(5) Programmation structurée : sous-programmes

Il est facile de construire directement un programme simple. Cependant, dès que le problème à programmer est compliqué ou que le code rédigé dépasse une ou deux pages, on ne peut conserver une vision global de ce qu'on fait sans décomposer le travail en sous-tâches. La plupart des langages ont cette possibilité.

1.) Quelques exemples

Exemple 1 : Utilisation d'une fonction
Calcul de C(n,p)=n! / (p!*(n-p)!) avec 0<=p<=n.

```
PROGRAM Cnp ;
VAR n, p, C : integer ;

FUNCTION facto(k:integer):integer ;    signification : facto : integer --> integer, k |--> k!
Var i, tmp : integer;
Begin
tmp:=1 ;
for i:=1 to k do tmp:=tmp*i ;
facto:=tmp ;                           on affecte à facto (nom de la fonction) sa valeur de sortie
End ;                                  dans les premieres version, on ne peut affecter qu'une fois.

BEGIN
(* saisir n et p *)
repeat
   write('n (0<=n) ? ') ; readln(n) ;
   write('p (0<=p<=n) ? ') ; readln(n) ;
   until (n>=0) and (n>=p) (and (p>=0) ;
(* calcul de C(n,p) *)
C:= facto(n)/(facto(p)*facto(n-p)) ;          la fonction retourne une valeur qui peut être
(* affichage du résultat *)                   utilisée comme telle
writeln('C(',n,',',p,')=',C) ;
END.
```

Exemple 2 : Utilisation de procédures
Tri dans un tableau.

```
PROGRAM TrierTableau ;
VAR T : array[1..100] of integer ;              variable globale

PROCEDURE saisir ;
Var i : integer ;                               variable locale
Begin
for i:=1 to 100 do
    begin
    write(i,'-ième valeur ? ') ;
    readln(T[i]) ;
    end ;
End ;

PROCEDURE trier ;
Var i, j : integer ;
Begin
...
End ;

PROCEDURE afficher(debut, fin : integer) ;      procédure avec paramètres
Var i : integer ;
Begin
for i:=debut to fin do writeln('T[',i,']=',T[i]);
End ;
```

```
BEGIN
SAISIR ;
TRIER ;
AFFICHER(1,100) ;
END.
```

Exemple 3 : Fonction ET procédures
Une autre version de Cnp.

```
PROGRAM Cnp ;
Var C, n, p : integer ;

FUNCTION facto
... (idem) ...

PROCEDURE saisir ;
Begin
repeat
   write('n (0<=n) ? ') ; readln(n) ;
   write('p (0<=p<=n) ? ') ; readln(n) ;
   until (n>=0) and (n>=p) (and (p>=0) ;
End ;

PROCEDURE calculer ;
Begin
C:= facto(n)/(facto(p)*facto(n-p)) ;
End ;

PROCEDURE afficher ;
Begin
writeln('C(',n,',',p,')=',C) ;
End ;

BEGIN
saisir ;
calculer ;
afficher ;
END.
```

N.B. : il faut donc déclarer la fonction avant!

On voit que le programme construit peut, en quelque sorte, déléguer une partie de son travail à un sous-programme (procédure ou fonction). Chaque ssp constitue une unité de programmation qui peut posséder ses propres variables, constantes et types et faire appel à d'autres sous-programmes.

On voit dans l'exemple précédent que le corps du programme devient très lisible car il ne contient plus que les grandes lignes. Il faut cependant savoir éviter de créer une pléthore de ssp ("Le tact dans l'audace c'est de savoir jusqu'où l'on peut aller trop loin"). L'art de la programmation consiste notamment à choisir un bon découpage du programme (et donc une bonne analyse du problème à informatiser) et une bonne répartition entre procédures et fonctions.

2.) Quelques définitions

Définition 1 :
Un **sous-programme** (*routine* en anglais) est un ensemble d'actions auquel on donne un nom, qui est décrit formellement dans la partie déclarative du programme principal et qui peut être activé dans la partie exécutive du programme.
Ces actions agissent en général sur des données pour produire des résultats. Données et résultats sont les paramètres du sous-programme.

Définition 2 :

Une **fonction** réalise un certain nombre d'actions et produit comme résultat (on dit retourne) une valeur (de type simple ou structurée), qui peut être intégrée dans une expression plus élaborée, affectée à une variable, etc. Les contraintes de types sont les mêmes que pour les expressions déjà vues.
Une **procédure** réalise un certain nombre d'actions mais ne retourne rien.
N.B. : cette manière de voir est différente de celles utilisées par d'autre langages (absence de procédure en Lisp, procédures retournant par défaut un entier en C, etc.).

3.) La déclarations des sous-programmes

Format de la déclaration :

PROCEDURE nom_procedure[([variable[, variable]* **:** type][**;** variable[, variable]* : type]*)]? **;**
```
procedure sans ;
procedure un(x:real);
procedure unun(x,y:real);
procedure deux(x:real; k:integer);
procedure trois(x:real;c:char;k:integer);
```

FUNCTION nom_fonction[([variable[, variable]* **:** type][**;** variable[, variable]* : type]*)]? **:** type **;**
```
function sans:integer;
function un(x:real):char;
...
```

On peut avoir ou non une liste de paramètres d'entrée, le sous-programme peut ou non retourner une valeur.
Limitations du type d'entrée (type simple seulement) dans les ssp en Turbo Pascal jusqu'à la version 5. A partir de Turbo Pascal version 6, il n'y a plus de limitation sur le type de paramètres d'entrée. Limitations sur le type de paramètre en sortie en Turbo Pascal : type simple + string (pas string[k] ni array) ; cette limitations n'existe normalement pas sous GPC.

On a donc quatre format de déclaration de sous-programmes :

entrée \ sortie	sans	avec paramètre
sans	PROCEDURE exp : clrscr	FUNCTION : paramètres exp :
avec paramètre(s)	PROCEDURE(Liste paramètres) exp : write(...)	FUNCTION(Liste param.) : paramètres exp : sin(x)

N.B. : Pascal contient un grand nombre de ssp et de fonctions prédéfinies (leur demander ce qu'ils connaissent) : sin, sqrt, sqr, ..., clrscr, write, gotoXY(colonne>=1, ligne>=1),...

Sont donc prédéfinies :
-) function sin (x:real) : real
type de ssp, nom, liste de paramètres de données avec leurs noms, type du résultat.
-) procedure write (...)
type de ssp, nom liste de paramètres de données séparés par des virgules.

N.B. : certains de ces ssp ne sont accessibles que grâce à une prédéclaration : USES nomunité ;.

Attention : dans la déclaration les données sont séparées par des points-virgules (paramètres formels) mais par des virgules dans l'activation (paramètres "réels"). Voir plus loin.

N.B. : Pour contourner les limitations de types de paramètres, transformation d'une fonction en procédure en remplaçant le résultat par une donnée passée "par variable" (=par adresse) et non par valeur : voir ch (6).

Exercices : déclarer les fonctions suivantes :
`puissance : x-->x`^{n}, n entier naturel (puis relatif)
`caesar : c,n -->`n-ième successeur de c
`cesar : s,n-->t,` codage de César de la chaîne (utiliser caesar).
...

4.) Activation : l'utilisation des sous-programmes

Lors de chaque appel à un ssp, les paramètres effectifs doivent correspondre en nombre , en ordre et en type aux paramètres formels de l'en-tête du sous-programme.

L'activation effective du sous-programme se fait dans la partie exécutive du programme par une instruction formée du nom du programme suivi de la liste des paramètres effectifs (des valeurs) remplaçant les paramètres formels dans les calculs.
Cette activation peut avoir lieu autant de fois qu'on veut avec différents paramètres effectifs, chaque activation sera réalisée indépendamment des autres.

4.1.) fonctions

Une fonction est activée par l'évaluation d'une expression qui la contient (affectation, test, écriture) et qui précise ses paramètres d'entrée.
Exp : x:=sin(y) ; C:= facto(n)/(facto(p)*facto(n-p)) ;

Pour activer une fonction, il faut donc connaître son nom, la liste de ses données d'entrée, avec leurs types, et le type de son résultat, toutes indications regroupées dans l'en-tête de la fonction.

4.2.) procédures

Une procédure est activée par une instruction placée dans le corps (partie exécutive) du programme principal ou d'un sous-programme qui précise ses paramètres d'entrée.

Pour activer une procédure, il faut donc connaître son nom et la liste de ses données d'entrée, avec leurs types, toutes indications regroupées dans l'en-tête de la fonction.

5.) variables locales vs globales

La déclaration d'un sous-programme comporte trois parties :

1) "En-tête" comprenant le type du ssp (function ou procedure), son nom, ses éventuels paramètres d'entrée et de sortie.

2) Partie "déclarations" propre au ssp : liste des objets locaux (types, constantes et variables) qui sont des objets éphémères qui apparaissent au début d'une activation du ssp et disparaissent à la fin (l'ordinateur leur réserve une zone mémoire qui est ensuite libérée).

3) Partie "instructions" : bloc d'instructions (délimité par Begin et End;) qui peuvent utiliser les objets propres au ssp (variables locales) et les objets définis dans l'en-tête du programme (variables globales).

Règle 1 : si une variable locale a même nom qu'une variable globale, la première a priorité. Le sous-programme n'a donc plus accès à la variable globale.

Une bonne habitude : toujours déclarer les indices de boucles en local.

Règle 2 : le nom de la fonction joue le rôle d'une variable locale et doit être affecté, de manière unique dans le corps de la fonction, d'une valeur compatible avec le type résultat de la fonction.

Exemple : Travail sur les points du plan (exercice en fournissant les en-têtes?)

```
PROGRAM plan ;
TYPE coordonnee = real ;
          point = array[1..2] of coordonnee ;
Var M, N, P : point ;
       x, y : coordonnee ;
```

```
FUNCTION saisirpoint : point ;
Var M : point ;
Begin
writeln('Saisie d'un nouveau point :') ;
write('abscisse du point ? ') ; readln(M[1]) ;
write('ordonnée de M ? ') ; readln(M[2]) ;
saisirpoint:=M ;
End ;

FUNCTION distance(X,Y : point) ;
Begin
distance:=sqrt((X[1]-Y[1])^2+(X[2]-Y[2])^2) ;
End ;

FUNCTION est_rectangle(M, N, P : point) : boolean ;
Var test : boolean ;
Begin
test:=(distance(M,N)^2+distance(N,P)^2=distance(M,P)^2) ;
test:=test or (distance(M,N)^2=distance(N,P)^2+distance(M,P)^2) ;
test:=test or (distance(N,P)^2=distance(M,N)^2+distance(M,P)^2) ;
est_rectangle:=test ;
End ;

FUNCTION typedetriangle(M,N,P : point) : string ;
Var S : string ;
Begin
if est_rectangle(M,N,P) then S:=' rectangle '
else if est_equilateral(M,N,P) then S:=' équilatéral '
else if est_isocele(M,N,P) then S:=' isocèle '
else if est_plat(M,N,P) then S:=' plat '
else S:=' quelconque ' ;
typedetriangle:=S ;
End ;

BEGIN
M:=saisirpoint ;
N:=saisirpoint ;
P:=saisirpoint ;
writeln('le triangle MNP est : ',typedetriangle(M,N,P)) ;
END.
```

Exercices

1) Modifier le programme sur les triangles pour n'importe quelle dimension (>2).

2) (fondamental)
a) Reprendre le programme d'affichage des chiffres du TP4 : en choisissant convenablement les sous-programmes, alléger le code principal et supprimer les redondances.
b) Reprendre dans le même esprit les autres programmes des TP 1 à 6 (à faire à la maison).

3) Contruire un programme qui cherche tous les nombres premiers de 2 à n (n saisi par l'utilisateur).

Analyse descendante du problème (du général au particulier) :
Il faut vérifier pour chaque nombre de 2 à n s'il est premier.
Un nombre est premier ssi il n'est divisible que par 1 et lui-même.
Un nombre p est divisible par q ssi (p mod q)=0.

Construction ascendante de l'algorithme / du programme :
fonction estpremier(p : entier) : entier
    test:=vrai ;
    pour i de 2 a trunc(sqrt(p)) faire si (p mod i)=0 alors test<-faux ;
    retourner test ;

Algorithme Chercherpremier :

Saisir n ;
Pour i de 2 à n faire si estpremier(i) alors afficher i ;

Remarque importante : l'habitude des petits problèmes nous a habitué à pratiquer une méthode ascendante (souvent même sans réelle analyse). On voit que d'autres manières de procéder existent (quatre fondamentales : analyse vs construction & ascendante vs descendante).

4) Construire un programme qui cherche tous les nombres parfaits de 1 à n.
Un nombre est parfait s'il est la somme de ses diviseurs "propres" (1 compris et lui-même exclus).

5) Construire un programme qui cherche tous les pairs d'amis parmi les nombres de 1 à n.
Deux nombres sont amis si chacun est la somme des diviseurs "propres" de l'autre.