les tours de boucle initialement prévus se fassent. Cela est rendu possible grâce à l'instruction `Exit For` qui, logiquement, doit se situer dans une alternative car, ainsi, en fonction du résultat de la condition de l'alternative, on sort ou non de la boucle :

```
For var = expValDeb To expValFin Step pas
   Instructions
     Exit For
   Instructions
Next var
```

Si `expValDeb` vaut d et `expValFin` vaut f, les instructions sont alors effectuées **au plus** $f-d+1$ fois.

Les trois exemples qui suivent illustrent cette sortie par l'instruction `Exit`.

Attention : l'expérience nous a montré que, si la boucle `For` sans instruction `Exit`, est habituellement rapidement comprise, par contre l'utilisation de la boucle `For` avec instruction `Exit` est source de nombreuses erreurs.

Ainsi, pour des raisons pédagogiques, nous vous conseillons, lors d'une première lecture, de sauter les trois exemples qui suivent pour étudier directement la boucle `Do` et de ne revenir à ces exemples qu'une fois les différentes possibilités de sortie de la boucle `Do` bien comprises.

3.2.7 Exemple 3 : boucle `For` pour `existeParfait`

Dans l'exemple qui suit, nous désirons écrire une fonction booléenne `existeParfait` qui renvoie `True` si et seulement si, il existe un nombre parfait compris entre deux entiers (n et m) donnés en arguments de la fonction.

La sortie de la boucle s'effectue dès qu'un nombre parfait a été trouvé entre n et m, juste après que la valeur de sortie de la fonction a été mise à `True` par l'instruction `existeParfait = True`.

Dans le cas où aucun nombre parfait n'existe entre n et m, la sortie de la boucle s'effectue normalement, une fois que les $m-n+1$ boucles ont été effectuées, et la sortie de la fonction se fait avec la valeur `False`, initialisée par l'instruction `existeParfait = False`, juste avant l'entrée dans la boucle.

```
Function existeParfait(ByVal n As Integer, _
                       ByVal m As Integer) As Boolean
' Hypothèse : n > 0 et m > 0
' Renvoie : True ssi il existe un nombre parfait entre n et m (inclus)
   Dim i As Integer
   existeParfait = False
   For i = n To m
      If estParfait(i) Then
         existeParfait = True
         Exit For
      End If
   Next i
End Function
```

Voici une autre implantation de la fonction `existeParfait` dans laquelle on sort directement de la fonction par `Exit Function` (et non plus seulement de la boucle par `Exit For`), dès qu'un nombre parfait a été trouvé.

L'instruction `existeParfaitBis = False` peut se mettre soit juste avant la boucle `For` (ici mise en commentaire) soit juste après la boucle `For`, puisque la sortie « normale » de la boucle `For` ne s'effectue que s'il n'existe pas de nombre parfait entre n et m.

```
Function existeParfaitBis(ByVal n As Integer, _
                          ByVal m As Integer) As Boolean
   Dim i As Integer
  ' existeParfaitBis = False
   For i = n To m
      If estParfait(i) Then
         existeParfaitBis = True
         Exit Function
      End If
   Next i
   existeParfaitBis = False
End Function
```

3.2.8 Exemple 4 : boucle For pour plusPetitParfait

Nous désirons écrire une fonction plusPetitParfait qui renvoie le plus petit nombre parfait compris entre n et m, s'il en existe un, et renvoie 0 sinon.

Dans cet exemple, lorsque l'on a trouvé un nombre parfait (qui se trouve être le plus petit), on sort de la boucle et, en même temps, de la fonction par une instruction Exit Function :

```
Function plusPetitParfait(ByVal n As Integer, _
                          ByVal m As Integer) As Integer
' Hypothèse : n > 0 et m > 0
' Renvoie : le plus petit nombre parfait entre n et m (inclus)
' s'il en existe un, 0 sinon
   Dim i As Integer
   For i = n To m
      If estParfait(i) Then
         plusPetitParfait = i
         Exit Function
      End If
   Next i
   plusPetitParfait = 0
End Function
```

3.2.9 Exemple 5 : boucle For pour plusGrandParfait

L'exemple suivant traite de la fonction plusGrandParfait qui renvoie le plus grand nombre parfait entre n et m, s'il en existe un, et renvoie 0 sinon.

Cette fonction est tout à fait similaire à la fonction précédente, mais illustre l'utilisation d'un pas négatif pour la boucle For.

```
Function plusGrandParfait(ByVal n As Integer, _
                          ByVal m As Integer) As Integer
' Hypothèse : n > 0 et m > 0
' Renvoie : le plus grand nombre parfait entre n et m (inclus)
' s'il en existe un, 0 sinon
   Dim i As Integer
   For i = m To n Step -1
      If estParfait(i) Then
         plusGrandParfait = i
         Exit Function
      End If
   Next i
   plusGrandParfait = 0
End Function
```

3.3 Itération : boucle Do

Dans le cas de la boucle For, on connaît *a priori* le nombre de tours de boucle, ou au moins le nombre maximum de tours de boucle. Dans le cas, où l'on ne connaît pas, *a priori*, le nombre de tours de boucle, il est préférable d'utiliser la boucle Do. Mais attention, puisque l'on ne connaît pas, *a priori*, le nombre de tours de boucle, il faut programmer avec soin le ou les tests d'arrêts de la boucle car, sinon, on risque de « boucler », c'est-à-dire que la même séquence d'instructions se reproduit indéfiniment.

3.3.1 Syntaxe de la boucle Do

La syntaxe de la boucle Do peut se décliner de multiples façons :

– On continue la boucle tant que la condition expBool est vraie :

```
Do While expBool
   Instructions
Loop
```

– On recommence la boucle tant que la condition expBool est vraie :

```
Do
   Instructions
Loop While expBool
```

– On sort en début de boucle lorsque la condition expBool est vraie, autrement dit, on continue la boucle jusqu'à ce que la condition expBool soit vraie :

```
Do Until expBool
   Instructions
Loop
```

– On sort en fin de boucle lorsque la condition expBool est vraie, autrement dit, on recommence la boucle jusqu'à ce que la condition expBool soit vraie :

```
Do
   Instructions
Loop Until expBool
```

Remarque 1 : lorsque la condition est en début de boucle, il y a 0 ou plus de tours de boucle, et lorsque la condition est en fin de boucle il y a 1 ou plus de tours de boucle.

Remarque 2 : dans une boucle Do, toute expression `Until expBool` peut être remplacée par l'expression `While not(expBool)` de même que toute expression `While expBool` peut être remplacée par l'expression `Until not(expBool)`. Il est effectivement équivalent de dire « je récris ma fonction tant qu'elle ne fonctionne pas » ou « je récris ma fonction jusqu'à ce qu'elle fonctionne ». Le choix entre les mots Until et While se fait en fonction de la préférence que l'on a pour une expression booléenne ou sa négation.

3.3.2 Exemple 1 : boucle Do pour le calcul du reste

On désire implanter le calcul du reste de la division de p par q, non plus de façon récursive comme cela a déjà été fait, mais de façon itérative, c'est-à-dire, en utilisant les boucles.

L'idée est d'itérer l'action de « retrancher q à p » jusqu'à ce que $p < q$. Lorsque, au bout d'un certain nombre de boucles (éventuellement 0), p devient inférieur à q, le résultat de la fonction est la valeur de p ainsi obtenue par retranchement successif de q.

Voici une implantation de cette fonction où l'on garde intacte la valeur de p initiale :

```
Function resteIterAux(ByVal p As Integer, _
                      ByVal q As Integer) As Integer
' Hypothèse : p >= 0 et q > 0
' Renvoie : le reste de la division de p par q
    Dim pAux As Integer
    pAux = p
    Do Until pAux < q
        pAux = pAux - q
    Loop
    resteIterAux = pAux
End Function
```

Voici aussi une autre implantation où l'on retranche directement à la variable p la valeur de q :

```
Function resteIter(ByVal p As Integer, _
                   ByVal q As Integer) As Integer
   Do Until p < q
      p = p - q
   Loop
   resteIter = p
End Function
```

Nous donnons, ci-dessous, une illustration de l'évolution du contenu des variables p et q lorsque la fonction a été appelée pour les valeurs de 43 et de 13 pour p et q. La première ligne du tableau correspond aux valeurs de p et q avant d'entrer dans la boucle puis les lignes suivantes correspondent aux valeurs de p et q juste avant le `Loop` :

p	q
43	13
30	13
17	13
4	13
On sort, résultat = 4	

3.3.3 Exemple 2 : boucle Do pour le calcul du quotient

On désire maintenant implanter le calcul du quotient de la division de p par q, non plus de façon récursive, comme cela a déjà été fait, mais de façon itérative.

Comme pour le calcul du reste, l'idée est d'itérer l'action de « retrancher q à p » jusqu'à ce que $p < q$ mais en comptant le nombre de fois où l'on retranche q. Pour cela il faut déclarer une variable pour compter (par exemple *compt*), l'initialiser (par l'instruction `compt = 0`) et incrémenter (augmenter de 1) le compteur à chaque tour de boucle (par l'instruction `compt = compt +1`) :

```
Function quotientIter(ByVal p As Integer, _
                      ByVal q As Integer) As Integer
' Hypothèse : p >= 0 et q > 0
' Renvoie : le quotient de la division de p par q
   Dim compt As Integer
   compt = 0
   Do Until p < q
      p = p - q
      compt = compt + 1
   Loop
   quotientIter = compt
End Function
```

Et voici une illustration de l'évolution du contenu des variables p, q et *compt* lorsque la fonction a été appelée pour les valeurs de 43 et de 13 pour p et q. La première ligne du tableau correspond aux valeurs de p, q et *compt* avant d'entrer dans la boucle puis les lignes suivantes correspondent aux valeurs de p, q et *compt* juste avant le `Loop` :

p	q	compt
43	13	0
30	13	1
17	13	2
4	13	3
On sort, résultat = 3		

3.3.4 Sortie d'une boucle `Do` par `Exit`

Dans les exemples que nous venons de voir, la sortie de boucle se fait soit en début soit en fin de boucle lorsque la condition `expBool` du `while` n'est plus vérifiée ou lorsque la condition `expBool` du `until` est enfin vérifiée. Mais en fait, la syntaxe de la boucle `Do` est encore plus générale :

```
Do [while expBool][until expBool]
   Instructions
     Exit Do
   Instructions
Loop [while expBool][until expBool]
```

Ainsi est-il possible de sortir de la boucle `Do`, de l'intérieur de la boucle, par l'instruction `Exit Do`. Cette instruction doit se trouver dans une alternative car, ainsi, en fonction du résultat de la condition de l'alternative, on sort ou non de la boucle.

3.3.5 Exemple 3 : boucle Do pour le calcul du pgcd

On désire implanter, de façon itérative, une fonction pgcd qui renvoie la plus grand commun diviseur de deux entiers positifs m et n.

On part pour cela des équations suivantes (qui permettent d'implanter facilement la fonction de façon récursive) :
$pgcd(m,0) = m$
$pgcd(0,n) = n$
$pgcd(m,n) = pgcd(m-n,n)$ si $m > n$
$pgcd(m,n) = pgcd(m,n-m)$ sinon

L'idée itérative que l'on en déduit est de retrancher le plus petit entier au plus grand et ce, jusqu'à ce qu'un des deux nombres soit nul.

On retrouve ainsi dans le corps de la fonction le test

```
If n > m
```

qui permet de traiter séparément les deux cas de $n > m$ puis de $n <= m$, puis les tests sur la nullité de m ou de n.

```
Function pgcd1iter(ByVal m As Integer, _
                   ByVal n As Integer) As Integer
' Hypothèse : m >= 0 et n >= 0
' Renvoie : le plus grand commun diviseur de m et de n
    Do ' pgcd(m_initial, n_initial) = pgcd(m, n)
      If n > m Then
         If m = 0 Then
            pgcd1iter = n: Exit Function
         Else
            n = n - m
         End If
      Else
         If n = 0 Then
            pgcd1iter = m
            Exit Function
         Else
            m = m - n
         End If
      End If
    Loop
End Function
```

Voici une illustration de l'évolution de contenu des variables m et n lorsque la fonction a été appelée pour $m = 12$ et $n = 30$. La première ligne du tableau correspond aux valeurs de m et n avant d'entrer dans la boucle puis les lignes suivantes correspondent aux valeurs de m et n juste avant le Loop :

m	n
12	30
12	18
12	6
6	6
0	6
On sort, résultat = 6	

Une autre implantation peut être obtenue en partant des équations suivantes (qui permettent aussi d'implanter d'une autre façon récursive la fonction) :
$pgcd(m,0) = m$
$pgcd(m,n) = pgcd(n, m \bmod n)$ si $n \neq 0$

```
Function pgcd2iter(ByVal m As Integer, _
                   ByVal n As Integer) As Integer
   Dim aux As Integer
   Do Until n = 0 ' pgcd(m_initial, n_initial) = pgcd(m, n)
   ' m, n <- n, m mod n :
      aux = n
      n = m Mod n
      m = aux
   Loop
   pgcd2iter = m
End Function
```

La variable `aux` permet de sauvegarder l'ancienne valeur de n pour l'affecter ensuite à m.

3.4 Choix entre différents types de boucle

En conclusion de ce chapitre, on récapitule quelques critères simples qui permettent, au programmeur débutant, de choisir un type de boucle pour une résolution itérative d'un problème donné.

Toute boucle `For` peut se traduire en boucle `Do`, ainsi la boucle :

```
For i = 1 To n
   Instructions
Next i
```

peut s'écrire de façon équivalente en :

```
i = 1
Dim naux as Integer
naux = n
Do While i <= naux
   Instructions
   i = i + 1
Loop
```

Mais notez bien que, dans la boucle `For`, l'initialisation et l'incrémentation de la variable i ainsi que le test $i <= n$ sont faits automatiquement

dans la gestion de la boucle par la machine, alors que dans la boucle `Do` cette initialisation, cette incrémentation ainsi que le test doivent être faits explicitement par le programmeur.

3.4.1 Choix entre une boucle `For` ou une boucle `Do`

Le choix, pour le programmeur, entre une boucle `For` ou une boucle `Do` se fait essentiellement sur sa connaissance ou non *a priori* du nombre de tours de boucle :

- on connaît le nombre de tours de boucle ou le nombre maximum de tours de boucle, on utilise de préférence la boucle `For` ;
- on ne connaît pas le nombre maximum de tours de boucle, on utilise la boucle `Do`.

3.4.2 Choix entre différentes boucles `Do`

Si, pour résoudre un problème, notre choix s'est porté sur la boucle `Do`, on doit s'assurer que l'on ne va pas boucler indéfiniment. Où doit-on faire le test de fin-continuation de boucle ? en début de boucle ? en fin de boucle ? à l'intérieur de la boucle ?

- Si le nombre de tours de boucle est un entier :
  - il y a zéro ou plus de tours de boucle, la condition est en début de boucle :

```
Do While expBool
   Instructions
Loop
```

    ou bien

```
Do Until Not(expBool)
   Instructions
Loop
```

  - il y a 1 ou plus de tours de boucle, la condition est en fin de boucle :

```
Do
   Instructions
Loop While expBool
```

    ou bien

```
Do
   Instructions
Loop Until Not(expBool)
```

- Si le nombre de tours de boucle n'est pas obligatoirement un entier :

```
Do [while expBool][until expBool]
   Instructions
     Exit Do
   Instructions
Loop [while expBool][until expBool]
```