

PARTIE 11 :

PROCEDURES ET FONCTIONS

« L'informatique semble encore chercher la recette miracle qui permettra aux gens d'écrire des programmes corrects sans avoir à réfléchir. Au lieu de cela, nous devons apprendre aux gens comment réfléchir » - Anonyme

1. Fonctions personnalisées

1.1 De quoi s'agit-il ?

Une application, surtout si elle est longue, a toutes les chances de devoir procéder aux mêmes traitements, ou à des traitements similaires, à plusieurs endroits de son déroulement. Par exemple, la saisie d'une réponse par oui ou par non (et le contrôle qu'elle implique), peuvent être répétés dix fois à des moments différents de la même application, pour dix questions différentes.

La manière la plus évidente, mais aussi la moins habile, de programmer ce genre de choses, c'est bien entendu de répéter le code correspondant autant de fois que nécessaire. Apparemment, on ne se casse pas la tête : quand il faut que la machine interroge l'utilisateur, on recopie les lignes de codes voulues en ne changeant que le nécessaire, et roule Raoul. Mais en procédant de cette manière, la pire qui soit, on se prépare des lendemains qui déchantent...

D'abord, parce que si la structure d'un programme écrit de cette manière peut paraître simple, elle est en réalité inutilement lourdingue. Elle contient des répétitions, et pour peu que le programme soit joufflu, il peut devenir parfaitement illisible. Or, le fait d'être facilement modifiable donc lisible, y

compris - et surtout - par ceux qui ne l'ont pas écrit est un critère essentiel pour un programme informatique ! Dès que l'on programme non pour soi-même, mais dans le cadre d'une organisation (entreprise ou autre), cette nécessité se fait sentir de manière aiguë. L'ignorer, c'est donc forcément grave.

En plus, à un autre niveau, une telle structure pose des problèmes considérables de maintenance : car en cas de modification du code, il va falloir traquer toutes les apparitions plus ou moins identiques de ce code pour faire convenablement la modification ! Et si l'on en oublie une, patatras, on a laissé un bug.

Il faut donc opter pour une autre stratégie, qui consiste à séparer ce traitement du corps du programme et à regrouper les instructions qui le composent en un module séparé. Il ne restera alors plus qu'à appeler ce groupe d'instructions (qui n'existe donc désormais qu'en un exemplaire unique) à chaque fois qu'on en a besoin. Ainsi, la lisibilité est assurée ; le programme devient **modulaire**, et il suffit de faire une seule modification au bon endroit, pour que cette modification prenne effet dans la totalité de l'application.

Le corps du programme s'appelle alors la **procédure principale**, et ces groupes d'instructions auxquels on a recours s'appellent des **fonctions** et des **sous-procédures** (nous verrons un peu plus loin la différence entre ces deux termes).

Reprenons un exemple de question à laquelle l'utilisateur doit répondre par oui ou par non.

Mauvaise Structure :

...

Ecrire "Etes-vous marié ?"

Rep1 ← ""

```
TantQue Rep1 <> "Oui" et Rep1 <> "Non"
```

```
  Ecrire "Tapez Oui ou Non"
```

```
  Lire Rep1
```

```
FinTantQue
```

```
...
```

```
Ecrire "Avez-vous des enfants ?"
```

```
Rep2 ← ""
```

```
TantQue Rep2 <> "Oui" et Rep2 <> "Non"
```

```
  Ecrire "Tapez Oui ou Non"
```

```
  Lire Rep2
```

```
FinTantQue
```

```
...
```

On le voit bien, il y a là une répétition quasi identique du traitement à accomplir. A chaque fois, on demande une réponse par Oui ou Non, avec contrôle de saisie. La seule chose qui change, c'est le nom de la variable dans laquelle on range la réponse. Alors, il doit bien y avoir un truc.

La solution, on vient de le voir, consiste à **isoler les instructions** demandant une réponse par Oui ou Non, et à appeler ces instructions à chaque fois que nécessaire. Ainsi, on évite les répétitions inutiles, et on a découpé notre problème en petits morceaux autonomes.

Nous allons donc créer une **fonction** dont le rôle sera de **renvoyer** la réponse (oui ou non) de l'utilisateur. Ce mot de "fonction", en l'occurrence, ne doit pas nous surprendre : nous avons étudié précédemment des fonctions fournies avec le langage, et nous avons vu que le but d'une fonction était de renvoyer une valeur. Eh bien, c'est exactement la même chose ici, sauf que c'est nous qui allons créer notre propre fonction, que nous appellerons RepOuiNon :

```
Fonction RepOuiNon() en caractère
```

```
Truc ← ""
```

```
TantQue Truc <> "Oui" et Truc <> "Non"
```

```
  Ecrire "Tapez Oui ou Non"
```

```
  Lire Truc
```

```
FinTantQue
```

```
Renvoyer Truc
```

```
Fin
```

On remarque au passage l'apparition d'un nouveau mot-clé : **Renvoyer**, qui indique quelle valeur doit prendre la fonction lorsqu'elle est utilisée par le programme. Cette valeur renvoyée par la fonction (ici, la valeur de la variable Truc) est en quelque sorte contenue dans le nom de la fonction lui-même, exactement comme c'était le cas dans les fonctions prédéfinies.

Une fonction s'écrit toujours **en-dehors de la procédure principale**. Selon les langages, cela peut prendre différentes formes. Mais ce qu'il faut comprendre, c'est que ces quelques lignes de codes sont en quelque sorte des satellites, qui existent en dehors du traitement lui-même. Simplement, elles sont à sa disposition, et il pourra y faire appel chaque fois que nécessaire. Si l'on reprend notre exemple, une fois notre fonction RepOuiNon écrite, le programme principal comprendra les lignes :

Bonne structure :

```
...
```

```
Ecrire "Etes-vous marié ?"
```

```
Rep1 ← RepOuiNon()
```

```
...
```

```
Ecrire "Avez-vous des enfants ?"
```

```
Rep2 ← RepOuiNon()
```

```
...
```

Et le tour est joué ! On a ainsi évité les répétitions inutiles, et si d'aventure, il y avait un bug dans notre contrôle de saisie, il suffirait de faire

une seule correction dans la fonction RepOuiNon pour que ce bug soit éliminé de toute l'application. Elle n'est pas belle, la vie ?

Toutefois, les plus sagaces d'entre vous auront remarqué, tant dans le titre de la fonction que dans chacun des appels, la présence de **parenthèses**. Celles-ci, dès qu'on déclare ou qu'on appelle une fonction, sont obligatoires. Et si vous avez bien compris tout ce qui précède, vous devez avoir une petite idée de ce qu'on va pouvoir mettre dedans...

1.2 Passage d'arguments

Reprenons l'exemple qui précède et analysons-le. On écrit un message à l'écran, puis on appelle la fonction RepOuiNon pour poser une question ; puis, un peu plus loin, on écrit un autre message à l'écran, et on appelle de nouveau la fonction pour poser la même question, etc. C'est une démarche acceptable, mais qui peut encore être améliorée : puisque avant chaque question, on doit écrire un message, autant que cette écriture du message figure directement dans la fonction appelée. Cela implique deux choses :

- lorsqu'on appelle la fonction, on doit lui préciser quel message elle doit afficher avant de lire la réponse
- la fonction doit être « prévenue » qu'elle recevra un message, et être capable de le récupérer pour l'afficher.

En langage algorithmique, on dira que **le message devient un argument (ou un paramètre) de la fonction**. Cela n'est certes pas une découverte pour vous : nous avons longuement utilisé les arguments à propos des fonctions prédéfinies. Eh bien, quitte à construire nos propres fonctions, nous pouvons donc construire nos propres arguments. Voilà comment l'affaire se présente...

La fonction sera dorénavant déclarée comme suit :

Fonction RepOuiNon(Msg en Caractère) en Caractère

Ecrire Msg

Truc ← ""

TantQue Truc <> "Oui" et Truc <> "Non"

Ecrire "Tapez Oui ou Non"

Lire Truc

FinTantQue

Renvoyer Truc

Fin Fonction

Il y a donc maintenant entre les parenthèses une variable, Msg, dont on précise le type, et qui signale à la fonction qu'un argument doit lui être envoyé à chaque appel. Quant à ces appels, justement, ils se simplifieront encore dans la procédure principale, pour devenir :

...

Rep1 ← RepOuiNon("Etes-vous marié ?")

...

Rep2 ← RepOuiNon("Avez-vous des enfants ?")

...

Et voilà le travail.

Une remarque importante : là, on n'a passé qu'un seul argument en entrée. Mais bien entendu, on peut en passer autant qu'on veut, et créer des fonctions avec deux, trois, quatre, etc. arguments ; Simplement, il faut éviter d'être gourmands, et il suffit de passer ce dont on en a besoin, ni plus, ni moins !

Dans le cas que l'on vient de voir, le passage d'un argument à la fonction était élégant, mais pas indispensable. La preuve, cela marchait déjà très bien avec la première version. Mais on peut imaginer des situations où il faut absolument concevoir la fonction de sorte qu'on doive lui transmettre un certain nombre d'arguments si l'on veut qu'elle puisse remplir sa tâche.

Prenons, par exemple, toutes les fonctions qui vont effectuer des calculs. Que ceux-ci soient simples ou compliqués, il va bien falloir envoyer à la fonction les valeurs grâce auxquelles elle sera censé produire son résultat (pensez tout bêtement à une fonction sur le modèle d'Excel, telle que celle qui doit calculer une somme ou une moyenne). C'est également vrai des fonctions qui traiteront des chaînes de caractères. Bref, dans 99% des cas, lorsqu'on créera une fonction, celle-ci devra comporter des arguments.

1.3 Deux mots sur l'analyse fonctionnelle

Comme souvent en algorithmique, si l'on s'en tient à la manière dont marche l'outil, tout cela n'est en réalité pas très compliqué. Les fonctions personnalisées se déduisent très logiquement de la manière nous nous avons déjà expérimenté les fonctions prédéfinies.

Le plus difficile, mais aussi le plus important, c'est d'acquérir le réflexe de **constituer systématiquement les fonctions adéquates quand on doit traiter un problème donné**, et de flairer la bonne manière de découper son algorithme en différentes fonctions pour le rendre léger, lisible et performant.

Cette partie de la réflexion s'appelle d'ailleurs l'**analyse fonctionnelle** d'un problème, et c'est toujours par elle qu'il faut commencer : en gros, dans un premier temps, on découpe le traitement en modules (algorithmique fonctionnelle), et dans un deuxième temps, on écrit chaque module (algorithmique classique). Cependant, avant d'en venir là, il nous faut découvrir deux autres outils, qui prennent le relais là où les fonctions deviennent incapables de nous aider.

2. Sous-Procédures

2.1 Généralités

Les fonctions, c'est bien, mais dans certains cas, ça ne nous rend guère service.

Il peut en effet arriver que dans un programme, on ait à réaliser des tâches répétitives, mais que ces tâches n'aient pas pour rôle de générer une valeur particulière, ou qu'elles aient pour rôle d'en générer plus d'une à la fois. Vous ne voyez pas de quoi je veux parler ? Prenons deux exemples.

Premier exemple. Imaginons qu'au cours de mon application, j'aie plusieurs fois besoin d'effacer l'écran et de réafficher un bidule comme un petit logo en haut à gauche. On pourrait se dire qu'il faut créer une fonction pour faire cela. Mais quelle serait la valeur renvoyée par la fonction ? Aucune ! Effacer l'écran, ce n'est pas produire un résultat stockable dans une variable, et afficher un logo non plus. Voilà donc une situation où j'ai besoin de répéter du code, mais où ce code n'a pas comme rôle de produire une valeur.

Deuxième exemple. Au cours de mon application, je dois plusieurs fois faire saisir un tableau d'entiers (mais à chaque fois, un tableau différent). Là encore, on serait tenté d'effectuer toutes ces saisies de tableaux dans une seule fonction. Mais problème, une fonction ne peut renvoyer qu'une seule valeur à la fois. Elle ne peut donc renvoyer un tableau, qui est une série de valeurs distinctes.

Alors, dans ces deux cas, faute de pouvoir traiter l'affaire par une fonction, devra-t-on en rester au code répétitif dont nous venons de dénoncer si vigoureusement les faiblesses ? Mmmmmh ? Vous vous doutez

bien que non. Heureusement, tout est prévu, il y a une solution. Et celle-ci consiste à utiliser des **sous-procédures**.

En fait, **les fonctions** - que nous avons vues - **ne sont finalement qu'un cas particulier des sous-procédures** - que nous allons voir : **celui où doit être renvoyé vers la procédure appelante une valeur et une seule**. Dans tous les autres cas (celui où on ne renvoie aucune valeur, comme celui où on en renvoie plusieurs), il faut donc avoir recours non à la forme particulière et simplifiée (la fonction), mais à la forme générale (la sous-procédure).

Parlons donc de ce qui est commun aux sous-procédures et aux fonctions, mais aussi de ce qui les différencie. Voici comment se présente une sous-procédure :

Procédure Bidule(...)

...

Fin Procédure

Dans la procédure principale, l'appel à la sous-procédure Bidule devient quant à lui :

Appeler Bidule(...)

Établissons un premier état des lieux.

- Alors qu'une fonction se caractérisait par les mots-clés **Fonction ... Fin Fonction**, une sous-procédure est identifiée par les mots-clés **Procédure ... Fin Procédure**. Oui, je sais, c'est un peu trivial comme remarque, mais, bon, on ne sait jamais.
- Lorsqu'une fonction était appelée, sa valeur (retournée) était toujours affectée à une variable (ou intégrée dans le calcul d'une expression). L'appel à une procédure, lui, est au contraire toujours une **instruction autonome**. "Exécute la procédure Bidule" est un ordre qui se suffit à lui-même.

- Toute fonction devait, pour cette raison, comporter l'instruction "Renvoyer". Pour la même raison, **l'instruction "Renvoyer" n'est jamais utilisée dans une sous-procédure**. La fonction est une valeur calculée, qui renvoie son résultat vers la procédure principale. La sous-procédure, elle, est un traitement ; elle ne "vaut" rien.
- Même une fois qu'on a bien compris les trois premiers points, on n'est pas complètement au bout de nos peines.

2.2 Le problème des arguments

En effet, il nous reste à examiner ce qui peut bien se trouver dans les parenthèses, à la place des points de suspension, aussi bien dans la déclaration de la sous-procédure que dans l'appel. Vous vous en doutez bien : c'est là que vont se trouver les outils qui vont permettre l'échange d'informations entre la procédure principale et la sous-procédure (en fait, cette dernière phrase est trop restrictive : mieux vaudrait dire : entre la procédure appelante et la procédure appelée. Car une sous-procédure peut très bien en appeler elle-même une autre afin de pouvoir accomplir sa tâche)

De même qu'avec les fonctions, les valeurs qui circulent depuis la procédure (ou la fonction) appelante vers la sous-procédure appelée se nomment des **arguments**, ou des **paramètres en entrée** de la sous-procédure. Comme on le voit, qu'il s'agisse des sous-procédure ou des fonctions, ces choses jouant exactement le même rôle (transmettre une information depuis le code donneur d'ordres jusqu'au code sous-traitant), elle portent également le même nom. Unique petite différence, on a précisé cette fois qu'il s'agissait d'arguments, ou de paramètres, **en entrée**. Pourquoi donc ?

Tout simplement parce que dans une sous-procédure, on peut être amené à vouloir renvoyer des résultats vers le programme principal ; or, là,

à la différence des fonctions, rien n'est prévu : la sous-procédure, en tant que telle, ne "renvoie" rien du tout (comme on vient de le voir, elle est d'ailleurs dépourvue de l'instruction "renvoyer"). Ces résultats que la sous-procédure doit transmettre à la procédure appelante devront donc eux aussi être véhiculés par des paramètres. Mais cette fois, il s'agira de paramètres fonctionnant dans l'autre sens (du sous-traitant vers le donneur d'ordres) : on les appellera donc des **paramètres en sortie**.

Ceci nous permet de reformuler en d'autres termes la vérité fondamentale apprise un peu plus haut : **toute sous-procédure possédant un et un seul paramètre en sortie peut également être écrite sous forme d'une fonction** (et entre nous, c'est une formulation préférable car un peu plus facile à comprendre et donc à retenir).

Jusque là, ça va ? Si oui, prenez un cachet d'aspirine et poursuivez la lecture. Si non, prenez un cachet d'aspirine et recommencez depuis le début. Et dans les deux cas, n'oubliez pas le grand verre d'eau pour faire passer l'aspirine.

Il nous reste un détail à examiner, détail qui comme vous vous en doutez bien, a une certaine importance : comment fait-on pour faire comprendre à un langage quels sont les paramètres qui doivent fonctionner en entrée et quels sont ceux qui doivent fonctionner en sortie...

2.3 Comment ça marche tout ça ?

En fait, si je dis qu'un paramètre est "en entrée" ou "en sortie", j'énonce quelque chose à propos de son rôle dans le programme. Je dis ce que je souhaite qu'il fasse, la manière dont je veux qu'il se comporte. Mais les programmes eux-mêmes n'ont cure de mes désirs, et ce n'est pas cette classification qu'ils adoptent. C'est toute la différence entre dire qu'une prise électrique sert à brancher un rasoir ou une cafetière (ce qui caractérise son rôle), et dire qu'elle est en 220 V ou en 110 V (ce qui

caractérise son type technique, et qui est l'information qui intéresse l'électricien). A l'image des électriciens, les langages se contrefichent de savoir quel sera le rôle (entrée ou sortie) d'un paramètre. Ce qu'ils exigent, c'est de connaître leur voltage... pardon, le **mode de passage** de ces paramètres. Il n'en existe que deux :

- le passage **par valeur**
- le passage **par référence**

...Voyons de plus près de quoi il s'agit.

Reprenons l'exemple que nous avons déjà utilisé plus haut, celui de notre fonction RepOuiNon. Comme nous l'avons vu, rien ne nous empêche de réécrire cette fonction sous la forme d'une procédure (puisque'une fonction n'est qu'un cas particulier de sous-procédure). Nous laisserons pour l'instant de côté la question de savoir comment renvoyer la réponse (contenue dans la variable Truc) vers le programme principal. En revanche, nous allons déclarer que Msg est un paramètre dont la transmission doit se faire par valeur. Cela donnera la chose suivante :

Procédure RepOuiNon(Msg en Caractère par valeur)

Ecrire Msg

Truc ← ""

TantQue Truc <> "Oui" et Truc <> "Non"

Ecrire "Tapez Oui ou Non"

Lire Truc

FinTantQue

??? Comment transmettre Truc à la procédure appelante ???

Fin Procédure

Quant à l'appel à cette sous-procédure, il pourra prendre par exemple cette forme :

```
M ← "Êtes-vous marié ?"  
Appeler RepOuiNon(M)
```

Que va-t-il se passer ?

Lorsque le programme principal arrive sur la première ligne, il affecte la variable M avec le libellé "Êtes-vous marié". La ligne suivante déclenche l'exécution de la sous-procédure. Celle-ci crée aussitôt une variable Msg. Celle-ci ayant été déclarée comme un paramètre passé **par valeur**, Msg va être affecté avec le même contenu que M. Cela signifie que **Msg est dorénavant une copie de M**. Les informations qui étaient contenues dans M ont été intégralement recopiées (en double) dans Msg. Cette copie subsistera tout au long de l'exécution de la sous-procédure RepOuiNon et sera détruite à la fin de celle-ci.

Une conséquence essentielle de tout cela est que si d'aventure la sous-procédure RepOuiNon contenait une instruction qui modifiait le contenu de la variable Msg, cela n'aurait aucune espèce de répercussion sur la procédure principale en général, et sur la variable M en particulier. **La sous-procédure ne travaillant que sur une copie de la variable qui a été fournie par le programme principal, elle est incapable, même si on le souhaitait, de modifier la valeur de celle-ci.** Dit d'une autre manière, dans une procédure, **un paramètre passé par valeur ne peut être qu'un paramètre en entrée.**

C'est en même temps une limite (aggravée par le fait que les informations ainsi recopiées occupent dorénavant deux fois plus de place en mémoire) et une sécurité : quand on transmet un paramètre par valeur, on est sûr et certain que même en cas de bug dans la sous-procédure, la valeur de la variable transmise ne sera jamais modifiée par erreur (c'est-à-dire écrasée) dans le programme principal.

Admettons à présent que nous déclarions un second paramètre, Truc, en précisant cette fois qu'il sera transmis **par référence**. Et adoptons pour la procédure l'écriture suivante :

Procédure RepOuiNon(Msg **en Caractère par valeur**, Truc **en Caractère par référence**)

Ecrire Msg

Truc ← ""

TantQue Truc <> "Oui" et Truc <> "Non"

Ecrire "Tapez Oui ou Non"

Lire Truc

FinTantQue

Fin Fonction

L'appel à la sous-procédure deviendrait par exemple :

M ← "Etes-vous marié ?"

Appeler RepOuiNon(M, T)

Ecrire "Votre réponse est ", T

Dépiautons le mécanisme de cette nouvelle écriture. En ce qui concerne la première ligne, celle qui affecte la variable M, rien de nouveau sous le soleil. Toutefois, l'appel à la sous-procédure provoque deux effets très différents. Comme on l'a déjà dit, la variable Msg est créée et immédiatement affectée avec une copie du contenu de M, puisqu'on a exigé un passage par valeur. Mais en ce qui concerne Truc, il en va tout autrement. Le fait qu'il s'agisse cette fois d'un passage par référence fait que **la variable Truc ne contiendra pas la valeur de T, mais son adresse, c'est-à-dire sa référence.**

Dès lors, **toute modification de Truc sera immédiatement redirigée, par ricochet en quelque sorte, sur T.** Truc n'est pas une variable ordinaire : elle ne contient pas de valeur, mais seulement la référence à une valeur, qui elle, se trouve ailleurs (dans la variable T). Il s'agit donc d'un

genre de variable complètement nouveau, et différent de ce que nous avons vu jusque là. Ce type de variable porte un nom : on l'appelle **un pointeur**. Tous les paramètres passés par référence sont des pointeurs, mais les pointeurs ne se limitent pas aux paramètres passés par référence (même si ce sont les seuls que nous verrons dans le cadre de ce cours). Il faut bien comprendre que ce type de variable étrange est géré directement par les langages : **à partir du moment où une variable est considérée comme un pointeur, toute affectation de cette variable se traduit automatiquement par la modification de la variable sur laquelle elle pointe.**

Passer un paramètre par référence, cela présente donc deux avantages. Et d'une, on gagne en occupation de place mémoire, puisque le paramètre en question ne recopie pas les informations envoyées par la procédure appelante, mais qu'il se contente d'en noter l'adresse. Et de deux, **cela permet d'utiliser ce paramètre tant en lecture (en entrée) qu'en écriture (en sortie)**, puisque toute modification de la valeur du paramètre aura pour effet de modifier la variable correspondante dans la procédure appelante.

Nous pouvons résumer tout cela par un petit tableau :

	passage par valeur	passage par référence
utilisation en entrée	oui	oui
utilisation en sortie	non	oui

Mais alors, demanderez-vous dans un élan de touchante naïveté, si le passage par référence présente les deux avantages présentés il y a un instant, pourquoi ne pas s'en servir systématiquement ? Pourquoi

s'embêter avec les passages par valeur, qui non seulement utilisent de la place en mémoire, mais qui de surcroît nous interdisent d'utiliser la variable comme un paramètre en sortie ?

Eh bien, justement, parce qu'on ne pourra pas utiliser comme paramètre en sortie, et que cet inconvénient se révèle être aussi, éventuellement, un avantage. Disons la chose autrement : **c'est une sécurité**. C'est la garantie que quel que soit le bug qui pourra affecter la sous-procédure, ce bug ne viendra jamais mettre le foutoir dans les variables du programme principal qu'elle ne doit pas toucher. Voilà pourquoi, lorsqu'on souhaite définir un paramètre dont on sait qu'il fonctionnera exclusivement en entrée, il est sage de le verrouiller, en quelque sorte, en le définissant comme passé par valeur. Et Lycée de Versailles, ne seront définis comme passés par référence que les paramètres dont on a absolument besoin qu'ils soient utilisés en sortie.

3. Variables publiques et privées

Résumons la situation. Nous venons de voir que nous pouvions découper un long traitement comportant éventuellement des redondances (notre application) en différents modules. Et nous avons vu que les informations pouvaient être transmises entre ces modules selon deux modes :

- si le module appelé est une fonction, par le **retour du résultat**
- dans tous les cas, par la **transmission de paramètres** (que ces paramètres soient passés par valeur ou par référence)

En fait, il existe un troisième et dernier moyen d'échanger des informations entre différentes procédures et fonctions : c'est de ne pas avoir besoin de les échanger, en faisant en sorte que ces procédures et fonctions partagent littéralement les mêmes variables. Cela suppose d'avoir

recours à des variables particulières, lisibles et utilisables par n'importe quelle procédure ou fonction de l'application.

Par défaut, une variable est déclarée au sein d'une procédure ou d'une fonction. Elle est donc créée avec cette procédure, et disparaît avec elle. Durant tout le temps de son existence, une telle variable n'est visible que par la procédure qui l'a vu naître. Si je crée une variable Toto dans une procédure Bidule, et qu'en cours de route, ma procédure Bidule appelle une sous-procédure Machin, il est hors de question que Machin puisse accéder à Toto, ne serait-ce que pour connaître sa valeur (et ne parlons pas de la modifier). Voilà pourquoi ces variables par défaut sont dites **privées**, ou **locales**.

Mais à côté de cela, il est possible de créer des variables qui certes, seront déclarées dans une procédure, mais qui du moment où elles existeront, seront des variables communes à toutes les procédures et fonctions de l'application. Avec de telles variables, le problème de la transmission des valeurs d'une procédure (ou d'une fonction) à l'autre ne se pose même plus : la variable Truc, existant pour toute l'application, est accessible et modifiable depuis n'importe quelle ligne de code de cette application. Plus besoin donc de la transmettre ou de la renvoyer. Une telle variable est alors dite **publique**, ou **globale**.

La manière dont la déclaration d'une variable publique doit être faite est évidemment fonction de chaque langage de programmation. En pseudo-code algorithmique, on pourra utiliser le mot-clé **Publique** :

Variable Publique Toto en Numérique

Alors, pourquoi ne pas rendre toutes les variables publiques, et s'épargner ainsi de fastidieux efforts pour passer des paramètres ? C'est très simple, et c'est toujours la même chose : **les variables globales consomment énormément de ressources en mémoire**. En conséquence, le principe qui doit présider au choix entre variables publiques et privées

doit être celui de **l'économie de moyens** : on ne déclare comme publiques que les variables qui doivent absolument l'être. Et chaque fois que possible, lorsqu'on crée une sous-procédure, on utilise le passage de paramètres plutôt que des variables publiques.

4. peut-on tout faire ?

A cette question, la réponse est bien évidemment : oui, on peut tout faire. Mais c'est précisément la raison pour laquelle on peut vite en arriver à faire aussi absolument n'importe quoi.

N'importe quoi, c'est quoi ? C'est par exemple, comme on vient de le voir, mettre des variables globales partout, sous prétexte que c'est autant de paramètres qu'on n'aura pas à passer.

Mais on peut imaginer d'autres atrocités.

Par exemple, une fonction, dont un des paramètres d'entrée serait passé par référence, et modifié par la fonction. Ce qui signifierait que cette fonction produirait non pas un, mais deux résultats. Autrement dit, que sous des dehors de fonctions, elle se comporterait en réalité comme une sous-procédure.

Ou inversement, on peut concevoir une procédure qui modifierait la valeur d'un paramètre (et d'un seul) passé par référence. Il s'agirait là d'une procédure qui en réalité, serait une fonction. Quoique ce dernier exemple ne soit pas d'une gravité dramatique, il participe de la même logique consistant à embrouiller le code en faisant passer un outil pour un autre, au lieu d'adopter la structure la plus claire et la plus lisible possible.

Enfin, il ne faut pas écarter la possibilité de programmeurs particulièrement vicieux, qui par un savant mélange de paramètres passés par référence, de variables globales, de procédures et de fonctions mal

choisies, finiraient par accoucher d'un code absolument illogique, illisible, et dans lequel la chasse à l'erreur relèverait de l'exploit.

Trèfle de plaisanteries : le principe qui doit guider tout programmeur est celui de la solidité et de la clarté du code. **Une application bien programmée est une application à l'architecture claire, dont les différents modules font ce qu'ils disent, disent ce qu'il font, et peuvent être testés (ou modifiés) un par un sans perturber le reste de la construction.** Il convient donc :

1. de **limiter au minimum l'utilisation des variables globales.** Celles-ci doivent être employées avec nos célèbres amis italo-arméniens, c'est-à-dire avec parcimonie et à bon escient.
2. de **regrouper sous forme de modules distincts** tous les morceaux de code qui possèdent une certaine unité fonctionnelle (programmation par "blocs"). C'est-à-dire de faire la chasse aux lignes de codes redondantes, ou quasi-redondantes.
3. de faire de ces modules **des fonctions lorsqu'ils renvoient un résultat unique**, et **des sous-procédures dans tous les autres cas** (ce qui implique de ne **jamais** passer un paramètre par référence à une fonction : soit on n'en a pas besoin, soit on en a besoin, et ce n'est alors plus une fonction).

Respecter ces règles d'hygiène est indispensable si l'on veut qu'une application ressemble à autre chose qu'au palais du facteur Cheval. Car une architecture à laquelle on ne comprend rien, c'est sans doute très poétique, mais il y a des circonstances où l'efficacité est préférable à la poésie. Et, pour ceux qui en douteraient encore, la programmation informatique fait (hélas ?) partie de ces circonstances.

5. ALGORITHMES FONCTIONNELS

Pour clore ce chapitre, voici quelques mots supplémentaires à propos de la structure générale d'une application. Comme on l'a dit à plusieurs

reprises, celle-ci va couramment être formée d'une procédure principale, et de fonctions et de sous-procédures (qui vont au besoin elles-mêmes en appeler d'autres, etc.). L'exemple typique est celui d'un menu, ou d'un sommaire, qui « branche » sur différents traitements, donc différentes sous-procédures.

L'algorithme fonctionnel de l'application est le **découpage** et/ou la **représentation graphique** de cette structure générale, ayant comme objectif de faire comprendre d'un seul coup d'œil quelle procédure fait quoi, et quelle procédure appelle quelle autre. L'algorithme fonctionnel est donc en quelque sorte la construction du squelette de l'application. Il se situe à un niveau plus général, plus abstrait, que l'algorithme normal, qui lui, détaille pas à pas les traitements effectués au sein de chaque procédure.

Dans la construction – et la compréhension – d'une application, les deux documents sont indispensables, et constituent deux étapes successives de l'élaboration d'un projet. La troisième – et dernière – étape, consiste à écrire, pour chaque procédure et fonction, l'algorithme détaillé.

Exemple de réalisation d'un algorithme fonctionnel : Le Jeu du Pendu

Vous connaissez tous ce jeu : l'utilisateur doit deviner un mot choisi au hasard par l'ordinateur, en un minimum d'essais. Pour cela, il propose des lettres de l'alphabet. Si la lettre figure dans le mot à trouver, elle s'affiche. Si elle n'y figure pas, le nombre des mauvaises réponses augmente de 1. Au bout de dix mauvaises réponses, la partie est perdue.

Ce petit jeu va nous permettre de mettre en relief les trois étapes de la réalisation d'un algorithme un peu complexe ; bien entendu, on pourrait toujours ignorer ces trois étapes, et se lancer comme un dératé directement dans la gueule du loup, à savoir l'écriture de l'algorithme définitif. Mais, sauf à être particulièrement doué, mieux vaut respecter le canevas qui suit, car les difficultés se résolvent mieux quand on les saucissonne...

Etape 1 : le dictionnaire des données

Le but de cette étape est d'identifier les informations qui seront nécessaires au traitement du problème, et de choisir le type de codage qui sera le plus satisfaisant pour traiter ces informations. C'est un moment essentiel de la réflexion, qu'il ne faut surtout pas prendre à la légère... Or, neuf programmeurs débutants sur dix bâclent cette réflexion, quand ils ne la zappent pas purement et simplement. La punition ne se fait généralement pas attendre longtemps ; l'algorithme étant bâti sur de mauvaises fondations, le programmeur se rend compte tout en l'écrivant que le choix de codage des informations, par exemple, mène à des impasses. La précipitation est donc punie par le fait qu'on est obligé de tout reprendre depuis le début, et qu'on a au total perdu bien davantage de temps qu'on en a cru en gagner...

Donc, avant même d'écrire quoi que ce soit, les questions qu'il faut se poser sont les suivantes :

- de quelles informations le programme va-t-il avoir besoin pour venir à bout de sa tâche ?
- pour chacune de ces informations, quel est le meilleur codage ? Autrement dit, celui qui sans gaspiller de la place mémoire, permettra d'écrire l'algorithme le plus simple ?

Encore une fois, il ne faut pas hésiter à passer du temps sur ces questions, car certaines erreurs, ou certains oublis, se payent cher par la suite. Et inversement, le temps investi à ce niveau est largement rattrapé au moment du développement proprement dit.

Pour le jeu du pendu, voici la liste des informations dont on va avoir besoin :

- une liste de mots (si l'on veut éviter que le programme ne propose toujours le même mot à trouver, ce qui risquerait de devenir assez rapidement lassant...)
- le mot à deviner
- la lettre proposée par le joueur à chaque tour
- le nombre actuel de mauvaises réponses
- et enfin, last but not least, l'ensemble des lettres déjà trouvées par le joueur. Cette information est capitale ; le programme en aura besoin au moins pour deux choses : d'une part, pour savoir si le mot entier a été trouvé. D'autre part, pour afficher à chaque tour l'état actuel du mot (je rappelle qu'à chaque tour, les lettres trouvées sont affichées en clair par la machine, les lettres restant à deviner étant remplacées par des tirets).
- à cela, on pourrait ajouter une liste comprenant l'ensemble des lettres déjà proposées par le joueur, qu'elles soient correctes ou non ; ceci permettra d'interdire au joueur de proposer à nouveau une lettre précédemment jouée.

Cette liste d'informations n'est peut-être pas exhaustive ; nous aurons vraisemblablement besoin au cours de l'algorithme de quelques variables supplémentaires (des compteurs de boucles, des variables temporaires, etc.). Mais les informations essentielles sont bel et bien là. Se pose maintenant le problème de choisir le mode de codage le plus futé. Si, pour certaines informations, la question va être vite réglée, pour d'autres, il va falloir faire des choix (et si possible, des choix intelligents !). C'est parti, mon kiki :

- Pour la liste des mots à trouver, il s'agit d'un ensemble d'informations de type alphanumérique. Ces informations pourraient faire partie du corps de la procédure principale, et être ainsi stockées en mémoire vive, sous la forme d'un tableau de chaînes. Mais ce n'est

certainement pas le plus judicieux. Toute cette place occupée risque de peser lourd inutilement, car il n'y a aucun intérêt à stocker l'ensemble des mots en mémoire vive. Et si l'on souhaite enrichir la liste des mots à trouver, on sera obligé de réécrire des lignes de programme... Conclusion, la liste des mots sera bien plus à sa place dans un fichier texte, dans lequel le programme ira piocher un seul mot, celui qu'il faudra trouver. Nous constituerons donc un fichier texte, appelé dico.txt, dans lequel figurera un mot par ligne (par enregistrement).

- Le mot à trouver, lui, ne pose aucun problème : il s'agit d'une information simple de type chaîne, qui pourra être stocké dans une variable appelée mot, de type caractère.
- De même, la lettre proposée par le joueur est une information simple de type chaîne, qui sera stockée dans une variable appelée lettre, de type caractère.
- Le nombre actuel de mauvaises réponses est une information qui pourra être stockée dans une variable numérique de type entier simple appelée MovRep.
- L'ensemble des lettres trouvées par le joueur est typiquement une information qui peut faire l'objet de plusieurs choix de codage ; rappelons qu'au moment de l'affichage, nous aurons besoin de savoir pour chaque lettre du mot à deviner si elle a été trouvée ou non. Une première possibilité, immédiate, serait de disposer d'une chaîne de caractères comprenant l'ensemble des lettres précédemment trouvées. Cette solution est loin d'être mauvaise, et on pourrait tout à fait l'adopter. Mais ici, on fera une autre choix, ne serait-ce que pour varier les plaisirs : on va se doter d'un tableau de booléens, comptant autant d'emplacements qu'il y a de lettres dans le mot à deviner. Chaque emplacement du tableau correspondra à une lettre du mot à trouver, et indiquera par sa valeur si la lettre a été découverte ou non

(faux, la lettre n'a pas été devinée, vrai, elle l'a été). La correspondance entre les éléments du tableau et le mot à deviner étant immédiate, la programmation de nos boucles en sera facilitée. Nous baptiserons notre tableau de booléens du joli nom de « verif ».

- Enfin, l'ensemble des lettres proposées sera stockée sans soucis dans une chaîne de caractères nommée `Propos`.

Nous avons maintenant suffisamment gambergé pour dresser le tableau final de cette étape, à savoir le dictionnaire des données proprement dit :

.

Nom	Type	Description
Dico.txt	Fichier texte	Liste des mots à deviner
Mot	Caractère	Mot à deviner
Lettre	Caractère	Lettre proposée
MovRep	Entier	Nombre de mauvaises réponses
Verif()	Tableau de Booléens	Lettres précédemment devinées, en correspondance avec Mot
Propos	Caractère	Liste des lettres proposées

Etape 2 : l'algorithme fonctionnel

On peut à présent passer à la réalisation de l'algorithme fonctionnel, c'est-à-dire au découpage de notre problème en blocs logiques. Le but de la manœuvre est multiple :

- faciliter la réalisation de l'algorithme définitif en le tronçonnant en plus petits morceaux.
- Gagner du temps et de la légèreté en isolant au mieux les sous-procédures et fonctions qui méritent de l'être. Eviter ainsi éventuellement des répétitions multiples de code au cours du

programme, répétitions qui ne diffèrent les unes des autres qu'à quelques variantes près.

- Permettre une division du travail entre programmeurs, chacun se voyant assigner la programmation de sous-procédures ou de fonctions spécifiques (cet aspect est essentiel dès qu'on quitte le bricolage personnel pour entrer dans le monde de la programmation professionnelle, donc collective).

Dans notre cas précis, un premier bloc se détache : il s'agit de ce qu'on pourrait appeler les préparatifs du jeu (choix du mot à deviner). Puisque le but est de renvoyer une valeur et une seule (le mot choisi par la machine), nous pouvons confier cette tâche à une fonction spécialisée `ChoixDuMot` (à noter que ce découpage est un choix de lisibilité, et pas une nécessité absolue ; on pourrait tout aussi bien faire cela dans la procédure principale).

Cette procédure principale, justement, va ensuite avoir nécessairement la forme d'une boucle `Tantque` : en effet , tant que la partie n'est pas finie, on recommence la série des traitements qui représentent un tour de jeu. Mais comment, justement, savoir si la partie est finie ? Elle peut se terminer soit parce que le nombre de mauvaises réponses a atteint 10, soit parce que toutes les lettres du mot ont été trouvées. Le mieux sera donc de confier l'examen de tout cela à une fonction spécialisée, `PartieFinie`, qui renverra une valeur numérique (0 pour signifier que la partie est en cours, 1 en cas de victoire, 2 en cas de défaite).

Passons maintenant au tour de jeu.

La première chose à faire, c'est d'afficher à l'écran l'état actuel du mot à deviner : un mélange de lettres en clair (celles qui ont été trouvées) et de tirets (correspondant aux lettres non encore trouvées). Tout ceci pourra être pris en charge par une sous-procédure spécialisée, appelée

AffichageMot. Quant à l'initialisation des différentes variables, elle pourra être placée, de manière classique, dans la procédure principale elle-même.

Ensuite, on doit procéder à la saisie de la lettre proposée, en veillant à effectuer les contrôles de saisie adéquats. Là encore, une fonction spécialisée, SaisieLettre, sera toute indiquée.

Une fois la proposition faite, il convient de vérifier si elle correspond ou non à une lettre à deviner, et à en tirer les conséquences. Ceci sera fait par une sous-procédure appelée VérifLettre.

Enfin, une fois la partie terminée, on doit afficher les conclusions à l'écran ; on déclare à cet effet une dernière procédure, FinDePartie.

Nous pouvons, dans un algorithme fonctionnel complet, dresser un tableau des différentes procédures et fonctions, exactement comme nous l'avons fait juste avant pour les données (on s'épargnera cette peine dans le cas présent, ce que nous avons écrit ci-dessus suffisant amplement. Mais dans le cas d'une grosse application, un tel travail serait nécessaire et nous épargnerait bien des soucis).

On peut aussi schématiser le fonctionnement de notre application sous forme de blocs, chacun des blocs représentant une fonction ou une sous-procédure :

A ce stade, l'analyse dite fonctionnelle est terminée. Les fondations (solides, espérons-le) sont posées pour finaliser l'application.

Etape 3 : Algorithmes détaillés

Normalement, il ne nous reste plus qu'à traiter chaque procédure isolément. On commencera par les sous-procédures et fonctions, pour terminer par la rédaction de la procédure principale.