Rappel : ce cours d'algorithmique et de programmation
est enseigné à l'Université Paris 7,
dans la spécialité PISE du Master MECI (ancien DESS AIGES)
par Christophe Darmangeat

Partie 11
Procédures 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 l'intitulé de la question, et 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.

Le jargon consacré parle d'ailleurs à ce sujet de **factorisation du code** : c'est une manière de parler reprise des matheux, qui « factorisent » un clacul, c'est-à-dire qui en regroupent les éléments communs pour éviter qu'ils ne se répètent. Cette factorisation doit, tant qu'à faire, être réalisée avant de rédiger le programme : il est toujours mieux de concevoir les choses directement dans leur meilleur état final possible. Mais même quand on s'est fait avoir, et qu'on a laissé passer des éléments de code répétitifs, il faut toujours les factoriser, c'est-à-dire les regrouper en fonctions, et ne pas laisser des redondances.

La phase de conception d'une application qui suit l'analyse et qui précède l'algorithmique proprement dite, et qui se préoccupe donc du découpage en modules du code s'appelle l'**analyse fonctionnelle** d'un problème. C'est une phase qu'il ne faut surtout pas omettre ! Donc, je répète, pour concevoir une application :

1. On identifie le problème à traiter, en inventoriant les fonctionnalités nécessaires, les tenants et les aboutissants, les règles explicites ou implicites, les cas tordus, etc. C'est l'**analyse**.
2. On procède à un découpage de l'application entre une procédure qui jouera le rôle de chef d'orchestre, ou de donneur d'ordre, et des modules périphériques (fonctions ou sous-procédures) qui joueront le rôle de sous-traitants spécialisés. C'est l'analyse fonctionnelle.
3. On détaille l'enchaînement logique des traitements de chaque (sous-)procédure ou fonction : c'est l'algortthimique.
4. On procède sur machine à l'écriture (et au test) de chaque module dans le langage voulu : c'est le codage, ou la programmation proprement dite.

Retour Haut de Page

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 ou 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 de manière groupée une série de valeurs. Problème, une fonction ne peut renvoyer qu'une seule valeur à la fois.

Alors, dans ces deux cas, faute de pouvoir traiter l'affaire par une fonction, on devra faire appel à un outil plus généraliste : la **sous-procédure**.

En fait, **les fonctions ne sont qu'un cas particulier des sous-procédures, dans lequel le module doit renvoyer vers la procédure appelante une valeur et une seule**. Donc, chaque fois que le module souhaité ne doit renvoyer aucune valeur, ou qu'il doit en renvoyer plusieurs, il faut donc avoir recours non à la forme particulière et simplifiée (la fonction), mais à la forme générale (la sous-procédure). Le rapport entre fonctions et sous-procédures est donc semblable à celui qui existe entre les boucles Pour et les boucles TantQue : les premières sont un cas particulier des secondes, pour lequels les langages proposent une écriture plus directe.

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.
- Dans le code appelant, la valeur (retournée) d'une fonction était toujours affectée à une variable (ou intégrée dans le calcul d'une expression). L'appel à une procédure, lui, est 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. C'est aussi pourquoi la déclaration de la fonction spécifiait le type de la valeur retournée, tandis que la déclaration de la sous-procédure ne spécifie rien de tel.

2.2 Le problème des arguments

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, c'est là que vont se trouver les outils qui vont permettre l'échange d'informations entre la procédure appelante et la sous-procédure.

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-procédures 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 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 les 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 règle 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**.

En réalité, il n'existe pas à proprement parler (sauf dans certains langages) de paramètres fonctionnant uniquement en sortie. La situation générale, c'est qu'un paramètre fonctionne soit en entrée seule (comme on l'a considéré jusqu'à présent) soit à la fois en entrée et en sortie. Autrement dit, un paramètre peut **toujours** être transmis depuis la procédure appelante vers la procédure appelée. Mais seuls les paramètres définis en sortie (dans la procédure appelée, oeuf corse) pourront transmettre en sens inverse une valeur vers la procédure appelante.

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 verre d'eau pour faire passer l'aspirine.

Il nous reste à savoir comment on fait 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 dans la déclaration d'une sous-procédure 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 en existe deux :

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

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

Prenons l'exemple d'un module à qui on fournirait une chaîne de caractères, et qui devrait en extraire (séparément) le premier et le dernier caractère. Il s'agirait bien d'une sous-procédure, puisque

celle-ci devrait fournir deux résultats distincts. Celle-ci devrait comporter trois arguments : la chaîne fournie, le premier et le dernier caractère. Laissons pour l"instant de côté la question de savoir comment ceux deux dernières informations pourront être véhiculées en sortie vers le programme principal. En ce qui concerne la chaîne en entrée, nous allons déclarer qu'elle est un paramètre transmis **par valeur**. Cela donnera la chose suivante :

```
Procédure FirstLast(Msg en Caractère par Valeur, Prems ..., Dern ...)
   ...
   ??? Left(Msg, 1)
   ??? Right(Msg, 1)
   ...
Fin Procédure
```

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

```
Variables Amouliner, Alpha, Omega en Caractère
Amouliner ← "Bonjour"
Appeler FirstLast(Amouliner, Alpha, Omega)
```

...où Alpha et Omega sont deux variables de type caractère.

Que va-t-il se passer en preil cas ? Lorsque le programme principal appelle la sous-procédure, celle-ci crée aussitôt trois nouvelles variables : Msg, Prems et Dern. Msg ayant été déclarée comme un paramètre passé **par valeur**, elle va être immédiatement affectée avec le contenu (actuel) de Amouliner. Autrement dit **Msg est dorénavant une copie de Amouliner**, copie qui subsistera tout au long de l'exécution de la sous-procédure FirstLast et sera détruite à la fin de celle-ci.

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

C'est certes une limite, mais c'est d'abord et avant tout 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.

Nous allons maintenant nous occuper des deux autres paramètres, Prems et Dern, qui, eux, doivent pouvoir fonctionner en sortie. Nous préciserons donc que ces paramètres seront transmis **par référence**. L'écriture complète de la sous-procédure est ainsi la suivante :

```
Procédure FirstLast(Msg en Caractère par Valeur, Prems en Caractère par Référence,
Dern en Caractère par Référence)
   Prems ← Left(Msg, 1)
   Dern ← Right(Msg, 1)
Fin Procédure
```

Quant à l'appel à la sous-procédure, lui, il ne change pas.

Dépiautons le mécanisme de cette nouvelle écriture. En ce qui concerne l'affectation de la variable Msg, rien de nouveau sous le soleil. Mais en ce qui concerne Prems et Dern, le fait qu'il s'agisse d'un passage par référence fait que **ces variables sont affectées non par le *contenu* des variables Alpha et Omega du programme appelant, mais par leur *adresse* (leur référence)**.

Cela signifie dès lors que **toute modification de Prems et Dern sera immédiatement redirigée, par ricochet en quelque sorte, sur Alpha et Omega**. Prems et Dern ne sont pas des variables ordinaires : elles ne contient pas de valeur, mais seulement la référence à une valeur, qui elle, se trouve ailleurs (respectivement, dans les variables Alpha et Omega). On peut dire que Prems et Dern ne sont pas une copie de Alpha et Omega, mais un alias, un autre nom, de ces variables. On se trouve donc en présence d'un genre de variable complètement nouveau par rapport à tout ce que nous connaissions. 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. Truc devient donc, en quelque sorte, un faux nez : c'est un pseudonyme pour la variable T. Et tout ce qui arrive à Truc arrive donc en réalité à T.

Passer un paramètre par référence, cela permet donc **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. Mais on comprend à quel point on ouvre ainsi la porte à des catastrophes : si la sous-procédure fait une bêtise avec sa variable, elle fait en réalité une bêtise avec la variable du programme principal...

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.

Pour résumer tout cela, filons la métaphore ; si la procédure principale est une entreprise, et que les fonctions et procédures sont des sous-traitants :

- l'entreprise peut transmettre à ces sous-traitants tous les documents nécessaires pour qu'ils effectuent leurs travaux.
- les sous-traitants « fonctions » rendent les résultats de leur travail en produisant leurs propres documents, qu'ils livrent dans une enveloppe qui porte leur tampon. Toutefois, chaque sous-traitant ne sait produire qu'un seul document. Inversement, les sous-traitants « procédures » ne produisent aucun document à leur nom. Tout au plus peuvent-ils modifier des documents à en-tête de l'entreprise, que celle-ci leur a confiés. En revanche, ces sous-traitants ont une capacité de travail illimitée, et peuvent modifier autant de documents que le souhaite l'entreprise donneuse d'ordres.
- lorsque l'entreprise s'adresse à un sous-traitant « fonction », elle lui remet exclusivement des copies de ses propres documents pour que le sous-traitant puisse accomplir sa tâche. En revanche, lorsque l'entreprise s'adresse à un sous-traitant « procédure », elle a le choix entre lui confier des copies ou ses documents originaux. Confier les originaux est la seule façon pour l'entreprise de bénéficier du travail de son sous-traitant « procédure », mais pour des raisons évidentes, elle le fera sous le patronage du plus célèbre couple italo-arménien, c'est-à-dire avec parcimonie et à bon escient.

Un dernier point concerne le traitement des tableaux. Lorsqu'un code doit traiter un tableau, faut-il considérer qu'il traite une valeur unique (auquel cas, on pourra le faire renvoyer par une fonction) ou une série de valeurs (auquel cas, il faudra passer le tableau par référence à une sous-procédure) ? La réponse à cette question dépend du langage ; certains admettent la première possibilité, d'autres obligent à la seconde. Dans ce cours, et par convention, nous choisirons l'option la plus contraignante, à savoir qu'un tableau représente forcément une série de valeurs et qu'il ne peut donc être renvoyé par une fonction.

Retour Haut de Page

3. Variables locales et globales

Encore un point. Nous venons de voir que nous pouvions (et devions) 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, sous les mêmes noms. 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 **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 **globale**.

Par défaut, toute variable est locale. La manière dont la déclaration d'une variable globale doit être faites est évidemment fonction de chaque langage de programmation. En pseudo-code algorithmique, on pourra utiliser le mot-clé **Globale** :

```
Variable Globale Toto en Numérique
```

Alors, pourquoi ne pas rendre toutes les variables globales, et s'épargner ainsi de fastidieux efforts pour passer des paramètres ? C'est très simple, et c'est toujours la même chose : d'une part, les variables globales consomment énormément de ressources en mémoire. En conséquence, le principe qui doit présider au choix entre variables globales et locales doit être celui de l'économie de moyens. Mais d'autre part, et surtout, multiplier les variables globales est une manière peu sécurisée de programmer. C'est exactement comme quand dans un navire, on fait sauter les compartiments internes : s'il y a une voie d'eau quelque part, c'est toute la coque qui se remplit, et le bateau coule. En programmant sous forme de modules ne s'échangeant des informations que via des arguments, on adopte une architecture compartimentée, et on réduit ainsi considérablement les risques qu'un problème quelque part contamine l'ensemble de la constrution.

Moralité, on ne déclare comme globales 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 globales.


Retour Haut de Page

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**. Quitte à radoter, je le répète une fois de plus, il convient :

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.