INSTITUT FRANÇAIS DE RECHERCHE POUR
LE DEVELOPPEMENT EN COOPERATION

ORSTOM

MAA - Milieux et activités agricoles

UR 3J Organisations régionales

Cours de programmation avancée en Pascal

Gérard Cochonneau

INTRODUCTION

Ce manuel de cours est la version française d'un manuel en portugais, du même auteur, utilisé lors de la réalisation d'un cours au Departamento de Informática de l'EMBRAPA (Empresa Brasileira de Pesquisa Agropecuária), du 16 au 27 juillet 1990. Ce cours, basé sur la version 5.5 do Turbo-Pascal, a été qualifié de "Pascal Avancé" car il s'agit d'un complément à un cours de Pascal Basique donné précédemment au même endroit.

L'élaboration de ce cours a eu pour but d'améliorer les connaissances pratiques des informaticiens de l'EMBRAPA impliqués dans le développement et la maintenance de programmes écrits en Pascal. La majorité des exemples et des exercices sont issus de situations réelles rencontrées dans le développement des modules de SISGEO. Le dernier chapitre traite de la Programmation Orientée Objet ; il s'agit seulement d'une initiation qui n'approfondit pas la totalité des techniques de la POO. Enfin, deux annexes rassemblent les en-têtes des routines disponibles dans les *units* standards du Turbo-Pascal ainsi que la signification des différents messages d'erreurs à l'éxécution.

Une disquette accompagne ce manuel. Elle contient :

- tous les fichiers nécessaires à la compilation et au test de tous les exemples présentés ;
- la correction des exercices proposés.

Chapitre I

Unités et overlays

A partir de la version 4.0 Turbo-Pascal dispose de la notion *d'unité*, indispensable pour utiliser de façon optimum les compilateurs des versions 4.0 et ultérieures et pour améliorer la modularité des programmes. Dans ce chapitre, nous définirons ce qu'est une *unité*, quels sont ses avantages ; nous verrons également comment utiliser les *unités* déjà disponibles et comment écrire ses propres *unités*.

1 Définition et description d'une unité.

1.1 Introduction.

En général, une fonction quelconque d'un système en cours de développement peut être logiquement divisée en plusieurs sous-fonctions relativement indépendantes qui ont seulement besoin de partager un nombre minimum d'informations. Prenons pour exemple une fonction d'émission des données d'un fichier : elle peut être divisée en trois sous-fonctions :

- obtenir la demande de l'utilisateur (critères de sélection des données, choix du périphérique de sortie, généralement obtenus à l'aide d'un écran de saisie),
- émission des données sur l'écran, si l'utilisateur a choisi ce périphérique,
- émission des données sur l'imprimante.

Les données qui doivent communiquer entre les différentes fonctions étant :

- le type de périphérique qui permettra de choisir entre les deux types d'émission,
- les critères de sélection.

En adaptant ce raisonnement au niveau physique, nous obtenons un programme principal :

- qui fait d'abord appel à un module pour saisir les critères de sélection et le type de périphérique,
- qui ensuite, en fonction du type de périphérique, exécute un module d'émission sur l'écran ou un module d'émission sur l'imprimante.

Une forme adéquate d'implémenter cela en Turbo-Pascal est justement l'utilisation de trois *unités* correspondantes utilisées par un programme principal.

Un autre cas souvent rencontré concerne l'utilisation, dans plusieurs fonctions du système, des mêmes sous-fonctions. Par exemple, le fichier dont les données sont à émettre dans l'exemple antérieur, devra également être accédé (en lecture et en écriture) par le programme de mise à jour du fichier.

L'utilisation d'une *unité* rassemblant les routines en relation avec ce fichier (ouverture, lecture, écriture, fermeture) constitue la meilleure manière d'implémenter cela en Turbo-Pascal ; cette *unité* étant ensuite utilisée par plusieurs programmes.

1.2 Structure d'une unité.

Une *unité* est un ensemble d'objets du langage Pascal (constantes, variables, types, procédures, fonctions), de la même manière qu'un programme normal. La structure d'une

unité est d'ailleurs très semblable à celle d'un programme, les principales différences se trouvent dans la partie d'en-tête et la partie de déclarations.

Une *unité* est divisée en trois parties :

- une partie *interface* qui contient les déclarations de type, de constantes, de variables, de fonctions et de procédures qui seront accessibles de l'extérieur de l'unité (déclarations *publiques*),
- une partie d'implémentation de ces fonctions et procédures, qui peut également contenir d'autres types, constantes, variables, fonctions et procédures qui resteront invisibles de l'extérieur de l'*unité* (déclarations *privées*),
- une partie d'initialisation, facultative, qui sera automatiquement exécutée au début des programmes qui utiliseront l'*unité*.

```
unit <identification>
interface
uses <listes d'unités> {facultatif}
  {déclarations publiques}

implementation
  {déclarations privées}
  {implémentation des procédures et fonctions}

begin
  {partie d'initialisation}
end.
```

L'en-tête de l'*unité* commence obligatoirement par le mot réservé *unit* suivi du nom de l'unité, sous une forme semblable au nom d'un programme.

1.3 Description de la partie d'interface.

Le mot réservé **interface** correspond au début de la partie d'interface de l'unité. Une unité peut utiliser d'autres unités (voir note d de l'exemple 1.6), mais ce n'est pas obligatoire. Si c'est le cas, elles doivent être indiquées dans la directive **uses** qui doit suivre immédiatement le mot réservé **interface**. Chaque unité de la liste doit être séparée de la précédente par une virgule, comme sur l'exemple suivant :

```
interface
uses crt, dos;
```

Déclarer ensuite tous les éléments qui doivent être visibles de l'extérieur de l'unité. La syntaxe est identique à celle utilisée dans un programme normal, en ce qui concerne les types, les constantes et les variables. Toute procédure ou fonction susceptible d'être appelée de l'extérieur de l'unité doit également être déclarée : il suffit pour cela de répéter son en-tête dans la partie d'interface en respectant les mêmes règles syntaxiques que pour l'en-tête utilisé pour définir une fonction ou une procédure dans un programme normal. La partie d'implémentation de la routine (variables locales, corps de la routine) doit seulement être introduite dans la partie d'implémentation de l'unité (voir note b de l'exemple 1.6).

1.4 Description de la partie d'implémentation.

Le mot réservé **implementation** marque la fin de la partie d'interface et le début de la partie d'implémentation, partie privée de l'unité. Tous les éléments déclarés dans la partie

d'interface sont visibles de l'intérieur de la partie d'implémentation. Il en est de même des éléments publics des unités déclarées dans la clause *uses* de la partie d'interface.

On peut également déclarer d'autres éléments (types, constantes, etc...) dans la partie d'implémentation ; ils seront visibles de l'intérieur de cette partie d'implémentation, mais invisibles pour les programmes ou les unités qui utilisent l'unité considérée.

Toutes les procédures et fonctions déclarées dans la partie d'interface doivent être définies dans la partie d'implémentation. Il est conseillé de définir l'en-tête de chaque routine de façon identique à la déclaration présente dans la partie d'interface (voir note c de l'exemple 1.6), mais ce n'est pas obligatoire : il suffit en fait d'indiquer le mot réservé *procedure* ou *function* suivi du nom de la routine, sans mentionner les éventuels paramètres ni le type de la fonction (voir note e de l'exemple 1.6).

D'autres routines, nécessaires à l'implémentation, peuvent être définies dans la partie d'implémentation. Elles peuvent être utilisées dans cette partie mais restent invisibles de l'extérieur de l'unité.

1.5 Description de la partie d'initialisation.

Il s'agit de la dernière partie de l'unité. Elle peut se réduire à la directive *end.* qui indique la fin de l'unité (voir note a de l'exemple 1.6). Dans le cas contraire, elle commence avec la directive *begin* et ressemble au corps d'un programme principal.

La partie d'initialisation contient des ordres exécutables quelconques et peut faire référence à tous les objets visibles de l'intérieur de l'unité : objets de la partie d'interface comme de la partie d'implémentation. Elle sera toujours exécutée, une seule fois, au début de l'exécution des programmes qui font appel à l'unité.

1.6 Un exemple commenté.

L'unité **fenetres**, présentée dans ce chapitre, est utilisée par l'unité **VisuTas** qui sera étudiée dans le chapitre II. Elle contient quelques routines utiles à la définitions de fenêtres sur l'écran. Les chapitres précédent ont fait référence aux notes présentées en commentaires dans le listing.

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{                              exemple d'unité                               }
{                     (disponible dans le fichier fenetres.pas)              }
{----------------------------------------------------------------------------}
unit fenetres;

interface  {note b : partie d'interface avec définition du type TypeFen,}
           {         d'une fonction et de trois procédures               }

uses crt;  {note d : l'unité fenetres utilise l'unité crt}

type
  TypeFen = record
              Ligne,
              Colonne,
              Largeur,
              Hauteur : byte;
              Titre   : string;
            end;
```

```
procedure AfficherTexte(x, y : byte; t : string);
function FenetreDefinie(l, c, larg, haut : byte; tit : string; var fen : TypeFen) : boolean;
procedure AfficherFenetre(fen : TypeFen);

implementation

uses afferr;  {note f : déclaration de l'unité afferr, dans la partie d'implémentation,}
              {         pour éviter l'erreur de référence circulaire                   }

procedure AfficherTexte(x, y : byte; t : string); {note c : en-tête identique à    }
                                                  {         celui de la déclaration}
begin
  if (x * y) <> 0        (*si x et y sont <> 0                    *)
    then gotoxy(x, y);   (*  déplacer le curseur                  *)
  write(t);              (*sinon, écrire à la position du curseur  *)
end;

function FenetreDefinie;  {note e : en-tête simplifié}
begin
  FenetreDefinie := true;
  if (c in [1..80]) and (c + larg in [2..81]) and
     (l in [1..25]) and (l + haut in [2..26])
    then with fen do
      begin
        Ligne := l;
        Colonne := c;
        Largeur := larg;
        Hauteur := haut;
        Titre := copy(tit, 1, larg-2);
      end
    else begin
      AfficherErreur('définition de fenêtre incorrecte');
      FenetreDefinie := false;
    end;
end;

function Chaine(n : byte; c : char) : string; {note g : procédure privée   }
                                              {         de l'unité fenetres}
var
  s : string;
begin
  fillchar(s, succ(n), c);
  s[0] := char(n);
  Chaine := s;
end;

procedure AfficherFenetre; {note e : en-tête simplifié}
var
  k,
  i : byte;

begin
  with fen do
    begin
      window(Colonne, Ligne, Colonne + Largeur, Ligne + Hauteur);
      k := (Largeur - 2 - length(Titre)) div 2;
      AfficherTexte(1, 1, '┌');
      AfficherTexte(0, 0, Chaine(k, '─'));
      AfficherTexte(0, 0, Titre);
```

```
      AfficherTexte(0, 0, Chaine(Largeur - 2 - k - length(Titre), '═'));
      AfficherTexte(0, 0, '╗');
      for i := 2 to pred(Hauteur) do
        begin
          AfficherTexte(1, i, '║');
          AfficherTexte(0, 0, Chaine(Largeur-2, ' '));
          AfficherTexte(Largeur, i, '║');
        end;
      AfficherTexte(1, Hauteur, '╚');
      AfficherTexte(0, 0, Chaine(Largeur-2, '═'));
      AfficherTexte(0, 0, '╝');
    end;
  end;

end.          {note e : partie d'initialisation inexistante}→
```

2 Principaux avantages de l'utilisation d'unités.

Les unités sont la base de la programmation modulaire en Turbo-Pascal ; elles permettent la division d'un programme en plusieurs modules logiques qui peuvent être compilés, et parfois testés, séparément.

Chaque unité est aussi une **bibliothèque** d'éléments publics (constantes, variables, types, routines) qui pourront être réutilisés dans d'autres unités ou dans des programmes qui utilisent l'unité à laquelle ils appartiennent. Ceci sans qu'il soit nécessaire de recompiler l'unité ni de disposer de son code source. Il suffit de pouvoir consulter une documentation qui décrit tous les éléments de la partie d'interface. C'est ainsi qu'une unité bien conçue et bien implémentée peut profiter à plusieurs programmes, voire plusieurs systèmes, sans avoir besoin d'être compilée à nouveau.

En temps de compilation d'un programme, les unités déjà compilées (c'est à dire déjà disponibles sous forme de fichiers d'extension *.TPU*), et qui n'ont pas été récemment modifiées, n'ont pas besoin d'être recompilées à chaque fois ; ce qui peut représenter une économie de temps significative. Toutefois, le temps nécessaire à la lecture des parties d'interface des fichiers d'extension *.TPU* doit être décompté de l'économie ainsi réalisée.

Le fait de pouvoir déclarer des variables dans la partie d'implémentation est également intéressant. Les variables ainsi déclarées sont locales à l'unité (et donc protégées contre une corruption venant de l'extérieur), mais globales pour toutes les procédures et fonctions implémentées dans l'unité (et donc utilisables comme les variables globales d'un programme).

3 Utilisation des unités.

3.1 Par un programme.

Toutes les unités utilisées par un programme doivent être citées dans une directive *uses* située dans la partie de déclarations d'un programme, préalablement à une référence quelconque à un des éléments de ces unités. Par exemple :

```
program Rien;
uses crt, graph, MonUnit;
```

La directive uses doit être unique à l'intérieur d'un programme et doit suivre l'en-tête facultatif du programme.

Si une des unités citées utilise une autre unité, il n'est pas nécessaire que celle-ci soit citée dans la directive *uses*. Par exemple, se MonUnit utilise l'unité **DOS**, celle-ci n'a pas besoin d'être déclarée dans la clause *uses* ci-dessus.

Par contre, si le programme **Rien** fait lui aussi appel à l'unité **DOS**, elle devra être citée avant MonUnit. Ce qui veut dire que l'ordre de l'énumération des unités n'est pas indifférent ; la règle est la suivante : une unité Y qui utilise une unité X déclarée dans la même clause *uses* doit être citée après X.

3.2 Par une autre unité.

Une unité peut utiliser d'autres unités, de la même manière qu'un programme principal. La directive *uses* doit être placée immédiatement après la directive *interface* et ne peut être utilisée qu'une seule fois dans la partie d'interface d'une unité.

La déclaration, dans la partie d'interface d'une unité a, de l'utilisation d'une unité b qui utilise l'unité a n'est pas permise. L'exemple suivant provoquerait une erreur **68 Circular unit reference** lors de la compilation :

```
unit a;                          unit b;
interface                        interface
  uses b;                          uses a;
```

Cette limitation peut être contournée par l'utilisation d'une directive *uses* dans la partie d'implémentation. L'unité présentée à suivre est utilisée par l'unité **fenetres** pour afficher un message d'erreur. Or, pour afficher le message, la routine AfficherErreur a besoin de la routine AfficherTexte, elle-même définie dans l'unité **fenetres**. Il s'agit donc d'un cas de *référence circulaire d'unité* qui a été résolu par l'utilisation de la directive *uses*, tant dans la partie d'implémentation de **fenetres** (voir note f de l'exemple 1.6), que dans celle AfficherErreur (voir plus bas).

Ceci n'est acceptable que si la partie d'interface de l'unité n'a pas besoin d'un élément de celle déclarée dans la partie d'implémentation, toujours située après la partie d'interface.

```
{---------------------------cours de Pascal-Avancé--------------------------}
{                                                                            }
{                    exemple de référence circulaire d'unités                }
{                     (disponible dans le fichier afferr.pas)                }
{----------------------------------------------------------------------------}
unit afferr;
interface

procedure AfficherErreur(t : string);

implementation
uses fenetres;

procedure AfficherErreur(t : string);
begin
  AfficherTexte(1, 25, t);
end;

end.
```

3.3 Compilation d'unités ou de programmes qui utilisent des unités.

Pour chaque unité citée dans la clause *uses*, le compilateur consulte d'abord les unités résidantes (celles de Turbo.Tpl) ; si l'unité recherchée n'y est pas trouvée, le compilateur recherche, sur le disque, un fichier d'extension *.TPU* portant le même nom que l'unité (dans le répertoire courant ou dans les répertoires signalés par la directive de compilation /U). Si le fichier n'est pas trouvé sur le disque et, en fonction de la méthode choisie pour la compilation, le compilateur pourra rechercher le fichier source de l'unité et le compiler avant de poursuivre la compilation du programme principal.

Au moment du traitement de la clause *uses*, toutes les informations de la partie d'interface de chaque unité citée sont mémorisées dans la table des symboles.

4 Quelques précautions pour l'utilisation d'unités.

L'utilisation d'unités, en permettant une meilleure modularisation des programmes, permet de diminuer sensiblement les risques d'erreurs telles que l'utilisation de la même variable globale en différentes parties du programme. Toutefois, la déclaration d'objets de même nom dans la partie d'interface de deux unités différentes reste autorisée. Quand ces deux unités seront utilisées dans la même directive *uses* d'un programme ou d'une autre unité, les objets de la dernière unité citée seront pris en compte.

Si les deux objets de même nom sont de type différent, une erreur de compilation sera généralement détectée. Dans le cas contraire, les résultats sont imprévisibles.

S'il devient nécessaire de faire la différence entre deux objets de même nom, il faut se souvenir que ceux-ci peuvent être qualifiés par le nom de l'unité à laquelle ils appartiennent ; par exemple : fenetres.AfficherErreur représente la procédure AfficherErreur de l'unité **fenetres**. Dans le chapitre II.6, les routines GetMem et FreeMem, redéfinies par le programme, exécutent les routines GetMem et FreeMem de l'unité **System**.

5 Les unités standards du Turbo-Pascal 5.5.

Le Turbo-Pascal 5.5 possède huit unités standards, dont quelques unes sont rassemblées dans la bibliothèque **TURBO.TPL**, chargée en mémoire au moment de la compilation :

- l'unité **SYSTEM** qui contient les routines de bas niveau utilisées durant l'exécution ; elle est toujours utilisée par toutes les unités et tous les programmes et ce, sans qu'il soit nécessaire de la citer dans une clause *uses* ;
- l'unité **CRT** qui contient toutes les routines pour l'utilisation du clavier et de l'écran en mode texte ;
- l'unité **DOS** qui contient les routines pour exécuter les fonctions du Dos, avoir accès à la date et à l'heure du système, etc... ;
- l'unité **PRINTER** qui permet d'utiliser l'imprimante ;
- l'unité **OVERLAY** qui permet la déclaration et l'utilisation d'overlays ; elle sera détaillée dans le chapitre I.7 ;
- l'unité **GRAPH** qui implémente les routines de graphique ; elle sera étudiée dans le chapitre VII ;
- l'unité **TURBO3** qui fournit des routines pour une meilleure compatibilité avec la version 3 du Turbo-Pascal ;
- l'unité **GRAPH3** qui implémente les anciennes routines de graphique de la version 3 du Turbo-Pascal.

Nous ne détaillerons pas ici tous les objets mis à disposition par chacune de ces unités. Deux chapitres de ce cours parleront plus spécifiquement des unités **OVERLAY** et **GRAPH**. L'annexe I résume les procédures et les fonctions des unités **SYSTEM**, **CRT**, **DOS** et **PRINTER**.

6 Description et utilisation des overlays.

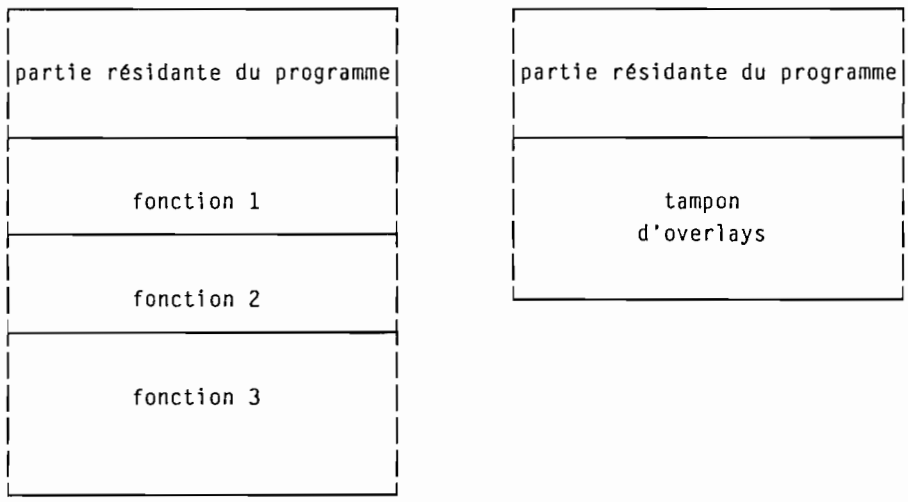
L'utilisation d'**overlay** permet d'exécuter des programmes de taille supérieure à la taille de la mémoire disponible de l'ordinateur. Déjà utilisé dans la version 3.0, le concept d'**overlay** avait disparu de la version 4.0, avant de réapparaître, sous une forme plus élaborée, dans la version 5.0.

6.1 Définitions.

Les **overlays** sont des parties de programme qui, au moment de l'exécution, occuperont à tour de rôle le même emplacement dans la mémoire. L'utilisation d'**overlays** conduit à une réduction significative de la taille totale de la mémoire nécessaire au fonctionnement d'un programme. Ainsi le programme est divisé en :

- une partie qui contient le programme principal et les routines d'intérêt général ;
- une partie qui contient les modules qui sont indépendants les uns des autres et, par conséquent, n'ont pas besoin d'être tous présents en mémoire au même moment.

Le cadre placé à gauche du schéma suivant représente l'espace total occupé par l'exécution d'un programme qui réalise trois fonctions d'un système. Les trois fonctions sont implémentées sous forme de trois modules (par exemple trois unités), appelés par un gestionnaire qui constitue le programme principal.



Le cadre placé à droite représente l'occupation mémoire du même programme principal après utilisation d'**overlays** : le programme principal a besoin du même espace mais, chaque module constituant un **overlay**, le **tampon d'overlay** n'occupe que la taille du plus grand module (dans ce cas, il s'agit de la fonction 3). Au cours de l'exécution, chacun des **overlays**, quand il sera appelé, sera chargé et exécuté au même emplacement de la mémoire.

Une unité est la plus petite partie d'un programme qui peut être utilisée comme **overlay**. A l'édition de liens, tous les **overlays** sont regroupés dans un fichier d'extension *.OVR*, alors que la partie résidante du programme reste dans le fichier d'extension *.EXE*. Dans la version 5.5, le fichier **.OVR** peut être concaténé à la fin du fichier *.EXE* pour constituer un seul

fichier ; il suffit pour ce faire d'utiliser la commande *copy* du **DOS** avec l'option /b qui permet de copier des fichiers binaires :

```
copy /b prog.exe+prog.ovr
```

Il est intéressant de noter que le terme *résidant* utilisé dans ce chapitre n'a rien à voir avec les *programmes résidants* (appelés aussi TSR-Terminate and Stay Resident) qui, à l'exemple de SideKick, restent présents en mémoire pendant l'exécution d'autres programmes. Dans notre cas, il s'agit simplement de la partie du programme qui n'a pas été implémentée sous forme d'**overlay**.

Au moment de l'exécution, le mécanisme est le suivant :

- pendant le chargement du programme exécutable, le système tente de réserver l'espace nécessaire au tampon d'overlay, entre la pile et la fin de la mémoire disponible ; si la tentative échoue, un message sera émis pour avertir l'utilisateur du manque de mémoire et le programme sera interrompu ;
- quand l'utilisation de la fonction 2 (par exemple) devient nécessaire, le programme la recherche dans le fichier d'extension *.OVR* et la charge dans le **tampon d'overlay**, où elle sera exécutée ;
- quand il sera nécessaire de faire appel à une autre fonction (la fonction 1 par exemple), le programme tentera de la charger dans le tampon, soit en remplacement de la fonction précédemment chargée, soit sans éliminer celle-ci lorsque la taille du tampon est suffisante pour contenir les deux (ce qui peut éviter un accès au disque lors d'une utilisation future de la fonction 2) ;
- quand le programme aura besoin de la fonction 3 qui, étant la plus grande, occupera tout l'espace, celle-ci sera récupérée dans le fichier *.OVR* et chargée à la place des précédentes.

Une possibilité intéressante est le chargement initial du fichier *.OVR* dans la mémoire EMS, au delà des 640 Ko de mémoire accessibles par le DOS. Ceci ne peut se faire que si l'ordinateur possède une extension de mémoire EMS (Extended Memory Specification) répondant aux spécifications LIMS (Lotus/Intel/MicroSoft). Même en utilisant cette possibilité, le **tampon d'overlay** reste nécessaire. L'utilisation de la mémoire EMS n'augmente donc pas la taille de la mémoire disponible pour le programme ou les données ; elle accélère seulement l'échange entre les **overlays** qui sera effectué de mémoire à mémoire, au lieu de disque à mémoire.

6.2 Comment utiliser une unité en overlay ?

L'implémentation d'unités en **overlays** est identique à celle d'unités classiques, à l'exception de quelques simples règles de programmation.

Toute unité susceptible d'être utilisée comme **overlay** doit avoir été compilée avec la directive $O+. D'autre part, toute procédure ou fonction déclarée dans l'unité doit être compilée avec la directive $F+, afin de pouvoir être appelée à distance. Si cette dernière exigence n'est pas respectée, aucune erreur ne sera détectée à la compilation, mais les résultats de l'exécution seront imprévisibles.

Bien qu'étant compilée avec la directive $O+, une unité peut être utilisée sans être placée en **overlay**.

Le plus simple, pour respecter les règles ci-dessus, est de placer les directives {$O+,F+} au début de chaque unité à placer en **overlay** et {$F+} au début de chacune des autres unités et du programme principal (ou placer $F+ comme option permanente du compilateur).

Dans le programme principal, la directive $O suivie du nom de l'unité permet d'obliger l'utilisation en **overlay** de l'unité en question. Par exemple :

```
program OvrEmpl;
{$F+}
uses overlay, crt, dos, fonct1, fonct2, fonct3;

{$O fonct1}
{$O fonct2}
{$O fonct3}
```

L'unité **OVERLAY**, unité standard du Turbo-Pascal, doit être citée dans la directive *uses* du programme principal, avant l'une quelconque des unités qui seront placées en **overlay**.

6.3 Limitations.

Les programmes qui utilisent des **overlays** doivent obligatoirement être compilés en utilisant le disque comme destination ; ils ne peuvent pas être compilés vers la mémoire.

Les unités **SYSTEM, CRT, OVERLAY, GRAPH, TURBO3,** et **GRAPH3** ne peuvent pas être placées en **overlay**. De manière plus générale, il en est de même de toutes les unités qui contiennent des procédures d'interruption (voir chapitre V).

Une unité ne peut pas déclarer d'autres unités comme **overlay**. Ce qui signifie qu'il ne peut exister qu'un seul niveau d'**overlay**, à l'exclusion de niveaux emboîtés.

En plus des règles énumérées ci-dessus, il est possible d'utiliser les procédures et les fonctions disponibles dans l'unité **OVERLAY**, décrite dans le chapitre suivant.

7 Unité OVERLAY.

7.1 Constantes et variables.

La variable **OvrResult**, de type *integer*, peut être utilisée comme code de retour de toutes les procédures et fonctions de l'unité **OVERLAY**. Elle peut prendre les valeurs suivantes :

- OvrOk (valeur 0) indique que tout s'est bien passé ;
- OvrError (valeur -1) indique une erreur du gestionnaire d'**overlay** ;
- OvrNotFound (valeur -2) si le fichier *.OVR* n'a pas été trouvé ;
- OvrNoMemory (valeur -3) si la mémoire est insuffisante pour le **tampon d'overlay** ;
- OvrIOError (valeur -4) en cas d'erreur de lecture du fichier *.OVR* ;
- OvrNoEMSDriver (valeur -5) si l'extension de mémoire EMS n'est pas installée ;
- OvrNoEMSMemory (valeur -6) si la mémoire EMS est insuffisante.

D'autres variables sont disponibles pour permettre au programmeur de gérer lui même le **tampon d'overlay**. Il s'agit de **OvrTrapCount, OvrLoadCount, OvrFileMode, OvrReadBuf**. Dans la presque totalité des programmes qui utilisent des **overlays**, il suffit de laisser faire le gestionnaire standard, sans intervenir. Nous citons ces variables seulement pour information. Dans des cas très spécifiques, se reporter au manuel de Turbo-Pascal pour les utiliser.

7.2 Procédures et fonctions.

La procédure OvrInit initialise le gestionnaire d'**overlay** et ouvre le fichier d'extension *.OVR* qui contient les **overlays**. Sa syntaxe est la suivante :

```
procedure OvrInit(NomFichier : string);
```

où NomFichier est le nom du fichier, éventuellement complété par le nom de l'unité logique et du chemin (nom complet). Quand le fichier *.OVR* a été placé à la fin du fichier *.EXE*, il suffit de passer comme paramètre le nom du ficher *.EXE* qui peut être récupéré sur la ligne de commande :

```
OvrInit(ParamStr(0));
```

Elle doit être appelée avant toutes les autres procédures et fonctions qui sont en relation avec les **overlays**, et avant l'allocation des variables dynamiques (voir chapitre II).

La procédure **OvrInitEMS** permet de charger le fichier *.OVR* dans la mémoire EMS. Elle n'utilise pas de paramètres :

```
procedure OvrInitEMS;
```

Si la mémoire EMS n'est pas installée ou si elle est de taille insuffisante pour recevoir le fichier **.OVR**, le programme continuera à fonctionner normalement en utilisant le disque. **OvrInitEMS** peut donc être appelée systématiquement.

D'autres routines sont disponibles et peuvent être appelées dans des cas spécifiques. Elles interviennent dans la gestion du **tampon d'overlays**. Il s'agit de :

```
function OvrGetBuf : longint;
procedure OvrSetBuf(Taille : longint);
procedure OvrClearBuf;
procedure OvrSetRetry(Taille : longint);
function OvrGetRetry : longint;
```

8 Recommandations pour l'utilisation d'overlays.

La structure du programme doit être étudiée avec soin pour choisir les unités à placer en **overlays**. Une unité susceptible d'être appelée souvent par une autre, placée également en **overlay**, ne doit pas être placée en **overlay**. Par exemple, quand l'unité d'administration d'un fichier est placée en **overlay**, il ne peut pas en être de même avec l'unité qui contient les routines d'accès au fichier : si c'était le cas, chaque fois qu'il serait nécessaire d'exécuter une de ces routines, il faudrait procéder à un échange d'**overlays**, ce qui nuirait aux performances du programme. A moins que, par hasard ou par une gestion adéquate du **tampon d'overlay**, les deux unités puissent cohabiter en mémoire au même instant.

Il est également conseillé d'éviter de placer en **overlay** les unités qui comportent une partie d'initialisation. Bien qu'elle soit exécutée une seule fois, cette partie d'initialisation serait chargée à chaque échange d'**overlay** et occuperait inutilement de l'espace en mémoire. Il vaut mieux regrouper toutes les parties d'initialisation d'unité dans une autre unité également placée en **overlay**. Elle sera ainsi chargée et exécutée au début du programme puis oubliée.

Pour initialiser l'utilisation du fichier **.OVR**, il vaut mieux implémenter une petite unité avec une partie d'initialisation qui ouvre le fichier d'**overlay** et réserve l'espace mémoire. Cette unité peut être citée dans la directive *uses*, avant toute autre. Se reporter à l'exemple présenté plus bas qui implémente un contrôle complet des erreurs, sans gestion personnalisée du **tampon d'overlay**.

Le premier listing présente un exemple d'unité pour initialiser l'usage des **overlays** :

```
{$O+,F+}
unit fitoovr;
interface
uses overlay;
implementation
begin
  OvrInit('SISFITO.OVR');
  case OvrResult of
    OvrNotFound : {erreur : le fichier SISFITO.OVR n'a pas été trouvé};
    OvrError    : {erreur : autre erreur dans la gestion des overlays};
  end;

  OvrInitEMS;
  case OvrResult of
    OvrError       : {erreur : autre erreur dans la gestion des overlays};
    OvrNoEMSDriver : {erreur : pilote de la mémoire EMS non installé};
    OvrNoEMSMemory : {erreur : taille de la mémoire EMS insuffisante};
  end;

end.
```

Le second présente le schéma du programme principal correspondant :

```
{$M 16384,0,655360}

program sisfito;

uses
  dos,
  crt,
  global,  {routines d'intérêt général}
  fitoovr, {initialisation du système d'overlays}
  sistab,  {gestion des tables}
  fito001, {gestion de l'herbier}
  fito002, {édition de l'herbier}
  fito003; {gestion de prêts}

{variables du programme principal}

begin
  {initialisations}
  ..............
  {gestionnaire de menu}
  ..............
  {appel des fonctions du système}
end.
```

Chapitre II

Pointeurs et utilisation de la mémoire dynamique

Contrairement au cas du langage C, il est possible de faire beaucoup de choses en Turbo-Pascal sans aborder l'utilisation des pointeurs. Il est pourtant une situation où l'usage des pointeurs devient indispensable : il s'agit de l'utilisation du tas pour mémoriser des tableaux de grandes dimensions ou des variables temporaires.

1 Propriété des pointeurs.

Un type *pointeur* est identifié par le symbole ^ placé devant le type de la variable. Par exemple, les déclarations qui suivent :

```
type
  PointeurEntier = ^integer;

var
  Entier1,
  Entier2 : PointeurEntier;
```

signifient que Entier1 et Entier2 sont des variables pointeurs qui pointent vers une adresse de la mémoire dont le contenu sera interprété comme étant la valeur d'un entier.

Du point de vue syntaxique, Entier1 représente l'adresse d'un entier, alors que Entier1^ représente le contenu (ou la valeur) de l'entier.

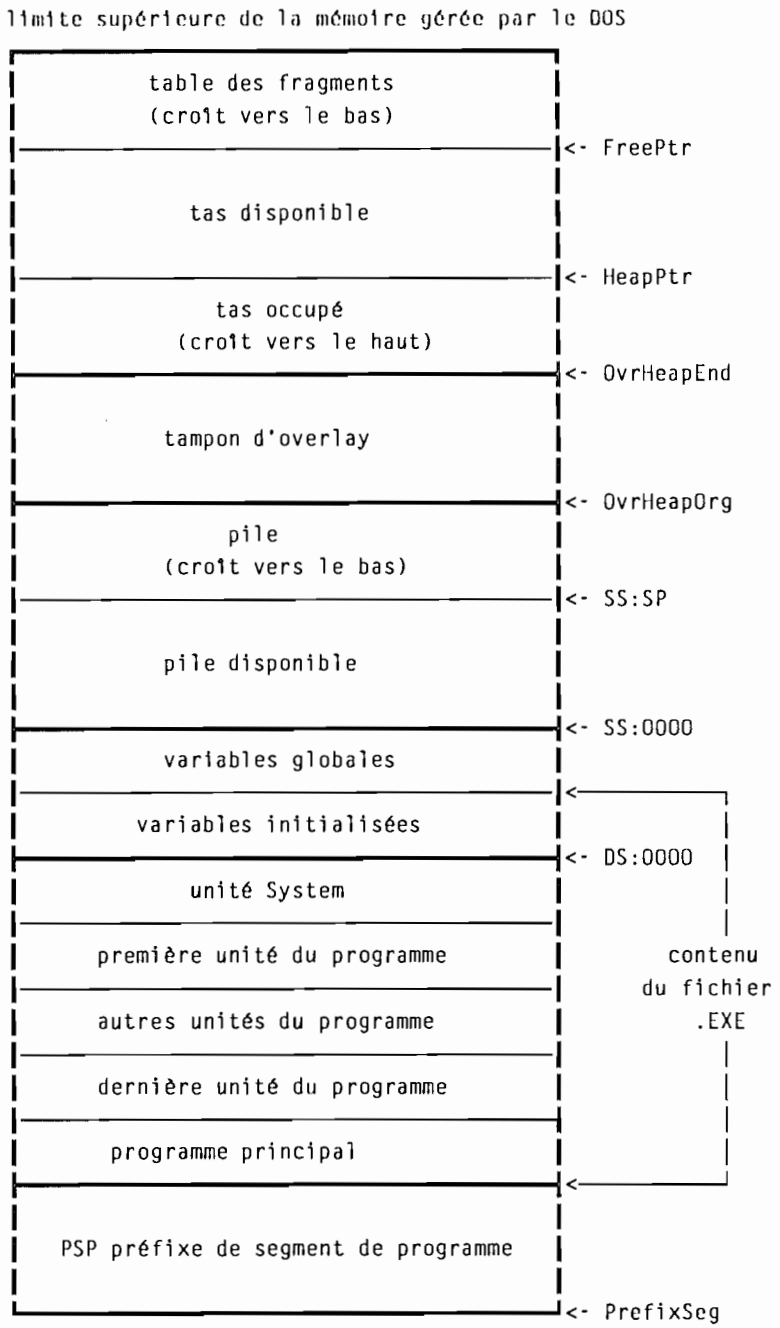
Physiquement, une variable de type pointeur, puisqu'elle contient une adresse longue, occupe 4 octets : deux pour le segment et deux pour le déplacement à l'intérieur du segment. On a coutume de représenter la valeur d'un pointeur par la valeur en hexadécimal du segment, suivi du symbole : et suivi de la valeur en hexadécimal du déplacement. Par exemple, un pointeur de valeur 0A6B:001F pointe vers l'adresse absolue 0A6CF, c'est à dire 0A6Bx10 + 1F (valeurs exprimées en hexadécimal).

Un pointeur peut désigner l'adresse d'une variable quelconque, y compris celle d'une structure ou d'un autre pointeur. Il faut signaler également que :

- le type *pointer* est un type standard du Turbo-Pascal qui désigne un pointeur vers une variable de type quelconque et constitué, comme tout pointeur, d'un segment et d'un déplacement ;
- la valeur *nil* est compatible avec tous les types de pointeurs et peut être utilisée pour représenter la valeur d'un pointeur dont la valeur n'a pas de signification, c'est à dire un pointeur qui ne pointe pas vers un objet défini ; *nil* équivaut à la valeur 0000:0000 ;
- les *opérateurs de comparaison* peuvent être utilisés pour comparer des variables de type pointeur ;
- on appelle *pointeur normalisé* un pointeur dont le déplacement à l'intérieur du segment est une valeur entre 00 et 0F.

2 Utilisation de la mémoire dynamique en Turbo-Pascal.

La zone de mémoire appelée tas occupe la partie disponible de la mémoire pendant l'exécution d'un programme. Elle est principalement utilisée pour contenir des variables (scalaires ou matricielles) qualifiées de *dynamiques* en raison des caractéristiques du *gestionnaire du tas* qui permet de les allouer puis de les libérer après utilisation.



Le grand avantage de l'utilisation de la mémoire dynamique est de permettre l'adaptation d'un programme aux possibilités de l'appareil sur lequel il doit être exécuté : au lieu de définir une matrice comme variable globale, de taille pré-définie, il est préférable de la définir au moment de l'exécution, en utilisant toute la mémoire disponible sur l'ordinateur utilisé. Un autre avantage est que cet espace, après libération de la variable, pourra être réutilisé par une autre variable, ce qui peut aider à dépasser les limitations dues au manque de mémoire.

La figure 1 représente, de façon schématisée, l'occupation de la mémoire durant l'exécution d'un programme écrit en Turbo-Pascal 5.0, et sera utile pour une meilleure compréhension de ce chapitre.

Normalement, l'utilisation de variables dynamiques consiste à :

- définir un pointeur de type adéquat pour pointer vers la variable qui sera utilisée,
- allouer la variable dans la mémoire dynamique (tas),
- utiliser la variable, de la même manière qu'une variable statique,
- libérer, sur le tas, l'espace occupé par la variable.

Comme on peut le voir sur la figure 1, le tas est divisé en deux parties :

- une première partie où sont placées les variables dynamiques,
- une seconde partie, appelée *table des fragments*, où sont gardées les adresses des espaces disponibles dans la première partie.

Turbo-Pascal met à la disposition du programmeur, plusieurs objets (variables globales, fonctions et procédures), détaillés dans les prochains chapitres, afin de réaliser les opérations nécessaires à l'utilisation de variables dynamiques.

3 Variables globales en relation avec l'utilisation de mémoire dynamique.

Les variables suivantes sont déclarées dans l'unité **SYSTEM** (toujours utilisée par un programme en Turbo-Pascal) et directement mises à jour par le gestionnaire de tas. Elles sont donc directement utilisables par le programmeur.

FreePtr pointe vers la fin de la table des fragments qui croît vers le bas, à chaque fois que de nouveaux trous deviennent disponibles sur le tas. Il faut noter que si le dernier espace libéré est voisin d'un espace déjà disponible, les adresses seront réorganisées pour obtenir une seul espace disponible ; ce qui signifie que la table des fragments peut également diminuer au moment de la libération de variables, jusqu'à disparaître quand plus aucune variable n'est allouée (tas complètement disponible).

HeapOrg pointe vers le début (limite inférieure) du tas ; sa valeur est connue immédiatement après le chargement du programme et doit rester la même jusqu'à la fin de son exécution.

HeapPtr pointe vers la fin (limite supérieure) du tas ; après chaque opération d'allocation ou de libération, sa valeur est remise à jour.

Deux procédures permettent d'allouer des variables dans la mémoire dynamique : il s'agit de **New** et **GetMem**. Deux méthodes permettent de libérer l'espace utilisé par des variables dynamiques précédemment allouées, la première utilise les procédures **Dispose** ou **FreeMem**, la seconde utilise **Mark** et **Release**.

4 Procédures et fonctions disponibles pour utiliser la mémoire dynamique.

4.1 Comment obtenir des informations sur la mémoire disponible.

La fonction **MemAvail** fournit la taille totale de la fraction du tas disponible, en ajoutant à l'espace disponible entre **HeapPtr** et **FreePtr**, la taille de tous les trous enregistrés dans la table des fragments.

La fonction **MaxAvail** fournit la taille du plus grand bloc de mémoire contiguë : il s'agit en fait du plus grand trou ou de l'espace disponible entre **HeapPtr** et **FreePtr**.

4.2 Comment allouer une variable.

Deux procédures sont disponibles :

- procedure **GetMem**(var p : pointer; t : word);

qui alloue un espace de t octets dont l'adresse est retournée dans le pointeur p ; la variable ainsi définie peut ensuite être référencée par p^ ;

- procedure **New**(var p : pointer);

qui réalise une opération similaire à celle réalisée par GetMem où t est remplacé par la taille de la variable pointée par p ; en d'autres termes New(p) est équivalent à GetMem(p, sizeof(p^)).
Dans les deux cas, si le plus grand trou disponible dans le tas est de taille inférieure à t ou sizeof(p^), il se produira une erreur **203 Heap overflow error**, p deviendra égal à *nil* et l'exécution du programme sera interrompue. Pour éviter ce problème, il est recommandé d'utiliser la fonction MaxAvail pour contrôler la situation du tas avant de tenter d'allouer la variable. Ce qui peut se faire ainsi :

```
if MaxAvail >= sizeof(p^)
  then New(p);
  else {mémoire insuffisante}
```

4.3 Comment libérer de l'espace dans la mémoire dynamique.

Pour libérer l'espace occupé par une variable dynamique, nous disposons également de deux procédures :

- procedure **FreeMem**(var p : pointer; t : word);

qui libère un espace de t octets, commençant à l'adresse pointée par p ;

- procedure **Dispose**(var p : pointer);

qui libère un espace de la taille de p^ qui commence à l'adresse pointée par p.

Une fois que la variable p a été libérée, il est interdit de l'utiliser, puisque p ne pointe plus vers une adresse valide de la mémoire dynamique (après libération, p est généralement égal à *nil*). Il faut ajouter que, au moment de l'appel à **FreeMem** ou **Dispose**, p doit pointer vers une adresse valide du tas.

Les pointeurs retournés par **GetMem** et **New** sont normalisés (voir définition plus haut) et, ce qui est très important, les pointeurs passés comme paramètres à **FreeMem** et **Dispose** **doivent être normalisés** : au cas où ils ne le seraient pas, aucun message d'erreur ne sera émis à l'exécution mais la mémoire ne sera pas libérée correctement, ce qui entraînera plus tard des erreurs fatales du gestionnaire de tas.

Une autre méthode permet de libérer toute la mémoire dynamique située au dessus d'une certaine adresse, indépendamment des variables allouées dans cet espace. Pour cela :

- il est tout d'abord nécessaire de mémoriser la limite supérieure du tas, par exemple en début de programme, ce qui peut être réalisé par :

procedure **Mark**(var p : pointer);

qui donne à p la valeur actuelle de la limite supérieure (en fait, l'instruction Mark(p) est équivalente à p := HeapPtr) ;

- et ensuite, après allocation et utilisation de variables dynamiques, libérer l'espace par :

procedure **Release**(p : pointer);

qui place dans HeapPtr la valeur de p (attention : Release(p) n'est pas équivalent à HeapPtr := p, car Release réorganise également la table des fragments).

Comme pour la méthode antérieure, une valeur allouée dans l'espace libéré ne peut plus être utilisée (sans être à nouveau allouée). Les manuels de Turbo-Pascal conseillent l'utilisation de Freemem et Dispose plutôt que Mark/Release, car cette dernière méthode oblige le programmeur à connaître parfaitement quelles sont les variables qui ont été libérées.

5 Comment éviter le débordement de la table des fragments.

Dans la table des fragments, sont conservées les adresses des espaces disponibles de la mémoire dynamique, libérés par **FreeMem** ou **Dispose**. **FreePtr** qui pointe vers la table des fragments, bien qu'étant déclaré comme *pointer*, est en réalité un pointeur vers une table de structures de type **FreeRec** :

```
type
  FreeRec = record
              OrgPtr,
              EndPtr : pointer;
            end;
FreeList = array[0..8191] of FreeRec;
FreeListP = ^FreeList;
```

Chaque bloc de mémoire libéré est défini par son adresse de début (OrgPtr) et son adresse de fin (EndPtr) qui sont des pointeurs normalisés. A tout moment, le nombre d'éléments de la table des fragments peut être calculé par la formule :

```
Compteur := (8192 - (ofs(FreePtr) div 8)) mod 8192;
```

ce qui signifie que la table des fragments peut contenir jusqu'à 8191 éléments.

Quand une ou plusieurs variables dynamiques non contiguës sont libérées, la table des fragments croît vers le bas pour mémoriser les adresses des espaces libérés. Ceci fonctionne tant qu'il y a suffisamment d'espace entre HeapPtr et FreePtr et tant que le nombre de fragments est inférieur à 8191. Dans le cas contraire une erreur interrompt l'exécution.

Pour prévenir le premier cas de figure, il est possible d'utiliser la variable **FreeMin** en lui donnant une valeur minimum de l'espace entre **HeapPtr** et **FreePtr**. A chaque appel de **GetMem** et **New**, le gestionnaire du tas vérifie que l'allocation de mémoire demandée ne va pas réduire l'espace disponible à une taille inférieure à **FreeMin**. Si c'est le cas, la variable ne sera pas allouée.

La prévention contre le second cas de figure fait l'objet du chapitre suivant.

6 Taux de granularité du gestionnaire de tas.

Par défaut, la granularité du gestionnaire de tas est de un octet. Ce qui signifie que si on alloue une variable de un octet, un seul octet sera effectivement occupé ; ce qui signifie également que, quand cette variable sera libérée, **un octet sera libéré mais huit seront occupés** dans la table des fragments pour conserver l'adresse de l'espace libéré (si celui-ci n'est pas voisin d'un espace déjà libéré). Cet exemple peut paraître peu réaliste, mais la même chose peut se produire quand on alloue des espaces de taille variable comme, par exemple, les lignes d'un traitement de texte. Considérons l'exemple de l'allocation puis de la libération d'une ligne de 50 octets : si l'allocation suivante nécessite 49 octets, il restera un espace d'un octet mémorisé comme tel dans la table des fragments. L'accumulation de ces espaces de petite taille conduit à une fragmentation excessive de la mémoire disponible, ce qui peut entraîner plus de 8191 fragments et produire une erreur d'exécution irrécupérable.

La solution pour éviter ce genre de problèmes est d'augmenter la *granularité* du gestionnaire du tas. Pour cela, le programme est obligé de définir et d'utiliser ses propres routines d'allocation et de libération de mémoire. On trouve ci-dessous les procédures GetMem et FreeMem qui permettront de maintenir une granularité de 16 octets :

```
procedure GetMem(var p : pointer; t : word);
begin
  t := succ(t div ,16)*16;
  system.getmem(p, t);
end;

procedure FreeMem(var p : pointer; t : word);
begin
  t := succ(t div 16)*16;
  system.freemem(p, t);
end;
```

Dans la plupart des cas, cependant, la limite de 8191 sera suffisante sans autre précaution.

7 Exemples.

7.1 Visualiser l'état de la mémoire dynamique.

L'unité **VisuTas**, disponible dans le fichier VISUTAS.PAS, peut être utilisée dans un programme quelconque pour montrer, à certains moments, l'état de la mémoire dynamique : espace disponible, taille du plus grand espace disponible, nombre et taille des trous de mémoire disponibles, etc...

En appelant la procédure MontrerTas, ces informations sont montrées dans une fenêtre large de 80 colonnes, ouverte sur la ligne choisie par le programmeur et comportant un titre passé en paramètre. Il suffit alors d'appuyer sur une touche quelconque pour montrer plus d'informations ou effacer la fenêtre et revenir au programme appelant.

7.2 Allocation de variables.

Le programme **AllocTas**, imprimé ci-dessous, ne fait qu'allouer puis libérer des variables dynamiques de divers types. Il fait également appel à l'unité **VisuTas** pour montrer l'état du tas après chaque opération.

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{          exemple d'allocation et libération de variables dynamiques        }
{                    (disponible dans le fichier alloctas.pas)               }
{----------------------------------------------------------------------------}
uses visutas, utiltas;
type
  TypeMat = array[1..MaxInt] of integer;
  TypeStr = array[1..50] of string;

var
  PtrMat  : ^TypeMat;
  PtrMat1 : ^TypeMat;
  PtrStr  : ^TypeStr;
  p       : pointer;
  s       : ^string;
  i : word;

{$F+}
function ErreurTas(t : word) : integer;
{$F-}
begin
  ErreurTas := 1;
end;

begin
  HeapError := @ErreurTas;
  MontrerTas(1, 'au début');
  if MaxAvail >= sizeof(PtrMat)
    then begin
      new(PtrMat);
      MontrerTas(1, 'après allocation d''un vecteur d''entiers');
    end
    else {Mémoire insuffisante};

  getmem(s, 50);
  if s <> nil
    then MontrerTas(1, 'après allocation d''une chaîne de 50 caractères')
    else {Mémoire insuffisante};

  getmem(PtrMat1, 500*sizeof(integer));
  if PtrMat1 <> nil
    then MontrerTas(1, 'après allocation d''un vecteur incomplet d''entiers')
    else {Mémoire insuffisante};

  new(PtrStr);
  if PtrStr <> nil
    then MontrerTas(1, 'après allocation d''un vecteur de chaînes de caractères')
    else {Mémoire insuffisante};

  {libération d''une chaîne de caractères sur deux-remarquez la normalisation du pointeur}

  if PtrStr <> nil
   then for i := 1 to 50 do
    if i mod 2 = 0
      then begin
        p := PtrStr;
        p := ptr(seg(p^), ofs(p^)+pred(i)*sizeof(string));
        freemem(PtrNormalise(p), sizeof(string));
```

```
    end;
  MontrerTas(1, 'après libération de la moitié des chaînes de caractères');

  {liberation de l''autre moitié des chaînes-remarquez la normalisation du pointeur}
  if PtrStr <> nil
   then for i := 1 to 50 do
    if i mod 2 = 1
      then begin
        p := PtrStr;
        p := ptr(seg(p^), ofs(p^)+pred(i)*sizeof(string));
        freemem(PtrNormalise(p), sizeof(string));
      end;
  MontrerTas(1, 'après libération de toutes les chaînes');

  if PtrMat <> nil
    then dispose(PtrMat);
  MontrerTas(1, 'après libération du vecteur d''entiers complet');

  if PtrMat1 <> nil
    then freemem(PtrMat1, 500*sizeof(integer));
  MontrerTas(1, 'après libération du vecteur d''entiers incomplet');

  if s <> nil
    then freemem(s, 50);
  MontrerTas(1, 'après libération de la chaîne de 50 caractères');
end.
```

Il faut signaler que la méthode utilisée dans ce programme permet de le protéger contre le manque de mémoire : la procédure ErreurTas contrôle les erreurs susceptibles de se produire pendant l'utilisation de **GetMem** et **New**. La routine standard, pointée par **HeapError**, retourne toujours la valeur 0, ce qui provoque une erreur d'exécution quand la mémoire disponible devient insuffisante pour contenir la nouvelle variable à allouer. Il vaut mieux implémenter une routine du type de ErreurTas qui retourne une valeur différente de zéro en cas d'erreur et tester la valeur du pointeur après la tentative d'allocation (en cas d'erreur, **GetMem** et **New** retournent un pointeur avec la valeur *nil*) :

```
{$F+} function ErreurTas(t : word)  : integer; {$F-}
begin
  ErreurTas := 1;
end;
```

Pour remplacer la routine standard par la nouvelle routine, il suffit d'affecter à la variable **HeapError**, l'adresse de cette dernière :

```
HeapError := @ErreurTas;
```

Enfin, dans le programme AllocTas, le lecteur pourra découvrir une autre méthode de contrôle : il s'agit de vérifier la taille de l'espace disponible avant d'appeler la routine d'allocation. Ceci a été fait dans le cas de l'allocation du tableau complet d'entiers.

7.3 Allocation de lignes de texte.

Le prochain exemple présente ce qui pourrait être la définition d'une ligne de texte dans un traitement de texte. Comme indiqué dans les routines d'allocation et de destruction de

lignes, aussi bien la description de la ligne que son contenu (texte) sont alloués sur le tas. Les lignes sont liées entre elles par l'intermédiaire des pointeurs LigPrec et LigSuiv.

```
{---------------------------cours de Pascal-Avancé--------------------------}
{                                                                            }
{            exemple incomplet d'allocation de lignes de texte               }
{----------------------------------------------------------------------------}
const
  MaxLigne = 255;

type
  LigText = string[MaxLigne];      {type : ligne de texte}
  PtrText = ^LigText;              {type : pointeur vers une ligne de texte}
  PtrLign = ^LigDesc;              {type : pointeur vers une description de ligne}
  LigDesc  = record                {type : description de ligne}
               LigSuiv,            {pointeur vers la ligne précédente}
               LigPrec : PtrLign;  {pointeur vers la ligne suivante}
               Texte   : PtrText;  {pointeur vers le texte contenu dans la ligne}
               Flags,              {flags pour caractériser la ligne}
               TextLen : integer;  {taille du texte de la ligne}
             end;

function NouvelleLigne(Taille : integer) : PtrLign;
{alloue une nouvelle ligne avec Taille octets réservés pour le texte}
var
  Ligne    : PtrLign;
  TailLigne : integer;

begin
  if (Maxavail > 0) and (Maxavail <= (Taille + sizeof(LigDesc)))
    then begin                     {si taille insuffisante, sortir}
      NouvelleLigne := nil;
      exit
    end;
  Getmem (Ligne,sizeof(LigDesc));   {alloue l'espace nécessaire à la description de la ligne}
  TailLigne := succ(Taille);        {taille du texte = Taille + 1      }
  Getmem(Ligne^.Texte, TailLigne);  {alloue Taille+1 espaces pour le texte}
  Ligne^.TextLen := pred(TailLigne);{initialise la taille du texte}
  Fillchar(Ligne^.Texte^,TailLigne, ' '); {remplit le texte de blancs}
  Ligne^.Flags := 0;                {initialise les flags             }
  NouvelleLigne := Ligne;
end;

procedure DetruireLigne(var Ligne : PtrLign);
{libère l'espace occupé par une ligne de texte}
begin
  FreeMem(Ligne^.Texte,succ(Ligne^.TextLen)); {libère l'espace occupé par le texte}
  FreeMem(Ligne,sizeof(LigDesc));             {libère l'espace occupé par la description}
end;
```

8 Exercices.

a/ Ecrire une fonction en Turbo-Pascal pour normaliser un pointeur. L'en-tête de la fonction peut être défini ainsi :

function PtrNormalise(p : pointer) : pointer;

b/ Ecrire une fonction pour transformer la valeur d'un pointeur en sa représentation usuelle (segment:déplacement) en valeurs hexadécimales ; exemple : 0A23:0052.

c/ Implémenter une unité avec les procédures nécessaires à la définition d'une matrice d'entiers, à deux dimensions, et qui utilise au maximum la mémoire disponible. La partie d'initialisation doit effecteur la réservation de l'espace disponible. Trois routines doivent être mises à disposition de l'utilisateur pour mémoriser une valeur X dans la position (I, J) de la matrice, récupérer la valeur de la position (I, J) et finalement libérer l'espace occupé initialement.

Chapitre III

Turbo-Access : gestionnaire de fichier séquentiel indexé.

Turbo-Access est un sous ensemble du Turbo-Toolbox, distribué séparément du Turbo-Pascal. Il s'agit d'un gestionnaire de fichiers structurés en arbres binaires (B-Tree), avec des fonctions comparables à celles disponibles en langage COBOL pour les fichiers déclarés avec *Organization Indexed Sequencial*, ce qui permet des recherches et des insertions très rapides. Bien qu'il ne soit pas nécessaire de connaître la théorie des B-Tree pour utiliser Turbo-Toolbox, celle-ci est décrite dans le manuel du Turbo-Toolbox. Cette théorie ne sera pas reprise dans ce support de cours ; seules seront données, dans le premier chapitre, les définitions indispensables. Nous détaillerons ensuite l'utilisation et l'installation du Turbo-Access, ainsi qu'un exemple.

1 Informations générales.

1.1 Définitions.

Le fichier logique à gérer est vu par le Turbo-Access comme deux fichiers physiques ou plus :

- le fichier des données, décrit comme *DataFile*, qui contient les informations à gérer ; chaque élément du fichier de données est un *enregistrement* ; un fichier de données peut contenir plus de deux milliards d'enregistrements et la taille maximum d'un enregistrement est de 65535 octets ;
- un ou plusieurs index, décrits comme *IndexFile*, où sont stockées les *clés* qui identifient un enregistrement de données.

La clé doit être constituée par un ou plusieurs champs de l'information à gérer (champs du fichier de données), de telle manière que le contenu d'une clé permette d'identifier complètement l'enregistrement correspondant dans le fichier de données. Toutefois, un paramètre du Turbo-Access autorise l'existence de clés dupliquées, ce qui nécessite quand même un travail supplémentaire de la part du programmeur pour différencier les données qui correspondent à deux clés identiques. A l'intérieur de chaque index, à chaque clé est associé un numéro d'enregistrement de données. Ainsi, chaque élément d'un fichier d'index est composé d'une clé et d'un numéro d'enregistrement, bien que, physiquement, ces éléments soient regroupés en enregistrements de n éléments appelés *pages*, n étant *la taille de la page.*

Pour un fichier déterminé, au moins un index doit être défini, sinon ce ne serait plus un fichier indexé. Par contre, l'existence du fichier de données n'est pas obligatoire : si tous les champs de données à gérer font partie de la clé, il suffit de gérer l'index.

Turbo-Access met à la disposition du programmeur deux ensembles de routines pour créer, administrer et supprimer des fichiers séquentiels indexés :

- un ensemble de routines de bas niveau qui permet d'implémenter toutes les possibilités offertes ;
- un ensemble de routines de plus haut niveau qui facilite le travail du programmeur en accédant au fichier sous une forme globale, sans distinguer le fichier de données et

l'index ; ce qui nécessite moins d'efforts de programmation ; en contrepartie, cette méthode ne supporte pas les clés dupliquées, ni les index multiples.

Dans de nombreux cas, la seconde méthode est suffisante. Il est également possible d'utiliser les deux méthodes en même temps, moyennant quelques précautions.

1.2 Types et déclarations.

Turbo-Access met à la disposition du programmeur des types et des variables prédéfinies. Pour utiliser les routines du Turbo-Access, il est nécessaire de respecter les règles qui vont être énumérées ci-dessous (les noms de variables des exemples seront utilisés également, dans les prochains chapitres, pour décrire les routines) :

- le fichier à gérer doit être déclaré du type **DataSet** afin d'être utilisé par les routines de haut niveau ; par exemple :

```
var
  Fichier : DataSet;
```

- ou alors, on doit déclarer un fichier de données du type **DataFile** et un index de type **IndexFile** pour travailler avec les routines de bas niveau ; par exemple :

```
var
  FichierDonnees : DataFile;
  FichierIndex   : IndexFile;
```

- le type d'enregistrement du fichier doit être déclaré comme une structure, en réservant une variable de type *longint* comme premier champ pour contenir l'état de l'enregistrement ; par exemple :

```
type
  EnregFichier = record
                   Status : longint;
                   {champs d'information}
                 end;
```

- le type de la clé doit être déclaré comme *string* et avec la taille exacte de la clé du fichier ; par exemple :

```
type
  EnregCle = string[10];
```

- le programmeur peut alors déclarer au moins une variable comme enregistrement du fichier et une variable comme clé :

```
var
  Enreg : EnregFichier;
  Cle   : EnregCle;
```

- dans le cas où l'on souhaite travailler avec les routines de bas niveau, il est également nécessaire de déclarer une variable de type *longint*, qui correspond au numéro de l'enregistrement de données ; par exemple :

```
var
  NumEnreg : longint;
```

- en cas d'utilisation des routines de haut niveau, la variable globale **TARecNum** contient toujours le numéro du dernier enregistrement accédé et peut être utile quand les deux méthodes (haut et bas niveau) sont utilisées pour le même fichier ;

- enfin, la variable booléenne **Ok** est utilisée comme code de retour de nombreuses routines des deux niveaux et peut être testée par le programme appelant.

2 Routines de haut niveau-unité TAHigh.

Les routines décrites ci-dessous font partie de **tahigh**, mais la variable **Ok**, utilisée comme code de retour, est définie dans l'unité **taccess**. En conséquence, pour utiliser ces routines, il est nécessaire de déclarer les deux unités (**taccess** et **tahigh**) dans la directive *uses* du programme. Comme **tahigh** utilise aussi **taccess**, cette dernière doit être citée en premier.

2.1 Créer un fichier séquentiel indexé (TACreate).

Pour créer un fichier séquentiel indexé, utiliser la procédure **TACreate** :

```
procedure TACreate(var Fichier     : DataSet;
                       NomFichier  : FileName;
                       TailleEnreg : integer;
                       NomIndex    : FileName;
                       TailleCle   : integer);
```

Fichier est une variable déclarée dans le programme appelant, de type *DataSet*. **NomFichier** et **NomIndex** sont les noms physiques du fichier de données et du fichier d'index associés à la variable **Fichier**. **TailleEnreg** et **TailleCle** sont respectivement la taille de l'enregistrement de données et la taille de la clé ; on a coutume de les définir en utilisant l'opérateur *sizeof* : sizeof(EnregFichier) pour **TailleEnreg** et pred(sizeof(EnregCle)) pour **TailleCle**.

TACreate crée le fichier de données et l'index et les laisse ouverts et prêts à l'utilisation. La variable Ok n'est pas affectée par l'exécution de **TACreate**. Seules des erreurs du DOS (répertoire ou disque plein) peuvent se produire durant l'exécution.

Voir exemple au paragraphe 2.2.

2.2 Ouvrir un fichier séquentiel indexé (TAOpen).

Pour ouvrir un fichier séquentiel indexé, on utilise la procédure **TAOpen** :

```
procedure TAOpen(var Fichier     : DataSet;
                     NomFichier  : FileName;
                     TailleEnreg : integer;
                     NomIndex    : FileName;
                     TailleCle   : integer);
```

Les paramètres sont identiques à ceux de **TACreate**. Toutefois, la variable **Ok** prendra la valeur *false* si le fichier ne peut pas être ouvert. Dans l'exemple ci-dessous, le programme tente d'ouvrir le fichier et, s'il n'y arrive pas, il fait appel à **TACreate** pour le créer :

```
TAOpen(FicPrix, 'PRIX.DAT', sizeof(EnregPrix),
                'PRIX.IND', pred(sizeof(EnregClePrix)));
if not Ok
  then TACreate(FicPrix, 'PRIX.DAT', sizeof(EnregPrix),
                         'PRIX.IND', pred(sizeof(EnregClePrix)));
```

2.3 Ecrire un enregistrement dans le fichier (TAWrite, TAInsert, TAUpdate).

TAWrite doit être utilisé pour écrire un nouvel enregistrement ou réécrire un enregistrement mis à jour. Voici le prototype de la procédure :

```
procedure TAWrite(var Fichier : DataSet;
                  var Enreg;
                  var Cle);
```

Fichier doit être une variable de type *DataSet*, qui représente un fichier séquentiel indexé ouvert par **TAOpen** ou **TACreate**. **Enreg** est une variable de type **EnregFichier** et **Cle** est une variable de type **EnregCle**. Il est évident que tous les champs de **Enreg** doivent déjà avoir été remplis avec les valeurs correctes, y compris le champ Status qui doit avoir été mis à zéro. De la même façon, **Cle** doit avoir été construite avec les valeurs correspondantes des champs de l'enregistrement à écrire.

Si la clé existe déjà, l'enregistrement va remplacer celui qui existe déjà. Dans le cas contraire, l'enregistrement et la clé seront inclus respectivement dans le fichier de données et dans l'index. La variable **Ok** n'est pas affectée par l'exécution de **TAWrite** : la seule erreur possible est le manque d'espace sur le disque, ce qui provoquera une erreur du DOS.

Dans l'exemple présenté ci-dessous, l'enregistrement Prix et la cle ClePrix sont remplis avant l'appel de **TAWrite** pour écrire l'enregistrement sur disque :

```
with Prix do
  begin
    Nom := 'Brésil';
    Date := '900325';
    Result[1] := 01;
    ................
    Result[6] := 20;
    Status := 0;
  end;
ClePrix := Prix.Nom;
TAWrite(FicPrix, Prix, ClePrix);
```

TAWrite permet d'écrire des enregistrements sans se préoccuper si l'enregistrement existait déjà avec la même clé. Parfois, cette information est nécessaire pour le programme appelant, par exemple pour totaliser le nombre d'enregistrements nouveaux et le nombre d'enregistrements remplacés. Dans ce cas on préfèrera l'utilisation de **TAUpdate** et **TAInsert**, au lieu de **TAWrite**.

TAInsert, avec les mêmes paramètres que **TAWrite**, va tout d'abord contrôler l'existence dans le fichier d'un enregistrement ayant la même clé que celle passée en paramètre. S'il en existe déjà un, l'enregistrement ne sera pas inséré et **Ok** reviendra avec la valeur *false*. S'il n'en existe pas, l'enregistrement sera inséré et **Ok** reviendra avec la valeur *true*.

La logique de la procédure **TAUpdate** est semblable. Elle utilise également les mêmes paramètres, mais n'écrit l'enregistrement que si il est déjà présent dans le fichier.

L'utilisation de **TAWrite** présentée ci-dessus peut être remplacée par :

```
TAUpdate(FicPrix, Prix, ClePrix);
if Ok
  then inc(TotRemplaces) {enregistrement remplacé}
  else begin             {enregistrement inséré  }
    TAInsert(FicPrix, Prix, ClePrix);
    inc(TotInclus);
  end;
```

2.4 Lire un enregistrement du fichier (TARead).

La manipulation des enregistrements d'un fichier séquentiel indexé est basée sur les clés des enregistrements. Pour rechercher un enregistrement, il est donc nécessaire de définir d'abord la valeur de la clé, avant d'appeler la routine **TARead** :

```
procedure TARead(var Fichier : DataSet;
                 var Enreg;
                 var Cle;
                     RechercheExacte : boolean);
```

Fichier doit être un fichier de type *DataSet*, déjà ouvert. **Cle** et **Enreg** sont respectivement la clé à rechercher et la variable où sera retourné le contenu de l'enregistrement lu. Rappelons que la valeur de la clé doit avoir été définie avant l'appel. **RechercheExacte** est une variable booléenne pour préciser si on désire rechercher l'enregistrement dont la clé correspond exactement à la clé passée comme paramètre (valeur *true*), ou si on souhaite seulement récupérer le premier enregistrement dont la clé commence par les caractères de celle passée comme paramètre (valeur *false*). **Ok** revient avec la valeur *true* si un enregistrement a été trouvé, avec la valeur *false* dans le cas contraire.

Dans l'exemple ci-dessous, on cherche à lire l'enregistrement qui correspond au Grand Prix du Brésil. Pour cela, la clé est initialisée à la valeur 'BRESIL' :

```
ClePrix := 'BRESIL';
TARead(FicPrix, Prix, ClePrix, true);
if Ok
  then {enregistrement trouvé}
  else {enregistrement non trouvé}
```

Dans l'exemple suivant, on cherche à lire le premier GP qui commence par la lettre B :

```
ClePrix := 'B';
TARead(FicPrix, Prix, ClePrix, false);
if Ok
  then {un enregistrement a été trouvé (ce peut être Belgique)}
  else {aucun GP ne commence par la lettre B}
```

2.5 Effacer un enregistrement du fichier (TADelete).

Utiliser la procédure TADelete pour effacer un enregistrement et, par conséquent, la clé correspondante.

```
procedure TADelete(var Fichier : DataSet;
                   var Cle);
```

Fichier est un fichier séquentiel indexé déjà ouvert et **Cle**, de type **EnregCle**, doit avoir été initialisée avec la valeur de la clé qui correspond à l'enregistrement à supprimer. La valeur *true* sera retournée dans **OK** si la clé recherchée a été trouvée et exclue. La valeur *false* sera retournée si la clé n'a pas été trouvée. En réalité, la clé est retirée de l'index, mais l'enregistrement reste physiquement dans le fichier de données avec une valeur non nulle dans le champ d'état (Status) ; ce qui permettra la réutilisation du même espace pour introduire un nouvel enregistrement plus tard.

Dans l'exemple ci-dessous, on cherche à retirer le GP de Belgique du fichier :

```
ClePrix := 'BELGIQUE';
TADelete(FicPrix, ClePrix);
if not Ok
  then {GP de Belgique non trouvé};
```

2.6 Recherche séquentielle dans le fichier (TANext, TAPrev, TAReset).

La recherche séquentielle peut être réalisée par ordre croissant ou décroissant des clés de l'index.

TANext permet de lire un enregistrement dont la clé est la suivante par rapport à une clé déterminée :

```
procedure TANext(var Fichier : DataSet;
                 var Enreg;
                 var Cle);
```

Fichier est un fichier séquentiel indexé déjà ouvert. **Clé** est une variable de type **EnregCle** qui doit être initialisée avant l'appel. **Enreg** est une variable de type **EnregFichier** qui recevra le contenu de l'enregistrement trouvé.

Après l'exécution de **TANext** :

- si **Ok** a la valeur *false*, alors la clé passée comme paramètre est égale ou supérieure à la dernière clé du fichier ; par conséquent, aucune clé suivante n'a été trouvée et le contenu de **Clé** et **Enreg** est indéfini ; si on appelle à nouveau **TANext**, le fichier sera positionné à son début et le premier enregistrement sera récupéré ;
- si **Ok** a la valeur *true*, alors la clé suivante a été trouvée et retournée dans la variable **Cle** ; les données de l'enregistrement associé ont été placées dans la variable **Enreg**.

TAPrev, qui fonctionne de la même manière et avec les mêmes paramètres, permet de récupérer l'enregistrement avec la clé précédente.

TAReset permet de se placer au début du fichier :

```
procedure TAReset(var Fichier : DataSet);
```

Après un appel à **TAReset**, un appel à **TANext** retourne le premier enregistrement du fichier et un appel à **TAPrev** retourne le dernier.

Dans l'exemple ci-dessous, le fichier est placé à son début avant d'être parcouru séquentiellement en ordre croissant des clés :

```
TAReset(Prix);
Ok := true;
while Ok do
  begin
    TANext(FicPrix, Prix, ClePrix);
    if Ok
      then {traiter l'enregistrement (impression par exemple)}
```

```
        else {fin du fichier}
    end;
```

2.7 Fermer un fichier séquentiel indexé (TAClose, TAFlush).

Il est indispensable de fermer un fichier séquentiel indexé qui a été mis à jour, avant que le programme ne se termine. La fermeture automatique, qui est réalisée lorsqu'un programme rend le contrôle au système opérationnel, n'est pas suffisante pour maintenir la structure correcte des fichiers séquentiels indexés. Tout cela parce que, en plus de transférer, de la mémoire vers le disque, les tampons associés au fichier, il est nécessaire de mettre à jour certaines informations enregistrées dans le premier enregistrement de l'index.

Pour fermer un fichier ouvert, utiliser la procédure TAClose :

```
procedure TAClose(var Fichier : DataSet);
```

Il est évident qu'après avoir été fermé, aucune opération ne peut plus être faite sur le fichier ou son index, sans qu'il ait été réouvert par la procédure **TAOpen.**

Une autre procédure intéressante, comparable à l'instruction *flush* disponible dans le Turbo-Pascal pour un fichier classique, est la procédure **TAFlush** :

```
procedure TAFlush(var Fichier : DataSet);
```

Elle permet d'assurer l'intégrité du fichier, même en cas d'abandon du programme ou de coupure d'alimentation. Normalement, les dernières modifications faites dans le fichier restent dans la mémoire : par exemple, une page de clés ne sera pas écrite sur le disque avant qu'il soit nécessaire de libérer l'espace qu'elle occupe pour charger une autre page. Ce qui signifie que, en cas d'arrêt du programme, sans fermeture du fichier, le fichier de données et l'index n'auront pas le même niveau de mise à jour, ce qui entraînera des erreurs lors d'une utilisation ultérieure du fichier. **TAFlush** écrit sur le disque les dernières modifications faites, sans fermer le fichier.

C'est un bon choix d'utiliser **TAFlush** après l'écriture ou la suppression d'un enregistrement. En contrepartie, le traitement sera plus lent, à cause de l'exécution de **TAFlush** qui provoque quelques accès au disque ; mais cela peut être sans importance en cas de traitement interactif.

2.8 Supprimer un fichier séquentiel indexé (TAErase).

TAErase ne fait que supprimer du disque les fichiers physiques (données et index) associés à un fichier séquentiel indexé ouvert.

```
procedure TAErase(var Fichier : DataSet);
```

3 Routines de bas niveau-unité TAccess.

Pour utiliser les routines de bas niveau, il faut citer l'unité taccess dans la directive *uses* du programme. Les tâches réalisées par ces routines sont semblables à celles réalisées par les routines de haut niveau ; généralement, il est nécessaire de faire appel à deux routines de bas niveau (une pour la clé, une autre pour l'enregistrement) là où il suffisait d'une seule routine de haut niveau. Le grand avantage des routines de bas niveau est d'autoriser un accès plus

fin aux données, ce qui permet, entre autres choses, la gestion de plusieurs index pour un même fichier de données.

La plupart des opérations décrites dans les paragraphes suivants le sont pour un seul index. Dans le cas d'un fichier avec plusieurs index, il serait nécessaire de répéter l'opération pour chacun des index ou chacune des clés.

3.1 Créer un fichier séquentiel indexé (MakeFile, MakeIndex).

Les procédures **MakeFile** et **MakeIndex** doivent être utilisées pour créer, respectivement, le fichier de données et le (ou les) index.

```
procedure MakeFile(var FichierDonnees : DataFile;
                       NomFichier     : string;
                       TailleEnreg    : integer);
```

FichierDonnee est une variable déclarée dans le programme appelant, de type *DataFile*. **NomFichier** est le nom du fichier de données et **TailleEnreg** est la taille de l'enregistrement de données.

```
procedure MakeIndex(var FichierIndex : IndexFile;
                        NomIndex     : string;
                        TailleCle    : integer;
                        Dupliquees   : byte);
```

FichierIndex est une variable déclarée dans le programme appelant, de type *IndexFile*. **NomIndex** est le nom physique du fichier d'index. **TailleCle** est la taille de la clé du fichier. Enfin, **Dupliquees** est un *byte* pour indiquer si l'index peut contenir des clés dupliquées (clés identiques qui pointent vers des enregistrements différents) : la valeur 0 indique qu'il ne peut y avoir de clés dupliquées, la valeur 1 indique qu'il peut y en avoir.

On a coutume d'utiliser l'opérateur *sizeof* pour définir la taille de l'enregistrement et de la clé : sizeof(EnregFichier) pour TailleEnreg et pred(sizeof(EnregCle)) pour TailleCle.

Ces routines, en plus de créer les fichiers, les laissent ouverts et prêts à être utilisés. Si le fichier (données ou index) n'a pu être créé, la variable **Ok** revient avec la valeur *false*. Si le fichier est défini pour fonctionner avec plusieurs index, **MakeIndex** doit être appelé pour chacun des index.

Voir exemple au chapitre 3.2.

3.2 Ouvrir un fichier séquentiel indexé (OpenFile, OpenIndex).

Les routines **OpenFile** et **OpenIndex** permettent l'ouverture, respectivement, d'un fichier de données et d'un index déjà existants. Leurs paramètres sont identiques à ceux de **MakeFile** et **MakeIndex** :

```
procedure MakeFile(var FichierDonnees : DataFile;
                       NomFichier     : string;
                       TailleEnreg    : integer);

procedure MakeIndex(var FichierIndex : IndexFile;
                        NomIndex     : string;
                        TailleCle    : integer;
                        Dupliquees   : byte);
```

Si l'ouverture est couronnée de succès, **Ok** revient avec la valeur *true*. Dans le cas contraire (fichier non trouvé par exemple), **Ok** revient avec la valeur *false*. Si le fichier possède plusieurs index, **OpenIndex** doit être appelé pour chacun des index que l'on désire ouvrir. Parfois, il peut ne pas être nécessaire d'ouvrir tous les index.

Dans l'exemple ci-dessous, on essaie d'ouvrir le fichier de données et l'index du fichier des grands prix. Si l'ouverture ne réussit pas, les fichiers seront créés :

```
OpenFile(FicPrix, 'PRIX.DAT', sizeof(EnregPrix));
if Ok
  then OpenIndex(IndPrix, 'PRIX.IND', pred(sizeof(EnregClePrix)), 0);
if not Ok
  then begin
    MakeFile(FicPrix, 'PRIX.DAT', sizeof(EnregPrix));
    if Ok
      then OpenIndex(IndPrix, 'PRIX.IND', pred(sizeof(EnregClePrix)), 0);
  end;
if not Ok
  then {impossible de créer les fichiers}
```

3.3 Rechercher un enregistrement dans le fichier (FindKey, SearchKey, PrevKey, ClearKey).

Il s'agit en réalité de la recherche d'une clé dans le fichier d'indice, puisque, une fois la clé trouvée, il suffira de lire l'enregistrement associé, identifié par son numéro, dans le fichier de données.

Plusieurs procédures permettent de rechercher une clé ; elles utilisent toutes les mêmes paramètres :

- la variable **FichierIndex**, de type **IndexFile**, associée à l'index du fichier,
- le numéro d'enregistrement, **NumEnreg**,
- la variable **Cle** qui contient, avant l'appel, la valeur de la clé à rechercher et, après l'appel, la valeur de la clé trouvée.

Toutes fonctionnent de manière identique : le fichier d'index doit être ouvert, la clé dont la valeur est passée dans **Cle** est recherchée et, en sortie de la procédure, si **Ok** est *true*, **Cle** contient la clé cherchée (qui dépend du type de recherche) et **NumEnreg** contient le numéro d'enregistrement associé. Nous allons maintenant détailler le fonctionnement de chacune de ces procédures.

La procédure **FindKey** recherche dans l'index, une clé dont la valeur soit égale à celle passée comme paramètre :

```
procedure FindKey(var FicIndex : IndexFile;
                  var NumEnreg : longint;
                  var Cle);
```

La procedure **SearchKey** recherche dans l'index, la clé dont la valeur est égale ou supérieure à celle passée comme paramètre :

```
procedure SearchKey(var FicIndex : IndexFile;
                    var NumEnreg : longint;
                    var Cle);
```

Si **Ok** revient avec la valeur *false*, la clé passée comme paramètre était supérieure à la dernière clé de l'index.

La procédure **NextKey** recherche la clé dont la valeur est immédiatement supérieure à celle de la clé passée comme paramètre :

```
procedure NextKey(var FicIndex : IndexFile;
                  var NumEnreg : longint;
                  var Cle);
```

Si **Ok** revient avec la valeur *false*, la clé passée comme paramètre était égale ou supérieure à la dernière clé de l'index.

La procédure **PrevKey** recherche la clé dont la valeur est immédiatement inférieure à celle passée comme paramètre :

```
procedure PrevKey(var FicIndex : IndexFile;
                  var NumEnreg : longint;
                  var Cle);
```

Si **Ok** revient avec la valeur *false*, la clé passée comme paramètre était inférieure à la première clé de l'index.

Enfin, la procédure **ClearKey**, qui utilise un seul paramètre, permet de se placer au début de l'index :

```
procedure ClearKey(var FicIndex : IndexFile);
```

Observation importante : après avoir fait appel à **AddKey** ou **DeleteKey**, il est indispensable d'appeler une des routines **SearchKey**, **FindKey** ou **ClearKey** avant d'appeler **NextKey** ou **PrevKey**, car ces deux dernières routines utilisent un pointeur interne qui est détruit par **AddKey** et **DeleteKey**. Voir, à ce sujet, l'exemple du paragraphe 3.6.

3.4 Ecrire un enregistrement dans le fichier (AddKey, AddRec, PutRec).

La procédure **AddRec** permet d'inclure un nouvel enregistrement dans le fichier de données :

```
procedure AddRec(var FicDonnees : DataFile;
                 var NumEnreg : longint;
                 var Enreg);
```

FicDonnees est un fichier de données de type **DataFile**, déjà ouvert par **OpenFile** ou **MakeFile**. **Enreg** contient les informations à écrire et le champ **Status** qui doit être mis à zéro. Après l'écriture, **NumEnreg** contient le numéro qui a été attribué à l'enregistrement.

La procédure **AddKey** permet d'insérer une nouvelle clé :

```
procedure AddKey(var FicIndex : IndexFile;
                 var NumEnreg : longint;
                 var Cle);
```

Avant l'appel, l'index doit avoir été ouvert, **Cle** doit contenir la clé à insérer et **NumEnreg** doit avoir la valeur du numéro de l'enregistrement associé à la clé. Généralement, NumEnreg est calculé immédiatement avant l'appel, par **AddRec**.

Il faut également signaler que, en cas d'appel à **AddRec** sans appel à **AddKey**, l'enregistrement sera écrit mais ne pourra jamais être récupéré, puisque la clé associée n'aura pas été insérée.

La procédure **PutRec** permet de réécrire, au même emplacement, un enregistrement déjà

existant, après l'avoir modifié par exemple :

```
procedure PutRec(var FicDonnees : DataFile;
                 var NumEnreg : longint;
                 var Enreg);
```

Les paramètres de **PutRec** sont identiques à ceux de **AddRec**, à la différence près que **NumEnreg** doit avoir été renseigné avant l'appel. Généralement, **NumEnreg** peut avoir été mémorisé au moment de la lecture de l'enregistrement ou doit être déterminé à nouveau par la procédure **FindKey** (consulter l'exemple plus bas, dans ce même paragraphe).

Parmi les routines de bas niveau, il n'en existe aucune capable de décider automatiquement si un enregistrement doit remplacer un enregistrement déjà existant, ou s'il s'agit d'un nouvel enregistrement qui doit être inséré. C'est donc au programmeur de déterminer quelle est la situation du prochain enregistrement à écrire. Pour ce faire, l'exemple ci-dessous peut servir de modèle :

```
with Prix do           {remplir l'enregistrement}
  begin
    Nom       := 'Brésil';
    Result[1] := 01;
    ..............
    Result[6] := 20;
    Status    := 0;
  end;
ClePrix := 'BRESIL'; {définir la clé}

FindKey(IndPrix,       {chercher si une clé identique existe}
        NumPrix, ClePrix);

if not Ok
  then begin             {la clé n'existe pas encore}
    AddRec(FicPrix,      {écrit l'enregistrement et retourne son numéro}
           NumPrix, Prix);
    AddKey(IndPrix,      {insère la clé correspondante dans l'index}
           NumPrix, ClePrix);
  end
  else begin             {la clé existe déjà et NumPrix
                          contient maintenant le numéro de l'enregistrement associé}
    PutRec(FicPrix,      {il suffit alors de réécrire l'enregistrement au même emplacement}
           NumPrix, Prix);
  end;
```

Il est évident que, si le fichier a été défini avec plusieurs index, une clé devra être construite pour chacun d'eux, avec les champs correspondants de l'enregistrement.

3.5 Lire un enregistrement dans le fichier (GetRec).

Pour lire un enregistrement de données, il est tout d'abord nécessaire de connaître son numéro. Pour cela, utiliser une des routines de recherche de clé. Il suffit ensuite de faire appel à la procédure **GetRec** :

```
procedure GetRec(var FicDonnees : DataFile;
                     NumEnreg : longint;
                 var  Enreg);
```

FicDonnees est un fichier, de type **DataFile**, qui doit être ouvert. **NumEnreg** est une variable de type *longint* qui doit contenir, avant l'appel, le numéro de l'enregistrement à lire. **Enreg** est une variable du type **EnregFichier** qui recevra les données de l'enregistrement lu.

L'exemple ci-dessous montre comment rechercher puis lire l'enregistrement du fichier des grands prix, associé au Brésil :

```
ClePrix := 'BRESIL';
FindKey(IndPrix, NumPrix, ClePrix);
if Ok
  then GetRec(FicPrix, NumPrix, Prix);
  else {enregistrement non trouvé}
```

Le second exemple montre comment lire l'enregistrement correspondant à la première clé de l'index :

```
ClearKey(IndPrix);
NextKey(IndPrix, NumPrix, ClePrix);
if Ok
  then GetRec(FicPrix, NumPrix, Prix)
  else {fichier vide}
```

3.6 Effacer un enregistrement du fichier (DeleteKey, DeleteRec).

La procédure **DeleteKey** doit être utilisée pour détruire une clé de l'index :

```
procedure DeleteKey(var FicIndex : IndexFile;
                    var NumEnreg : longint;
                    var Cle);
```

La variable **FicIndex**, de type **IndexFile**, représente un index déjà ouvert. **Cle** doit avoir la valeur de la clé à détruire. Si la clé est trouvée, elle sera retirée de l'index, **Ok** reviendra avec la valeur *true*, et le numéro de l'enregistrement sera retourné dans la variable **NumEnreg**.

La procédure **DeleteRec** doit être utilisée pour effacer un enregistrement du fichier des données :

```
procedure DeleteRec(var FicDonnees : DataFile;
                    var NumEnreg : longint);
```

La variable **FicDonnees** représente un fichier, de type **DataFile**, déjà ouvert. Avant l'appel, **NumEnreg** doit contenir le numéro de l'enregistrement à effacer.

Normalement, ces deux procédures sont utilisées ensemble : **DeleteKey** pour effacer la clé et connaître le numéro de l'enregistrement associé et **DeleteRec** pour effacer l'enregistrement associé. L'exemple ci-dessous est plus que l'effacement d'un enregistrement ; il efface tous les enregistrements de grand prix dont la première lettre est un "B" :

```
ClePrix := 'B';
SearchKey(IndPrix,                    {recherche la première clé qui commence par B}
          NumPrix, ClePrix);
while Ok and (ClePrix[1] = 'B') do    {tant que la clé trouvée commence par B}
  begin
    DeleteKey(IndPrix,                {effacer la clé}
              NumPrix, ClePrix);
```

```
    if Ok
      then DeleteRec(FicPrix,        {effacer l'enregistrement}
                     NumPrix);
    SearchKey(IndPrix,               {rechercher la clé suivante}
              NumPrix, ClePrix);
  end;
```

Il faut souligner l'utilisation de **SearchKey**, au lieu de **NextKey**, pour rechercher la clé suivante, puisque l'utilisation antérieure de **DeleteKey** empêche un fonctionnement correct de **NextKey**.

3.7 Fermer un fichier séquentiel indexé (CloseFile, CloseIndex, FlushFile, FlushIndex).

S'ils ont été mis à jour, il est indispensable de fermer le fichier de données et l'index, avant de terminer le programme. La fermeture automatique qui se produit lorsque le programme rend le contrôle au système opérationnel n'est pas suffisante pour maintenir une structure correcte du fichier séquentiel indexé. La raison en est que, en plus de copier, de la mémoire vers le disque, les tampons associés au fichier, il est également nécessaire de mettre à jour certaines informations stockées dans le premier enregistrement de l'index.

Pour fermer un fichier de données, utiliser la procédure **CloseFile** :

```
procedure CloseFile(var FicDonnees : DataFile);
```

Pour fermer un index déjà ouvert, utiliser la procédure CloseIndex :

```
procedure CloseIndex(var FicIndex : IndexFile);
```

Il est évident qu'après la fermeture des fichiers, aucune opération ne peut plus être faite sur ces derniers sans qu'ils aient été réouverts par **OpenFile** et **OpenIndex**.

Deux autres procédures, comparables à l'instruction *Flush* du Turbo-Pascal, sont disponibles pour les fichiers séquentiels indexés ; il s'agit de **FlushFile** et **FlushIndex** :

```
procedure FlushFile(var FicDonnees : DataFile);
procedure FlushIndex(var FicIndex : IndexFile);
```

Elles permettent d'assurer l'intégrité des fichiers, même en cas d'arrêt intempestif du programme ou de panne d'alimentation. Normalement, les dernières modifications faites sur le fichier restent en mémoire : par exemple, une page de clés ne sera écrite sur le disque que lorsqu'il sera nécessaire de récupérer l'espace qu'elle occupe en mémoire, pour charger une autre page. Cela signifie que, en cas d'arrêt brutal du programme, sans fermeture des fichiers, l'index et le fichier de données ne seront pas au même niveau de mise à jour, ce qui entraînerait des erreurs lors d'une prochaine utilisation des fichiers. **FlushIndex** et **FlushFile** écrivent sur le disque les dernières modifications faites, sans fermer les fichiers.

Ce peut être un bon choix d'utiliser **FlushFile** et **FlushIndex** après avoir écrit ou effacé un enregistrement. En contrepartie, le traitement sera plus lent, à cause des accès supplémentaires au disque ; mais ceci peut n'avoir que peu d'importance en cas de traitement interactif.

3.8 Supprimer un fichier séquentiel indexé (EraseFile, EraseIndex).

EraseFile permet d'effacer du disque le fichier de données associé à un fichier séquentiel indexé déjà ouvert :

```
procedure EraseFile(var FicDonnees : DataFile);
```

EraseIndex permet d'effacer du disque l'index associé à un fichier séquentiel indexé déjà ouvert :

```
procedure EraseIndex(var FicIndex : IndexFile);
```

3.9 Autres routines (FileLen, UsedRecs).

La fonction **FileLen** retourne le nombre d'enregistrements contenus dans le fichier de données :

```
function FileLength(var FicDonnees : DataFile) : longint;
```

Le résultat prend en compte le premier enregistrement (utilisé comme enregistrement de contrôle) ainsi que tous les enregistrements déjà effacés et non réutilisés.

La fonction **UsedRecs**, au contraire, retourne le nombre d'enregistrements qui contiennent effectivement des données :

```
procedure UsedRecs(var FicDonnees : DataFile) : longint;
```

Ces deux routines peuvent être utilisées pour déterminer le pourcentage de l'espace occupé par les données utiles :

```
Espace := UsedRecs(FicPrix) / FileLength(FicPrix);
```

Une valeur trop petite indique qu'une réorganisation de la base de données serait la bienvenue.

4 Un exemple d'utilisation du Turbo-Access.

L'exemple présenté dans ce chapitre sera également utilisé comme base des exercices du chapitre 7. Il s'agit de l'administration des résultats du championnat automobile de Formule 1. La raison de ce choix est bien entendu l'existence d'une rivalité Brésil/France, tout à fait d'actualité dans ce sport au moment du cours, et objet de discussions avec nos collègues Brésiliens. La version présentée est simplifiée au maximum, le but étant seulement de donner un exemple d'emploi du Turbo-Access.

Le programme se divise en trois parties :

- définition des fichiers séquentiels indexés ;
- utilisation du Turbo-Access (routines de haut niveau) pour administrer le fichier des pilotes et le fichier des grands prix ;
- utilisation de menus et de grilles d'écran simplifiées pour choisir les options et renseigner les champs de données.

Pour économiser l'espace, l'exemple n'a pas été imprimé ; il est seulement disponible sur la disquette. Dans la version utilisée comme exemple, le programme Formule1 permet de gérer

le fichier des pilotes et les résultats des grands prix. Il sera amélioré dans les exercices du paragraphe 7.

5 Installation et configuration du Turbo-Access.

Pour installer Turbo-Access, on peut se contenter de copier les fichiers TABUILD.EXE, TASIZES.PAS, TACCESS.PAS et TAHIGH.PAS dans le répertoire où est installé le compilateur Turbo-Pascal.

Quelques paramètres utilisés par les routines du Turbo-Access ont besoin d'être initialisés pour chaque fichier séquentiel indexé à gérer, ce qui oblige à recompiler les unités TACCESS (et éventuellement TAHIGH) avant de les utiliser. Si le système à développer gère plusieurs fichiers séquentiels indexés, il est conseillé de configurer ces paramètres de telle façon que la même unité soit utilisée pour tous les fichiers du système.

Le programme TABUILD.EXE permet de calculer ces variables et de recompiler les unités ; toutefois, deux paramètres doivent être renseignés au préalable :

- MaxDataType qui est le type d'enregistrement de données de plus grande taille, parmi les différents fichiers du système ;
- MaxKeyType qui est le type de clé de plus grande taille parmi les différentes clés des fichiers du système.

Avant d'exécuter TABUILD, il est donc nécessaire de préparer un fichier où seront définies ces deux informations. Dans l'exemple présenté plus haut, le fichier FORMULE1.TYP contient la définition des enregistrements et des clés des deux fichiers (Pilotes et Prix) et se termine par la définition de MaxDataType et MaxKeyType qui correspondent au fichier des pilotes :

```
MaxDataType = EnregPilote;
MaxKeyType = EnregClePilote;
```

Si l'installation du Turbo-Access a été faite comme décrit au début de ce paragraphe il suffit, pour exécuter TABUILD, de se placer dans le répertoire du Turbo-Pascal (par exemple CD \TP55) et exécuter TABUILD avec la syntaxe suivante :

```
TABUILD [options] Nom_de_Fichier
```

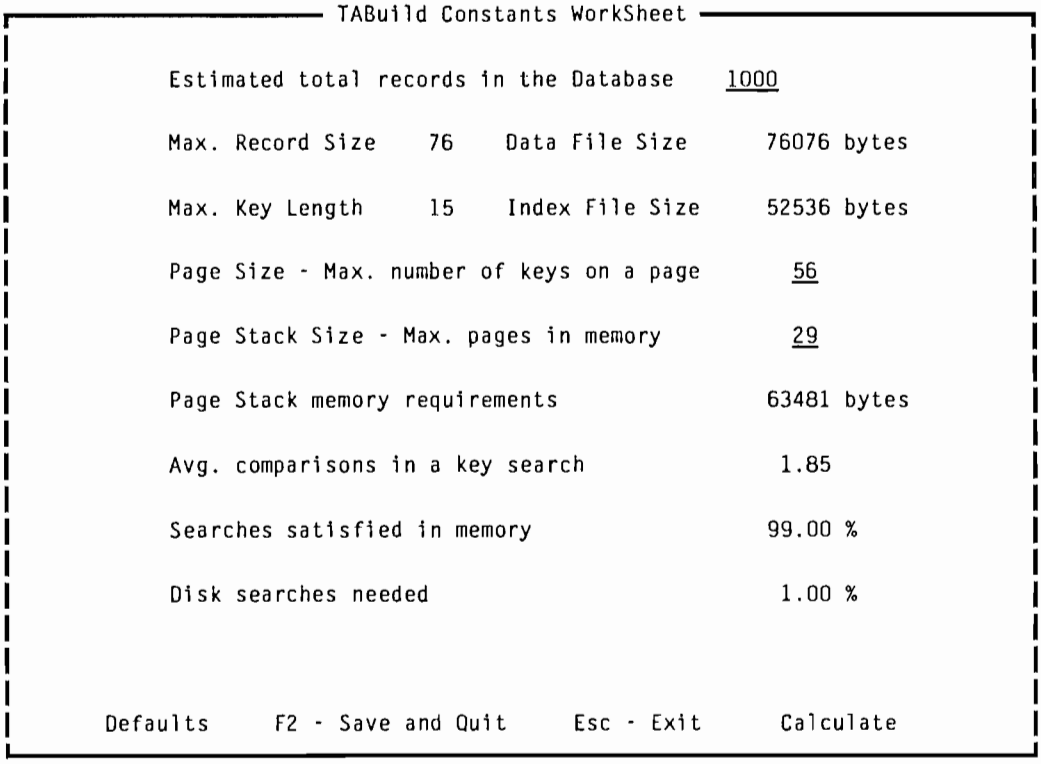
Où Nom_de_Fichier est le nom du fichier où sont déclarés MaxDataType et MaxKeyType. Le nom du fichier peut comprendre le nom du répertoire où il se trouve ; les fichiers TACCESS.TPU et TAHIGH.TPU seront créés dans ce répertoire. Les options sont :

- /H- pour ne compiler que les routines de bas niveau,
- /E- pour générer des messages d'erreur résumés en cas d'erreur lors de l'exécution du programme applicatif,
- /W+ pour ajuster les valeurs des paramètres de configuration (voir exemple plus bas),
- /$xx pour compiler les unités du Turbo-Access avec une ou plusieurs directives de compilation du Turbo-Pascal.

Pour en revenir à l'exemple du chapitre 4, la commande d'utilisation de TABUILD pourrait être :

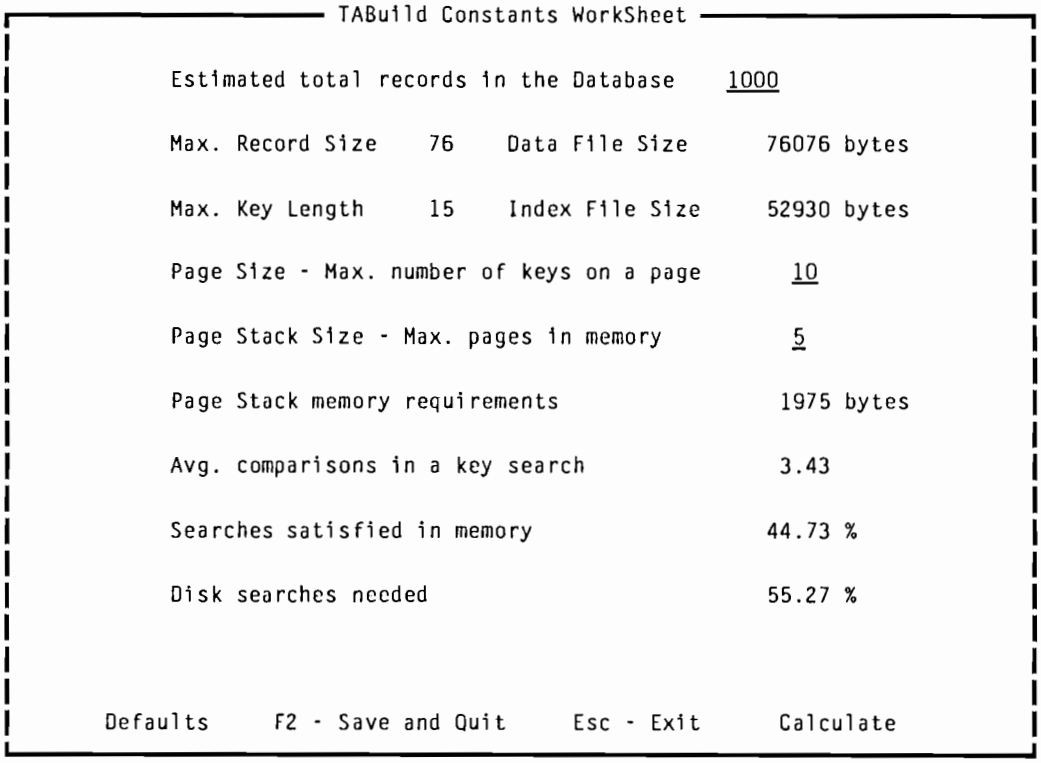
```
TABUILD /W+ \PASAV\FORMULE1.TYP
```

Le programme TABUILD inspecte le fichier FORMULE1.TYP et, puisque l'option /W+ a été utilisée, montre à l'écran les informations suivantes :



Dans cette grille, le programme TABUILD calcule, à partir de la taille maximum de l'enregistrement et de la clé, le nombre de clés par page et le nombre de pages gardées en mémoire en même temps. Pour cela, il se base sur l'utilisation de 64 K octets de mémoire. Avec un nombre arbitraire d'enregistrements (1000), il calcule alors les performances moyennes atteintes lors de la recherche des clés (dans l'exemple ci-dessus, une recherche nécessiterait en moyenne 1.85 comparaisons de clés et 99% d'entre elles seraient faites en mémoire).

Ces informations ne servent pas seulement pour satisfaire la curiosité : trois d'entre elles peuvent être modifiées (elles sont soulignées sur l'exemple) pour adapter les performances du programme qui utilisera les routines du Turbo-Access : si on diminue le nombre de clés par page (**PageSize**) et le nombre de pages en mémoire (**PageStackSize**), l'occupation mémoire sera moindre ; en contrepartie, une recherche prendra plus de temps et nécessitera plus d'accès au disque. La grille présentée ci-dessous a été obtenue après avoir diminué la valeur de PageSize à 10 et celle de PageStackSize à 5 et demandé un nouveau calcul en actionnant la touche C.



Le choix, entre minimiser l'espace occupé et minimiser le temps de recherche et les accès au disque, dépend beaucoup du système à développer. Si le système ne nécessite pas une taille mémoire importante, il sera avantageux de réserver le maximum de mémoire pour Turbo-Access. Dans le cas contraire, il sera plus judicieux que Turbo-Access occupe le moins d'espace possible, afin de laisser la mémoire ainsi économisée à la disposition du programme applicatif.

Le nombre total d'enregistrements peut également être modifié mais il est sans influence sur les performances futures du système ; seul l'espace occupé sur le disque sera recalculé. Le fichier physique pourra contenir plus d'enregistrements, ou moins.

Observation : quand le système administre plusieurs fichiers séquentiels indexés, il est conseillé de créer le fichier des types à utiliser d'une autre manière, pour ne pas regrouper dans le même fichier la définition de tous les fichiers. En appliquant cette observation à notre exemple, on pourrait utiliser un fichier, nommé par exemple GLOBAL.TYP, contenant seulement la définition de MaxDataType et MaxKeyType :

```
type
  MaxDataType = array[1..76] of bytes; {76 = taille de EnregPilote}
  MaxKeySize  = string[15];            {15 = taille de EnregClePilote}
```

6 Erreurs d'exécution-Réorganisation des fichiers.

6.1 Quelques conseils pour éviter les problèmes.

Déterminer avec soin les valeurs de MaxDataType et MaxKeyType, en tenant compte de tous les fichiers séquentiels indexés du système à développer. La modification *a posteriori*, sans être impossible, est assez laborieuse (voir chapitre 6.3). Après avoir configuré les unités par TABUILD, noter les valeurs créées dans le fichier TACCESS.DEF ou conserver ce fichier en lieu sûr.

On a pu noter que, dans toutes les routines, les paramètres Enreg et CleEnreg sont déclarés comme des variables sans type. Par conséquent, le compilateur ne peut contrôler le type de la variable passée comme paramètre, ce qui, en phase d'exécution peut nuire à l'intégrité du fichier ou de l'index. Vérifiez toujours que la clé n'est pas passée en place de l'enregistrement ou inversement.

Vérifiez également que les procédures de fermeture des fichiers sont bien exécutées en fin de mise à jour.

En cas d'erreur fatale au cours de l'exécution d'une routine de bas niveau ou de haut niveau, un message apparaît à l'écran et un fichier, nommé TACCESS.ERR, peut être consulté pour obtenir de plus amples informations. Le programmeur peut également implémenter sa propre routine d'erreur, par exemple pour assurer la fermeture correcte des fichiers, même en cas d'erreur : il suffit de la compiler avec la directive $F+ et de placer son adresse dans le pointeur TAErrorProc déclaré dans l'unité TACCESS. C'est ce qui a été fait dans le programme Formule1, présenté au chapitre 4, avec la procédure SOSEnCasDErreur. Ceci peut éviter la corruption des fichiers, non seulement en cas d'erreur du Turbo-Access, mais aussi en cas d'erreur quelconque.

6.2 Réorganisation des fichiers.

Une réorganisation des fichiers peut être conseillée quand le fichier de données contient beaucoup d'enregistrements effacés et non réutilisés.

La logique d'un programme de réorganisation est simple. Il suffit de parcourir séquentiellement l'index, récupérer dans le fichier de données l'enregistrement associé et le réécrire dans un nouveau fichier.

Un des exercices du chapitre 7 traite de l'implémentation d'un programme de réorganisation.

6.3 Réorganisation en cas de problème.

Elle peut devenir nécessaire pour récupérer les données, dans le cas où l'index ou le fichier de données ont été corrompus. On rencontre également des problèmes lors de l'utilisation d'une nouvelle version de TACCESS ou TAHIGH, soit parce-qu'il a été nécessaire de modifier les valeurs de MaxDataType ou MaxKeyType, soit parce-que TACCESS a été recompilé accidentellement avec des paramètres différents.

Dans ce dernier cas, un des messages **1003 Data file created with different record size** ou **1004 Index File created with different key or page size** sera produit.

Le programme présenté ci-dessous propose une solution pour ce type de situation : lire le fichier de données comme un fichier séquentiel et, pour chaque enregistrement utile, construire la clé et écrire l'enregistrement et la clé avec la nouvelle version des routines. Les enregistrements inutiles sont le premier, utilisé à des fins de contrôle, et tous ceux dont le champ Status est non nul (enregistrements effacés). Une réorganisation de ce type n'est possible que si les champs qui composent la clé sont répétés dans l'enregistrement de données, ce qui est toujours conseillé.

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{   exemple de réorganisation de sequ.indexé dont l'index est inexploitable  }
{                     (disponible dans le fichier reorgan.pas)               }
{----------------------------------------------------------------------------}
program reorgan; {réorganise un fichier séquentiel indexé,
                  par exemple, après avoir recompilé taccess}
uses
  crt,dos,taccess;

{définitions du fichier à traiter-doivent être adaptées pour chaque cas}
type
  TypeSortie = record
                 Status  : LongInt;        {status de l'enregistrement}
                 Nom,
                 Prenom  : string[15];
                 Numero  : string[2];
                 PaysOri : string[15];
                 Auto    : string[20];
               end;
  TypeCleSortie = string[15];
const
  NomFicSortie = 'PILOTES.OAT';
  NomIndSortie = 'PILOTES.IND';
  NomFicEntree = 'PILOTES.XXX';

{declarations du programme}
const
  SizeSortie = sizeof(TypeSortie);
  SizeCleSortie = pred(sizeof(TypeCleSortie));
  TotEnreg : longint = 0;

var
  FicEntree : file of TypeSortie;
  EnrSortie : TypeSortie;
  CleSortie : TypeCleSortie;
  FicSortie : DataFile;
  IndSortie : IndexFile;
  NumEnreg  : longint;

function CleCalculee : string;
{routine qui calcule la clé de l'enregistrement}
{doit être adaptée pour chaque cas}
var
  Aux4 : string[4];
begin
  with EnrSortie do
    begin
      CleCalculee := copy(Nom + '               ', 1, SizeCleSortie);
    end;
end;

procedure OuvrirFichierEntree;
{ouverture de l'ancien fichier de données en le considérant comme un
 fichier séquentiel}
begin
  assign(FicEntree, NomFicEntree);
  reset(FicEntree);
end;
```

```
procedure CreerFichierSortie;
{création du nouveau fichier séquentiel indexé}
begin
  MakeFile(FicSortie, NomFicSortie, SizeSortie);
  if Ok then
    MakeIndex(IndSortie, NomIndSortie, SizeCleSortie, 0);
  CloseFile(FicSortie);
  CloseIndex(IndSortie);
  OpenFile(FicSortie, NomFicSortie, SizeSortie);
  OpenIndex(IndSortie, NomIndSortie, SizeCleSortie, 0);
end;

procedure FermerFichierSortie;
begin
  closefile(FicSortie);
  closeindex(IndSortie);
end;

begin
  OuvrirFichierEntree;
  CreerFichierSortie;
  seek(FicEntree, 1); {pour ignorer le premier enregistrement}
  clrscr;
  while not eof(FicEntree) do
    begin
      read(FicEntree, EnrSortie);
      if EnrSortie.Status = 0
        then begin         {enregistrement avec données}
          addrec(FicSortie,NumEnreg,EnrSortie);
          CleSortie := CleCalculee;
          inc(TotEnreg);
          addkey(IndSortie, NumEnreg, CleSortie);
          gotoxy(10,10);write('récupération du n° ',TotEnreg:10,' ',CleSortie);
        end;
    end;
  FermerFichierSortie;
  clrscr;
  writeln('Récupération terminée - ', TotEnreg, ' enregistrements récupérés');
end.
```

Ce programme peut facilement être adapté pour le cas d'un autre fichier : il suffit de substituer la définition de l'enregistrement et de la clé, la routine de calcul de la clé et le nom des fichiers. Si le nouveau fichier indexé est créé avec le même nom, l'ancien fichier de données doit avoir été renommé avant d'exécuter le programme.

7 Exercices.

a) Modifier le programme Formule1 présenté comme exemple pour ajouter un second index au fichier des Pilotes et mettre au point la troisième fonction (Afficher le classement). La clé associée à ce fichier est le numéro de la voiture. Ceci conduit à modifier la gestion du fichier des Pilotes et la récupération du nom du pilote dans la mise à jour et l'émission des résultats.

b) Ecrire un programme pour réorganiser le fichiers des Grands Prix, en utilisant la récupération séquentielle des clés.

Chapitre IV

Tri interne

Le **Turbo-Sort** fait également partie du **Turbo-Toolbox**. Il propose des routines de tri d'informations d'un type quelconque. Le tri peut se faire en ordre croissant ou décroissant, basé sur une ou plusieurs clés. L'algorithme utilisé est le *Quick-Sort*, célèbre pour son efficacité et sa rapidité dans la plupart des cas. Le tri utilise toute la mémoire disponible avant d'utiliser le disque, dans le cas où la mémoire est insuffisante ; ceci se fait automatiquement, sans que le programmeur ait à s'en préoccuper.

1 Description et utilisation.

Deux unités sont disponibles : **SORT.TPU** et **LSORT.TPU**. La première permet de trier jusqu'à 32767 éléments et sera suffisante dans la plupart des cas ; la seconde permet de trier jusqu'à plus de deux milliards d'éléments mais est plus lente et ne doit être utilisée que dans les cas spécifiques où il est réellement prévu d'avoir à trier plus de 32767 éléments. Toutes les informations et les exemples donnés ci-dessous correspondent à l'utilisation de **SORT.TPU**, toutefois l'utilisation de **LSORT.TPU** est tout à fait semblable, le nom des routines étant simplement précédé de la lettre L (par exemple, à la fonction **TurboSort**, correspond la fonction **LTurboSort**).

1.1 Trois étapes à respecter.

Pareillement au *sort-interne* disponible en langage Cobol, le travail réalisé par le **Turbo-Sort** est divisé en trois phases distinctes :

- introduction des données à trier, ce qui correspond à l'*input procedure* du Cobol ;
- tri des données ;
- récupération des données triées, ce qui correspond à l'*output procedure* du Cobol.

Le programmeur doit implémenter les routines d'introduction et de récupération des données et celle qui compare un élément avec un autre, qui sera utilisée par le **Turbo-Sort**. Les règles pour implémenter ces routines sont décrites ci-dessous.

1.2 Routine d'introduction des données.

Il s'agit d'une procédure sans paramètres, compilée avec la directive $F+ car elle fera l'objet d'un appel long par la procédure **TurboSort**. Le nom de la procédure est sans importance et l'obtention des données à trier peut être réalisée de manière quelconque (lecture d'un fichier, calcul en mémoire). Mais chaque élément à trier doit être communiqué au **Turbo-Sort** par la procédure **SortRelease**.

Dans l'exemple ci-dessous, la procédure d'entrée des données lit tous les enregistrements du fichier des pilotes de l'exemple du chapitre III :

```
{$F+}
procedure LireDonnees;
{$F-}
var
  P : EnrPilote;
  C : EnrClePilote;
begin
  C := '';
  TANext(FicPilote, P, C);
  while Ok do
    begin
      SortRelease(P);
      TANext(FicPilote, P, C);
    end;
end;
```

1.3 La fonction de comparaison des éléments.

Elle réalise la comparaison de deux éléments en suivant les critères de tri imposés par le programmeur. Elle sera appelée souvent par la routine **TurboSort** ; par conséquent, elle doit obligatoirement être compilée avec la directive $F+ et être la plus efficace possible. La fonction doit utiliser deux paramètres, passés par adresse, du type des données à trier et son résultat doit être de type *boolean*. Dans le corps de la fonction, la comparaison doit calculer la valeur de la fonction qui doit être *true* quand le premier paramètre est inférieur au second.

L'exemple ci-dessous montre la routine de classement des pilotes par leur prénom, alors qu'initialement ils sont classés par nom de famille :

```
{$F+}
function PlusPetitPilote(var P, Q : EnrPilote);
{$F-}
begin
  PlusPetitPilote := P.Prenom < Q.Prenom;
end
```

1.4 Procédure de récupération des données.

Son rôle est de récupérer les informations après le tri. Cette fois, c'est la partie de récupération qui doit obligatoirement utiliser une routine de **Turbo-Sort** : la procédure **SortReturn**. Une fois récupérée, la donnée peut faire l'objet d'un traitement quelconque : visualisation à l'écran ou sur l'imprimante, écriture dans un fichier, etc...

La fonction **SortEOS** sert à connaître le moment où toutes les données ont été récupérées.

Dans l'exemple ci-dessous, on récupère les pilotes après tri et on montre le nom et le prénom de chacun d'eux à l'écran.

```
{$F+}
procedure RecupererDonnees;
{$F-}
var
  P : EnrPilote;
begin
  while not SortEOS do
    begin
      SortReturn(P);
      with P do
        writeln(Prenom, ' ', Nom);
    end;
end;
```

1.5 L'utilisation de la routine TurboSort.

La routine **TurboSort** est une fonction de type *integer* dont la valeur contient un code de retour après l'exécution du tri. La valeur 0 signifie que tout s'est bien passé, une valeur différente de 0 correspond à une erreur d'exécution. Les paramètres à passer sont :

- la taille des éléments à trier (il vaut mieux la calculer avec l'opérateur *sizeof*) ;
- l'adresse de la routine d'introduction des données ;
- l'adresse de la routine de comparaison ;
- l'adresse de la routine de récupération des données.

La fonction **TurboSort** peut renvoyer les valeurs suivantes :

- 0 quand tout s'est bien passé ;
- 3 en cas de mémoire insuffisante (taille de la mémoire disponible inférieure à trois fois la taille de l'élément à trier) ;
- 8 si la taille de l'élément à classer est inférieure à 2 ou supérieure à 32767 ;
- 9 si le nombre d'éléments à classer est supérieur à 32767 lors de l'utilisation de **SORT.TPU** ;
- 10 en cas d'erreur d'écriture sur disque ;
- 11 en cas d'erreur de lecture sur disque ;
- 12 en cas d'erreur de création des fichiers de travail (disque plein).

Pour conclure, dans le cas de notre exemple, la syntaxe serait :

```
if TurboSort(sizeof(EnrPilote), @LireDonnees, @PlusPetitPilote,
             @RecupererDonnees) <> 0
  then {une erreur d'est produite}
```

Le fichier TRIPILOT.PAS contient le programme complet d'où ont été extraits les exemples utilisés ci-dessus.

2 Un exemple complet.

L'exemple présenté dans ce paragraphe implémente une commande semblable au *dir* du DOS. Le programme **Repert** montre à l'écran la liste des fichiers d'un répertoire, en ordre alphabétique de nom et d'extension. Les sous-répertoires sont éliminés de la liste et, quand l'écran est plein, l'opérateur peut le consulter avant d'appuyer sur une touche quelconque

pour passer à la suite de la liste. Le tri des noms de fichiers est réalisé avec l'aide du **TurboSort.**

De plus, il est possible d'indiquer sur la ligne de commande plusieurs paramètres de sélection des fichiers. Par exemple :

```
repert *.pas *.tpu
```

permet d'obtenir la liste triée des fichiers d'extension *.PAS* et d'extension *.TPU*. Si aucun paramètre n'est précisé, tous les noms de fichier seront affichés.

Signalons également, dans cet exemple, l'emploi des types et des routines pré-déclarées de l'unité *Dos* du Turbo-Pascal.

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{                          exemple de sort interne                           }
{                   (disponible dans le fichier repert.pas)                  }
{----------------------------------------------------------------------------}
{affichage du contenu du répertoire en ordre alphabétique}
uses crt, dos, sort;

var
  Result : integer;
  NomFich : string[12];

{$F+}
procedure ObtenirFichiers;
var
  Fichier : SearchRec;
  i,
  j : byte;
begin
  {si aucun paramètre sur la ligne de commande, on
   assume *.*, c'est à dire tous les fichiers     }
  j := ParamCount;
  if j = 0
    then begin
      j := 1;
      NomFich := '*.*';
    end;

  for i := 1 to j do  {obtenir le nom des fichiers dont le nom correspond au paramètre}
    begin
      if ParamCount > 0
        then NomFich := ParamStr(i);
      FindFirst(NomFich, $3f, Fichier);   {recherche le premier fichier du répertoire
                                           qui correspond au paramètre}
      while DosError = 0 do
        begin
          if Fichier.Attr <> $10          {si le fichier n'est pas un sous-répertoire}
            then SortRelease(Fichier);    {communiquer Fichier au Turbo-Sort}
          FindNext(Fichier);              {rechercher le prochain fichier; s'il n'y en a plus,
                                           DosError prend la valeur 18}
        end;
    end;
end;
```

```
function PlusPetitNom(var a, b : SearchRec) : boolean;
begin
  PlusPetitNom := a.Name < b.Name;          {compare les noms de fichiers(nom+extension)}
end;

procedure AfficherFichiers;
var
  Fichier : SearchRec;
  Date : DateTime;
  i : byte;
  l : word;
  s : string;
  c : char;

  function Completer(i : integer) : string;
  var
    s : string[3];
  begin
    str(100+i:3, s);
    Completer := copy(s, 2, 2);
  end;

begin
  l := 0;
  writeln;
  while not SortEOS do
    begin
      SortReturn(Fichier);                    {récupère chaque fichier classé}
      UnPackTime(Fichier.Time, Date);     {transforme le temps en date/heure}
      with Fichier, Date do
        begin                               {affiche à l'écran les caractéristiques du fichier}
          i := pos('.', Name);
          s := Name;
          if i <> 0
            then s := copy(copy(Name, 1, pred(i))+'         ', 1, 9)+copy(Name, succ(i), 3);
          while length(s) < 12 do s := s + ' ';
          writeln(s, Size:10, '  ',
                  Day:2, '/', Completer(Month):2, '/', Year:4, '  ',
                  Completer(Hour):2, ':', Completer(Min):2, ':', Completer(Sec):2);
          inc(l);
          if l mod 23 = 0
            then begin                      {temporisation quand l'écran est plein}
              write('.....................................suite');
              c := readkey;
              while keypressed do c := readkey;
              writeln;
            end;
        end;
    end;                              {répète jusqu'au dernier fichier classé}
end;
{$F-}

begin
  Result := TurboSort(sizeof(SearchRec), @ObtenirFichiers, @PlusPetitNom,
                      @AfficherFichiers);
end.
```

3 Exercices.

a) Modifier le programme TRIPILOT pour obtenir la liste des pilotes en ordre décroissant des prénoms.

b) Compléter le programme REPERT pour autoriser les options /T, /D et /E, sur la ligne de commande, afin d'obtenir la liste en ordre croissant de taille de fichier, de date de mise à jour ou d'extension de fichier, respectivement. Pour ne pas trop compliquer le traitement des options, on admet que, en cas d'options multiples, seul la dernière sera prise en compte. Par exemple, **Repert *.pas *.tpu /T** devra produire la liste des fichiers par ordre croissant de taille de fichier.

Chapitre V

Fonctions du DOS. Interruptions. Instructions et directives "inline". Interface avec d'autres langages.

Cette partie du cours traite plus particulièrement des interactions entre le langage Turbo-Pascal et l'environnement du système opérationnel du PC. Par conséquent, il est bon que le lecteur ait des connaissances minimales de langage assembleur et du fonctionnement interne du DOS et du BIOS, pour profiter pleinement des informations données dans ce chapitre.

1 Exécution d'interruptions ou de fonctions du DOS.

Rappelons que l'utilisation du micro-ordinateur PC, XT ou AT est basée sur le BIOS (Basic Input Output System), ensemble de fonctions implantées dans la partie ROM de la mémoire et qui sert d'interface entre le processeur et l'utilisateur. Chacune de ces fonctions est accessible au processeur 80x86 par l'intermédiaire d'un procédé appelé *interruption*. Chaque interruption, identifiée par un numéro (de 0 à 255), met à disposition de l'utilisateur une ou plusieurs fonctions dont les paramètres (d'entrée) ou les résultats (en sortie) sont communiqués, entre l'appelant et l'interruption, par l'intermédiaire des registres du processeur.

1.1 Comment exécuter une interruption ?

Le type *registers*, déclaré dans l'unité **DOS**, permet de manipuler les registres du processeur :

```
type
  Registers = record
                case integer of
                0 : (ax, bx, cx, dx, bp, si, di, ds, es, flags : word);
                1 : (al, ah, bl, bh, cl, ch, dl, dh : byte);
                end;
              end;
```

Il s'agit d'une structure avec variante, qui autorise la manipulation des registres aussi bien en format 16 bits (ax, bx, cx, etc...) qu'en format 8 bits (al, ah, bl, bh, cl, ch, dl et dh).

La procédure *Intr*, déclarée dans l'unité DOS permet d'exécuter une interruption. En voici l'en-tête :

```
procedure Intr(NumInt : byte; var Regs : registers);
```

où NumInt est le numéro de l'interruption à exécuter et Regs est une variable utilisée pour communiquer des valeurs aux registres ou en récupérer.

Dans un programme, on doit respecter la séquence suivante :

- donner les valeurs correctes aux registres de la variable Regs : le registre AH doit contenir le numéro de la fonction à réaliser et, éventuellement, d'autres registres doivent contenir les paramètres ;
- faire appel à l'interruption, en exécutant la fonction Intr ;
- récupérer éventuellement les résultats retournés par l'interruption dans les registres de la variable Regs.

L'interruption 21h qui correspond aux nombreuses fonctions du système opérationnel DOS, peut être appelée directement par la procédure *MsDos* dont l'en-tête est le suivant :

```
procedure MsDos(var Regs : Registers);
```

En réalité, MsDos(Regs); est équivalent à Intr($21, Regs);.

1.2 Exemples d'appel d'interruption.

Le premier exemple est assez simple : il s'agit d'exécuter une copie de l'écran sur l'imprimante, de l'intérieur d'un programme, sans utiliser la touche correspondante du clavier (touche PrtScr ou équivalente). L'interruption 05h réalise cette tâche, sans aucun paramètre supplémentaire. Par conséquent, il suffit de déclarer la variable Regs et d'exécuter l'appel de la procédure :

```
uses dos;
var
  Regs : registers;
.................
intr(5, Regs);
```

Le second exemple est plus complet. Il s'agit d'obtenir la date et l'heure courantes, connues par le système opérationnel, et les transformer en chaîne de caractères. La fonction DataHeure, imprimée et commentée ci-dessous fait appel à l'interruption DOS (21h) pour récupérer la date et l'heure courantes :

```
function DateHeure : string;

var
  Aux3  : string[3];
  Aux4  : string[4];
  Aux18 : string[18];
  Regs  : Registers; (*variable qui contient les registres*)

begin

(*obtention de la date*)
  Regs.AH := $2A; (*AH (la partie haute du registre AX)
                       doit contenir le numéro de la fonction
                       que retourne la date courante (fontion 2Ah)*)

  Intr($21, Regs); (*appelle l'interruption DOS (21h)*)

  with Regs do                          (*après exécution de l'interruption:*)
    begin
      Str((DL+100):3,Aux3);             (* DL contient le jour              *)
      Aux18 := copy(Aux3,2,2)+'/';
      Str((DH+100):3,Aux3);             (* DH contient le mois              *)
```

```
      Aux18 := Aux18+copy(Aux3,2,2)+'/';
      Str(CX:4,Aux4);                    (* CX contient l'année              *)
      Aux18 := Aux18+Aux4+' à ';
    end;

(*obtention de l'heure*)
  Regs.AX := $2C00;(*AH (la partie haute du registre AX)
                      doit contenir le numéro de la fonction
                      qui retourne l'heure courante (fonction 2Ch)*)

  Intr($21, Regs); (*appelle l'interruption DOS (21h)*)

  with Regs do                           (*après exécution de l'interruption*)
    begin
      str(CH+100:3,Aux3);                (* CH contient l'heure             *)
      Aux18 := Aux18+copy(Aux3,2,2)+':';
      str(CL+100:3,Aux3);                (* CL contient les minutes         *)
      Aux18 := Aux18+copy(Aux3,2,2);     (* DH contient les secondes, et DL contient*)
    end;                                 (* les centièmes de secondes qui ne sont  *)
                                         (* pas pris en compte dans l'exemple      *)
  DateHeure := Aux18;                    (*DateHeure retourne une chaîne de caractères
                                           de la forme jj/mm/aaaa à hh:mm*)
end;
```

Rappelons que l'appel Intr($21, Regs) pourrait être remplacé par l'appel direct aux fonctions du DOS, c'est à dire MsDos(Regs). Dans les versions les plus récentes de Turbo-Pascal, l'unité **DOS** offre déjà des fonctions pour récupérer et initialiser la date et l'heure du système (voir annexe I). Par conséquent, les routines ci-dessus ont seulement valeur d'exemple.

Le troisième exemple vaut uniquement pour la version 3.3 du DOS ou les versions plus récentes. Elle sert à contourner une limite qui perturbe fréquemment le développement de grands systèmes, à savoir le nombre maximum de fichiers qu'il est possible de garder ouverts au même instant. Par défaut, la limite maximum de fichiers ouverts que peut supporter le DOS est de 20, y compris les périphériques standards (clavier, console).

Cette limite peut facilement être réduite par la directive FILES=n du fichier de configuration CONFIG.SYS (voir documentation du DOS). Mais, pour l'augmenter, il ne suffit pas de spécifier une valeur plus grande dans le fichier CONFIG.SYS, il est également nécessaire d'utiliser la fonction 67h du DOS. La fonction ci-dessous peut être utilisée pour cela :

```
function MaxFichier(n : byte) : boolean;
var
  Regs : registers;
  Version : word;

begin
  Version := DosVersion;
  if (lo(Version) > 3) or
     ( (lo(Version) = 3) and (hi(Version) >= 30) )
    then begin
      Regs.bx := n;
      Regs.ah := $67;
      MsDos(Regs);
      MaxFichier := (Regs.Flags and FCarry) = 0;
    end
```

```
    else MaxFichier := false;
end;
```

Cette fonction mérite quelques commentaires :

- notons l'utilisation de la fonction **DosVersion** de l'unité DOS pour déterminer quelle est la version du DOS en usage sur le micro-ordinateur ;

- si la version est inférieure à 3.3, la fonction 67h du DOS n'existe pas, par conséquent il est inutile de tenter de l'exécuter et MaxFichier retourne la valeur *false* pour indiquer que le nombre maximum de fichiers n'a pas été modifié ;

- si la version est supérieure ou égale à 3.3, les registres sont remplis avec les informations correctes : AH avec le numéro de la fonction 67h et BX avec le nouveau nombre de fichiers (n) passé comme paramètre ;

- après exécution de la fonction, la valeur des flags est testée pour vérifier si tout s'est bien passé : si le bit "Carry" des flags est à la valeur 1, cela signifie que l'opération n'a pas pu être réalisée ;

- l'utilisation de la fonction peut se faire comme dans l'exemple ci-dessous :

```
if MaxFichier(25)
  then {poursuivre le traitement}
else begin
  {message}
  halt;
end;
```

2 Instructions et directives "inline".

Les instructions *inline* du Turbo-Pascal facilitent l'utilisation de petites routines implémentées en langage assembleur. Elles permettent d'introduire du code en assembleur, directement dans le code source d'un programme ou d'une unité, au lieu d'utiliser une routine indépendante en langage assembleur qui nécessiterait d'être compilée sous forme d'un fichier *.OBJ* (voir chapitre 3).

L'utilisation des instructions *inline* doit se limiter à des routines très courtes, car elle devient trop complexe dès que la fonction requiert plusieurs variables et dépasse une dizaine d'instructions en assembleur. Elle requiert également des connaissances en assembleur.

2.1 Syntaxe des instructions inline.

La syntaxe d'une instruction *inline* est illustrée par l'exemple ci-dessous :

```
{---------------------------cours de Pascal-Avancé--------------------------}
{                                                                            }
{                          exemple d'instruction inline                      }
{                     (disponible dans le fichier inlin1.pas)                }
{----------------------------------------------------------------------------}

program inlin1;
uses crt;
var
  Rien : array[1..5000] of word;
  p : pointer;
  c : char;
  FilWrd : word;
  FilByt : array[1..2] of char absolute FilWrd;
```

```
procedure RemplirZone(var Zone; Compteur : word; Mot : word);
begin
  inline(
         $c4/$be/Zone/      {les di, Zone[bp]    }
         $8b/$8e/Compteur/  {mov cx, Compteur[bp]}
         $8b/$86/Mot/       {mov ax, Mot[bp]}
         $FC/               {cld                 }
         $f3/$ab);          {rep stosw           }
end;

begin

  {remplir les premières lignes de l'écran avec des
   caractères 'a' clignotants                      }
  p := Ptr($b800, 0);
  FilByt[1] := 'a';
  FilByt[2] := char(Red+Blink);
  RemplirZone(p^, 800, FilWrd);

  {remplir la variable Rien par des zéros}
  RemplirZone(Rien, sizeof(Rien) div 2, 0);

  {attendre qu'une touche soit actionnée}
  gotoxy(1,25);
  c := readkey;
end.
```

Chaque élément de l'instruction est séparé du suivant par le symbole "/" et représente une constante ou une variable sur un octet ou un mot (deux octets) de code exécutable.

La taille de l'élément peut être déterminée automatiquement. Dans le cas d'une constante, c'est évident : par exemple $40 va occuper un octet et $123f occupera deux octets. Dans le cas d'une variable, la valeur est calculée en ajoutant, à la valeur du déplacement de la variable à l'intérieur de son segment, la valeur d'une constante éventuelle qui suivrait le nom de la variable : par exemple Zone+1 serait remplacé par la valeur de l'adresse de Zone augmentée de 1.

La taille peut également être imposée, en utilisant les opérateurs < (pour forcer à un octet) ou > (pour forcer à un mot). Par exemple, />$20/ va générer deux octets ($20 et $00).

Le segment à utiliser comme base varie également en fonction de la variable utilisée, ce qui a une influence sur la codification de l'instruction. Il est indispensable de se souvenir que :

- pour une variable globale, ou une variable initialisée (constante avec type), on utilise le segment de données, c'est à dire DS ;
- pour une variable locale, déclarée dans une routine ou une variable passée comme paramètre, on utilise le segment de pile, c'est à dire BP, qui contient toujours l'adresse du sommet de la pile au moment d'entrer dans la routine.

D'autre part, tous les registres peuvent être modifiés à l'intérieur d'une instruction *inline*, à l'exception de BP, SP, SS et DS.

Avec ces éléments, il devient plus facile d'interpréter une instruction *inline*, principalement lorsque le code assembleur équivalent est disponible en commentaire sur la même ligne. Toutefois, le problème auquel le programmeur doit normalement faire face n'est pas d'interpréter une instruction *inline* déjà existante, mais bien de programmer sa propre

instruction *inline*. C'est pourquoi nous allons proposer dans le prochain paragraphe une méthodologie pour le faire, en se basant sur l'exemple précédent.

2.2 Méthode pour programmer une instruction inline.

Première étape : définir, en langage assembleur, ce que doit faire l'instruction. Il n'est pas nécessaire pour cela d'utiliser un compilateur : il suffit de mettre sur le papier, ce que serait la séquence d'instructions en assembleur (souvenez-vous qu'il s'agit de peu d'instructions). Pour l'exemple ci-dessus, ce serait :

```
les di, Zone[bp]      ;pour charger dans ES:DI, l'adresse de la zone à remplir
mov cx, Compteur[bp] ;pour placer dans CX la taille de la zone (en mots de 2 octets)
mov ax, Mot[bp]       ;pour placer en AX le mot qui va replir la zone
cld                   ;pour définir l'incrément de DI comme étant positif
rep stosw             ;pour répéter CX fois la copie de AX à l'adresse
                      ; pointée par ES:DI et incrémenter DI de deux.
```

Remarquons l'usage de [bp] comme segment de base des variables, puisqu'elles sont toutes passées comme paramètres.

Deuxième étape : utiliser l'utilitaire Debug du DOS (consulter le manuel du DOS) pour traduire cette séquence d'instructions en langage-machine. Pour cela, entrer dans Debug et utiliser la commande -a qui permet de saisir du code assembleur. Il est évident que le nom des variables ne peut pas être utilisé directement, on le remplacera donc par une valeur de déplacement sans signification, par exemple $1234, comme dans l'exemple ci-dessous :

```
C:>debug
-a
3F67:0100 les di, 1234[bp]
3F67:0104 mov cx, 1234[bp]
3F67:0108 mov ax, 1234[bp]
3F67:010C cld
3F67:010D rep stosw
3F67:010F
```

Après cela, utiliser la commande -u pour montrer à l'écran (ou mieux, avec copie sur l'imprimante) le code exécutable depuis l'adresse de début (toujours 100) jusqu'à l'adresse de fin des instructions saisies (dans notre cas : 10E) :

```
-u 100 10E
3F67:0100 C4BE3412      LES     DI,[BP+1234]
3F67:0104 8B8E3412      MOV     CX,[BP+1234]
3F67:0108 8B863412      MOV     AX,[BP+1234]
3F67:010C FC            CLD
3F67:010D F3            REPZ
3F67:010E AB            STOSW
```

Troisième étape : saisir dans le programme en Pascal, l'instruction *inline* correspondante, en répétant le code exécutable (qui apparait, sur le listing ci-dessus, dans la colonne située entre l'adresse et le code en assembleur), en remplaçant $1234 par le nom de la variable adéquate. Le lecteur pourra vérifier que, pour l'exemple vu plus haut, il retrouvera la même codification de l'instruction *inline* de la routine RemplirZone, imprimée au paragraphe 2.1. Pour faciliter l'interprétation et la maintenance des instructions *inline*, il est conseillé de placer en commentaire l'instruction assembleur équivalente sur chaque ligne.

2.3 Directives inline.

Bien qu'elles puissent être considérées comme des macros (équivalentes aux macros de l'assembleur), les directives *inline* sont très semblables aux instructions *inline*. Elles permettent d'écrire des séquences de code exécutable, présentées comme des procédures ou des fonctions, avec toutefois une particularité : au moment d'appeler une telle routine, ce n'est pas une instruction *CALL* qui est générée (comme pour une routine normale), mais simplement le code exécutable de la routine qui est inséré à l'endroit de l'appel. Ce qui revient à dire qu'au moment d'être appelée, une directive *inline* est transformée en instruction *inline*.

La syntaxe d'une directive *inline* est identique à celle d'une instruction *inline*, et donc aussi complexe. Par conséquent, l'usage de ces routines doit se limiter à de courtes séquences de code. Le fait que la séquence soit répétée à chaque appel renforce encore cette recommandation (économie d'espace).

Les exemples les plus simples de directives *inline* sont les deux procédures suivantes :

```
procedure ActiveInterruption; inline ($fa); {sti}
procedure DesactiveInterruption; inline ($fb); {cli}
```

utilisées pour implémenter les instructions SetInterrupt et ClearInterrupt du langage assembleur, qui autorisent ou interdisent la réalisation d'une interruption par le système opérationnel.

Notons l'absence de directives *begin* et *end* dans le corps de la procédure, ce qui est une caractéristique des directives *inline*.

Un autre exemple plus complet est l'appel d'une procédure par son adresse au lieu de son nom. Imaginons la situation suivante :

- une unité (appelée inlin2) réalise plusieurs tâches, sans importance pour notre exemple, mais qui effectuent certains contrôles et affichent des messages d'erreur quand c'est nécessaire ;
- l'auteur de cette unité ne souhaite pas en diffuser le code source, mais seulement le fichier d'extension *.TPU* ;
- d'autre part, il souhaite que le programmeur qui va utiliser cette unité ait la possibilité d'adapter les messages d'erreur dans sa langue.

Ceci peut se résoudre par une directive *inline* comme présenté ci-dessous :

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{                          exemple de directive inline                       }
{            (unit Inlin2 disponible dans le fichier inlin2.pas)             }
{----------------------------------------------------------------------------}

unit inlin2;

interface
uses crt;
var
  PtrAfficherErreur : pointer;

procedure NimporteQuoi;

implementation

{$F+}
```

```
procedure AfficherErreur(n : byte);
{$F-}
{fait appel à la routine pointée par PtrAfficherErreur}
  inline($FF/$1E/>PtrAfficherErreur);  {call dword ptr [PtrAfficherErreur]}

procedure AfficherErreurStandard(n : byte);
var
  c : char;
begin
  gotoxy(1,25);
  case n of
  1 : write('Disc full');
  2 : write('Drive not ready');
  3 : write('Insuficient memory');
  {.......}
  end;
  c := readkey;
end;

procedure NimporteQuoi;
{.........}
begin

  {instructions de la procédure}

  if true
    then AfficherErreur(3);

  {autres instructions}
end;

begin
  PtrAfficherErreur := @AfficherErreurStandard;
end.
```

La procédure AfficherErreur, utilisée pour afficher un message d'erreur dans la partie d'implémentation de l'unité, n'est pas une procédure normale mais une directive *inline* qui réalise une instruction *CALL* vers l'adresse mémorisée dans le pointeur PtrAfficherErreur. La partie d'initialisation de l'unité initialise la valeur le PtrAfficherErreur avec l'adresse de la procedure AfficherErreurStandard qui affiche des erreurs en anglais.

Le programme qui va utiliser l'unité, s'il ne modifie pas la valeur PtrAfficherErreur, va provoquer l'affichage des erreurs en anglais. Toutefois, il est très facile de forcer l'unité à afficher les messages en français :

- il suffit d'écrire dans le programme une routine personnalisée, structurée de la même façon que AfficherErreurStandard mais avec des messages en français ;
- et initialiser la valeur de PtrAfficherErreur, déclaré dans la partie d'interface de l'unité, avec l'adresse de la procédure personnalisée.

Le programme ci-dessous en est un exemple :

```
{--------------------------cours de Pascal-Avancé--------------------------}
{                                                                           }
{                          exemple de directive inline                      }
{          (programme inlin3 disponible dans le fichier inlin3.pas)         }
{---------------------------------------------------------------------------}

program inlin3;

uses crt, inlin2;

{$F+}
procedure AfficherErreurPersonnelle(n : byte);
{$F-}
var
  c : char;
begin
  gotoxy(1,25);
  case n of
  1 : write('Disque plein');
  2 : write('Unité de disquette non prête');
  3 : write('Mémoire insuffisante');
  {......}
  end;
  c := readkey;
end;

begin
  PtrAfficherErreur := @AfficherErreurPersonnelle;
  {.........}
  NimporteQuoi;
  {.........}
end.
```

Il est obligatoire de respecter certaines règles dans la programmation des routines qui vont être appelées par l'intermédiaire de directives *inline* :

- elles doivent avoir le même nombre de paramètres que ceux déclarés dans l'en-tête de la directive ; le type des paramètres doit également correspondre ;
- elles doivent être déclarées comme adresses longues, en utilisant la directive de compilation $F+ ;
- elles ne peuvent faire partie d'un autre bloc de code (elles ne peuvent être internes à une autre routine) ;
- le compilateur n'a pas possibilité de faire une quelconque vérification au sujet de ces règles mais, en cas d'erreur, les résultats de l'exécution seront imprévisibles.

3 Interface avec le langage assembleur.

Turbo-Pascal permet l'utilisation de routines externes écrites dans un autre langage capable de créer des fichiers d'extension *.OBJ* dans le format standard défini par Intel. Cette possibilité est principalement utilisée avec des routines écrites en langage assembleur. Dans un premier chapitre, nous allons expliquer comment Turbo-Pascal transmet et reçoit des paramètres à une procédure ou une fonction. La routine en assembleur devra être écrite en respectant ces conventions, pour garantir son bon fonctionnement. Ensuite, nous décrirons l'utilisation de la routine dans le programme Turbo-Pascal et un exemple complet sera détaillé.

Inversement, l'appel d'une routine en Turbo-Pascal par un autre langage ne peut être réalisé de cette manière car le Turbo-Pascal ne génère pas de code objet (fichier d'extension *.OBJ*).

3.1 Règles pour l'appel de routines et le passage de paramètres.

Le compilateur Turbo-Pascal traduit l'appel à une routine (procédure ou fonction) en une instruction *CALL* du langage assembleur 80x86. Les éventuels paramètres sont communiqués à la routine, ou retournés au programme appelant, par l'intermédiaire de la pile du 80x86.

En fonction du type ou de la déclaration des paramètres dans l'en-tête de la fonction, la méthode utilisée est différente.

3.1.1 Cas des paramètres passés par adresse.

Il s'agit des paramètres qui sont précédés du mot réservé *var* dans l'en-tête de la routine. Dans tous les cas, indépendamment du type, le programme stocke sur la pile un pointeur qui pointe vers l'adresse réelle du paramètre dans la mémoire.

3.1.2 Cas des paramètres passés par valeur.

Dans ce cas, la règle est plus complexe car elle dépend du type de paramètre et également de sa taille :

- pour un paramètre de type *shortint, byte, char, integer, word,* la valeur du paramètre est transmise comme un mot de deux octets (le premier étant le moins significatif) ; dans le cas de *shortint, byte* et *char* qui occupent seulement un octet, la valeur est stockée dans l'octet le moins significatif du mot, le plus significatif étant ignoré ;
- un paramètre de type *longint* est transmis en deux mots, le premier étant le moins significatif ;
- un paramètre de type *boolean* est transmis comme un *byte* de valeur 0 (pour *false*) ou 1 (pour *true*) ;
- un paramètre de type *énuméré* est transmis comme un *byte* (si le type contient 256 valeurs ou moins) ou comme un *word* (si le type contient plus de 256 valeurs) ;
- la valeur d'un paramètre de type *real* occupe 6 octets sur la pile ;
- les paramètres de type *single, double, extended* et *comp*, spécifiques du co-processeur mathématique, sont transmis respectivement sur 4, 8, 10 et 10 octets ; (attention à la compatibilité avec la version 4 du Turbo-Pascal où ces paramètres étaient transmis par la pile du 80x87) ;
- la valeur d'un paramètre de type *pointeur* occupe 4 octets sur la pile (le premier mot contient le déplacement, le second contient le segment) ;
- pour un paramètre de type *string*, un pointeur qui occupe 4 octets est stocké sur la pile et pointe vers l'adresse réelle du paramètre ;
- pour un paramètre de type *set*, un pointeur qui occupe 4 octets est stocké sur la pile et pointe vers une zone de 32 octets occupée par le paramètre ;
- pour un paramètre de type *array* ou *record* qui occupe moins de 5 octets, la valeur du paramètre est copiée sur la pile ; pour les paramètres de même type, mais de taille supérieure à 4 bytes, un pointeur est stocké sur la pile comme dans le cas d'un passage par adresse.

3.1.3 Cas des résultats de fonctions.

La méthode utilisée dépend également du type de fonction :

- le résultat d'une fonction de type *byte, boolean, shortint, char, énuméré* est retourné au programme appelant dans le registre AL ;
- le résultat d'une fonction de type *integer* ou *word* est retourné dans le registre AX ;
- le résultat d'une fonction de type *longint* est retourné dans les registres DX (mot le plus significatif) et AX (mot le moins significatif) ;
- le résultat d'une fonction de type *real* est retourné dans les registres DX (mot le plus significatif), BX (mot intermédiaire) et AX (mot le moins significatif) ;
- le résultat des fonctions de type *single, double, extended, comp*, spécifiques du co-processeur, est retourné sur la pile du 80x87 ;
- le résultat d'une fonction de type *pointeur* est retourné dans les registres DX (segment) et AX (déplacement) ;
- pour une fonction de type *string*, le processus est plus complexe : l'appelant réserve une zone temporaire sur la pile et transmet un pointeur qui pointe vers cette adresse, la routine mémorise son résultat dans cette zone où il sera récupéré par le programme appelant (voir exemple au chapitre 3.3).

3.1.4 Exemple de passage de paramètres.

Le petit programme PARAM.PAS, disponible sur la disquette, permet de vérifier quelques unes des règles énoncées ci-dessus. Voici le listing du code source de ce programme :

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{             programme pour illustrer le passage de paramètres              }
{                   (disponible dans le fichier param.pas)                   }
{----------------------------------------------------------------------------}

program param;
uses crt, dos, utiltas;

type
  TypePetitRecord = record
                      w : word;
                      c : char;
                      d : byte;
                    end;
  TypeGrandRecord = record
                      r : TypePetitRecord;
                      suivant : ^TypePetitRecord;
                    end;
const
  Entier : integer = 4096;
  Str    : string = 'Exemple de paramètre chaîne de caractères';
  Carac  : char = 'A';
  PetitRecord : TypePetitRecord = (w : $FFFF; c : 'X'; d : $FF);

var
  GrandRecord : TypeGrandRecord;
```

```
procedure AfficherPile(p : pointer; max : word; t : string);
{afficher à l'écran les "max" valeurs récemment stockées sur la pile,
 en lignes de 16 octets (soit 8 mots)                                  }
var
  l : word;

begin
  writeln;
  writeln('Pile ' + copy(t, 1, 70)+' :');
  writeln;
  write('        ');
  for l := 1 to 8 do write(l:5);
  l := 0;
  while l < max do
    begin
      if (l mod 8) = 0
        then begin
          writeln;
          write(PointeurHexa(p),'  ');
        end;
      write(MotHex(word(p^)), ' ');
      p := Ptr(seg(p^), ofs(p^)+2);
      inc(l);
    end;
  writeln;
  write('        ');
  for l := 1 to 8 do write(l:5);
end;

procedure SansNom(var i : integer; j : integer; s : string; Carac : char;
                  r1 : TypePetitRecord; r2 : TypeGrandRecord);
const
  Max = 16;
var
  OfsPile : word;
  p : pointer;
begin
  {les deux instructions ci-dessous permettent de mémoriser dans p
   la valeur du PointeurHexa vers la pile (SS:BP), au moment de l'arrivée
   dans la routine}
  inline($89/$e8/           {mov ax, ss}
         $89/$86/OfsPile);  {mov bp+OfsPile, ax}
  p := ptr(SSeg, OfsPile);

  AfficherPile(p, max, 'au début de SansNom');
end;

begin
  clrscr;
  {afficher les adresses et les valeurs des paramètres passés
   à la procédure SansNom}
  writeln('Variable      Adresse         Valeur');
  writeln;
  writeln('Entier        ',PointeurHexa(@Entier),'      ', Entier);
  writeln('Str           ',PointeurHexa(@Str),   '      ', Str);
  writeln('Carac         ',PointeurHexa(@Carac), '      ', Carac);
  with PetitRecord do
    writeln('PetitRecord    ',PointeurHexa(@PetitRecord),'      ', w, ' ', c, ' ', d);
  writeln('GrandRecord    ',PointeurHexa(@GrandRecord));
```

```
  SansNom(Entier, Entier, Str, Carac, PetitRecord, GrandRecord;
end.
```

Comme on le voit, le programme ne fait que :

- montrer une table où chaque ligne correspond à une variable et contient trois colonnes : le nom de la variable, son adresse (segment:déplacement) et sa valeur, au cas où elle a été initialisée ;

- et appeler la routine SansNom qui utilise la routine AfficherPile pour visualiser l'état de la pile à l'entrée de SansNom.

Il faut signaler que la routine AfficherPile peut être extraite de ce programme et utilisée ailleurs ; toutefois il sera toujours nécessaire de définir la valeur de p en utilisant l'instruction *inline* et les fonctions *SSeg* et *Ptr*.

Voici le résultat de l'exécution de PARAM.PAS, accompagné de commentaires :

```
Variable      Adresse         Valeur

Entier        2ABF:0002       4096
Str           2ABF:0004       Exemple de paramètre chaîne de caractères
Carac         2ABF:0104       A
PetitRecord   2ABF:0105       65535 X 255
GrandRecord   2ABF:0156

Pile au début de SansNom :

            1    2    3    4    5    6    7    8
2AFB:FED6  FFEE 05BF 0156 2ABF FFFF FF58 0041 0004
2AFB:FEE6  2ABF 1000 0002 2ABF 3209 4241 3A46 3130
            1    2    3    4    5    6    7    8
```

Commençons par une parenthèse :

- le dernier mot (sous le numéro 1, sur la première ligne), est seulement l'ancienne valeur de BP qui a été placée dans la pile par l'instruction PUSH BP, avant que l'instruction MOV BP,SP ne place en BP la valeur du déplacement sur la pile ;

- l'avant-dernier mot ($05C5) est l'adresse de retour dans le programme appelant, après l'exécution de la routine ; elle a été placée là par l'instruction *CALL* qui a appelé SansNome ; rappelons que si la routine était déclarée comme une adresse longue (directive $F+) deux mots seraient nécessaires pour stocker cette adresse (un pour le segment, un pour le déplacement).

Viennent ensuite les paramètres. Ils ont été stockés sur la pile par ordre d'apparition dans l'appel de SansNom, c'est pourquoi ils apparaissent en ordre inverse. Nous pouvons vérifier, en comparant les valeurs de la pile avec celles du tableau imprimé avant que :

- pour le paramètre GrandRecord, qui a été déclaré par valeur, seul un pointeur a été transmis (le mot 3 contient le déplacement, le mot 4 contient le segment) ; cela parce que GrandRecord est de type *record* et de taille supérieure à 4 ;

- alors que PetitRecord, de taille 4 a été passé par valeur : le mot 5 contient la valeur de w ($FFFF = 65535), le premier octet du mot 6 contient la valeur de c ($58 = X), le second octet contient la valeur de d ($FF = 255) ;

- le caractère Carac a été passé par valeur dans la partie la moins significative du mot 7 ($41 = A) ;

- pour le paramètre Str, qui est passé par valeur, seule l'adresse de Str a été chargée sur la pile (le mot 8 contient le déplacement et le mot 9 - le premier de la seconde ligne - contient le segment) ; la raison en est que Str est de type *string* ;

- enfin, le mot 10 contient la valeur de Entier (transmis par valeur), alors que les mots 11 et 12 contiennent l'adresse de Entier transmis par adresse (5E7F:0002).

3.2 Utilisation de routines en assembleur à l'intérieur du Turbo-Pascal.

3.2.1 Le point de vue du Turbo-Pascal.

La routine en langage assembleur, après avoir été compilée par un compilateur assembleur (Masm, Turbo-Assembler), devra être incorporée au programme pascal (ou à une unité) par l'intermédiaire de la directive $L suivie du nom du fichier d'extension *.OBJ* généré par le compilateur assembleur. Exemple d'utilisation : {$L routines.obj}.

Le nom de la procédure contenue dans le fichier et le nom du fichier peuvent être différents. D'ailleurs, le fichier *.OBJ* peut contenir plusieurs routines en langage assembleur.

Pour un fonctionnement correct du compilateur Turbo-Pascal, les routines du fichier *.OBJ* à utiliser doivent être déclarées comme *external*. Voici un exemple :

```
function Majuscules(s : string) : string; external;
```

La déclaration peut être placée en un endroit quelconque où pourrait être déclarée un routine normale, avant l'utilisation de la routine dans le programme. Dans le cas d'une unité, elle peut être placée dans la partie *interface* comme dans la partie *implementation*. D'autre part, la position de la directive $L associée n'a pas d'importance, elle doit seulement être placée en dehors du corps du programme et en dehors de la partie d'initialisation d'une unité. Plusieurs fichiers *.OBJ* peuvent être intégrés à un même programme pascal, chacun l'étant par une directive $L.

3.2.2 Le point de vue de l'assembleur.

Pour récupérer les paramètres, ou retourner des résultats au programme appelant, la routine en assembleur doit évidemment respecter les règles énoncées au chapitre 3.1. Ce qui impose :

- un début de routine de la forme :

```
PUSH BP
MOV BP,SP
```

- la récupération des paramètres éventuels, à partir de BP+4 pour une routine NEAR, à partir de BP+6 pour une routine FAR, en respectant les règles de passage de paramètres ;

- une sortie de routine de la forme :

```
MOV SP,BP
POP BP
RET taille totale des paramètres
```

De plus :

- toutes les procédures et fonctions déclarées dans la routine en assembleur doivent appartenir au segment CODE et toutes les variables privées de cette routines doivent

appartenir à un segment nommé DATA ; un nom de classe ne peut pas être attribué aux segments ; une valeur initiale donnée à ces variables sera ignorée par le Turbo-Pascal ;

- les routines en assembleur, appelées par le programme Turbo-Pascal, doivent être déclarées comme publiques par la directive PUBLIC ;

- les routines en assembleur ont accès à toutes les variables et routines publiques du programme ou de l'unité Turbo-Pascal : il suffit de les déclarer comme EXTRN dans le source en assembleur.

3.3 Un exemple de routine en assembleur utilisée en Turbo-Pascal.

Le but de l'unité présentée ci-dessous comme exemple est de fournir une fonction pour transformer une chaîne de caractères en majuscules, en prenant en compte les minuscules accentuées, ce que ne ferait pas la fonction UpCase disponible dans le Turbo-Pascal. Ceci peut servir pour trier des variables de type *string* et éviter que, par exemple dans le cas d'une table de pays, Pérou ne se retrouve placé après Philippines (ce qui serait le cas en respectant l'ordre ASCII pour lequel "é" est placé derrière toutes les lettres non accentuées).

On peut consulter ci-dessous le listing de la routine en assembleur Majuscules et le listing de l'unité Majuscul qui l'utilise.

```
;{---------------------------cours de Pascal-Avancé---------------------------}
;{                                                                            }
;{                          exemple de routine en assembleur                  }
;{                        (disponible dans le fichier majuscul.asm)           }
;{----------------------------------------------------------------------------}

; routine pour transformer une chaîne de caractères en majuscules,
; en tenant compte des minuscules accentuées

code    segment byte public
        assume cs:code
        public Majuscules

TabMaj label byte  ;tableau des majuscules qui correspondent aux
                   ;minuscules de code ASCII > 129
        db  'U', 'E', 'A', 'A', 'A', 'A', 'C', 'E', 'E', 'E'
        db  'I', 'I', 'I', 'A', 'A', 'E', 'Æ', 'Æ', 'O', 'O'
        db  'O', 'U', 'U', 'Y', 'O', 'U', '¢', '£', '¥', 'R'
        db  'F', 'A', 'I', 'O', 'U', 'N', 'N', 'A', 'O'

Majuscules proc far ; (function Majuscules(S : string) : string;)
        push    bp
        mov     bp, sp
        push    ds
        cld
        lds     si,ss:[bp+6]      ;DS:SI => S
        les     di,ss:[bp+10]     ;ES:DI => résultat de la fonction
        lodsb                     ;AL = Length(S)
        stosb                     ;initialiser la taille du résultat
        mov     ch, 0
        mov     cl, al
        jcxz    fin               ;terminer quand CX devient nul
Trans:
        lodsb                     ;prochain caractère en AL
        cmp     al,'a'
        jb      Stock             ;si AL < 'a', ne rien faire
        cmp     al,167
```

```
        ja      Stock           ;si AL > 167, ne rien faire
        cmp     al,'z'
        ja      MinAccentuee    ;traiter la minuscule accentuée
        xor     al, 20h         ;convertir une minuscule non accentuée
Stock:  stosb                   ;mémoriser le caractère transformé
        loop    Trans

fin:
        pop     ds
        pop     bp
        ret     4

MinAccentuee:                   ;transforme une minuscule accentuée
        cmp     al,129
        jb      Stock           ;si AL < 129, ne rien faire
        sub     al,129
        mov     bx,offset TabMaj
        xlat    cs:[bx]         ;rechercher dans le tableau, la majuscule correspondante
        jmp     Stock
Majuscules endp


code    ends
end
```

```
{----------------------------cours de Pascal-Avancé----------------------------}
{                                                                               }
{             exemple d'unité utilisant une routine écrite en assembleur        }
{                       (disponible dans le fichier majuscul.pas)               }
{-------------------------------------------------------------------------------}
unit majuscul;
interface

function Majuscules(s : string) : string;


implementation
{$L MAJUSCUL.OBJ}
function Majuscules(s : string) : string; external;
end.
```

Remarquons que ce traitement pourrait être réalisé en Pascal. Cependant, l'instruction XLAT de l'assembleur présente l'avantage d'être beaucoup plus rapide que son équivalent dans un langage évolué.

4 Définir et installer ses propres interruptions.

4.1 Définition et installation d'une interruption.

Il est possible, en Turbo-Pascal, de définir et installer ses propres interruptions pour les substituer à celles du BIOS ou du DOS.

Une procédure destinée à être installée comme interruption doit être déclarée comme

interrupt, et déclarer les registres comme pseudo-paramètres comme dans l'exemple ci-dessous :

```
procedure NouvInt(flags,ci,ip,ax,bx,cx,dx,si,di,ds,es,bp : word) : interrupt;
```

Le contenu de la procédure peut être défini comme pour une procédure normale. Toutefois, dans le cas d'une interruption "hard" (interruption exécutée automatiquement par le système opérationnel), les routines d'entrée/sortie et d'allocation de mémoire du Turbo-Pascal ne peuvent pas être utilisées, ni les fonctions du DOS, car elles ne sont pas réentrantes. A leur place, on doit utiliser les interruptions du BIOS.

Pour installer une interruption, utiliser les procédures GetIntVec pour sauver l'adresse originale de l'interruption et SetIntVec pour installer la nouvelle ou réinstaller l'ancienne. La syntaxe est :

```
GetIntVec(NumInt, p);
SetIntVec(NumInt, q)
```

où NumInt est un octet qui contient le numéro de l'interruption à traiter, p est une variable de type *pointer*, pour mémoriser l'ancienne adresse, et q est l'adresse de la procédure qui va devenir la nouvelle interruption (voir exemple ci-dessous).

4.2 Exemple d'interruption.

On trouvera ci-dessous le listing de la routine TEMPS qui permet à un programme applicatif de montrer sur l'écran, en permanence, l'heure courante.

```
{---------------------------cours de Pascal-Avancé--------------------------}
{                                                                            }
{                      exemple d'installation d'interruption                 }
{                      (disponible dans le fichier temps.pas)                }
{----------------------------------------------------------------------------}
unit Temps;
interface
uses crt,dos;
const
  {position et attributs pour afficher l'heure}
  LigHeur : byte = 3;
  ColHeur : byte = 72;
  AttHeur : byte = 0;
  TimeOn  : boolean = false;
var
  {adresse où débute la mémoire vidéo et taille d'une ligne}
  ScrAddr,
  TailLig: word;

procedure TempsOn;   {installe l'horloge}
procedure TempsOff;  {efface l'horloge  }

implementation
var
  SavInt1c : pointer;
const

  ecr_cga : word = $b800;  {adresse pour l'écran cga}
  ecr_mono: word = $b000;  {adresse pour lécran mono}
  AncInterv : word = $ffff;{mémorise l'heure antérieure en secondes}
```

```
procedure AfficherTemps(l, c, a : byte; var s : string);
var
  i : byte;
  j : word;
  w : word;
  b :array[1..2] of byte absolute w;
begin
  j := pred(l)*TailLig + 2*pred(c);
  for i := 1 to 8 do
    begin
      w := MemW[ScrAddr:j];
      b[1] := byte(s[i]);
      if a <> 0 then b[2] := a;
      MemW[ScrAddr:j] := w;
      inc(j, 2);
    end;
end;

{$F+}
procedure Int1c(flags, cs, ip, ax, bx, cx, dx, si, di, ds, es, bp : word);interrupt;

{routine à installer comme nouvelle interruption 1ch}
var
  Minutes,
  Secondes : byte;
  Interv,
  Heure    : word;
  StrHeur : string[8];
  AuxStr : string[3];
  Regs : registers;

begin
  {récupère l'heure directement en mémoire}
  Heure   := MemW[$0040:$006e];
  Interv := round((MemW[$0040:$006c])*1.0 / 18.2);{Heure en secondes}
  if AncInterv  <> Interv
    then begin {si l'heure a changé d'au moins une seconde, afficher à l'écran}
      AncInterv := Interv;
      {calcul de l'heure}
      Minutes := Interv div 60;
      Secondes := Interv mod 60;
      str(Heure+100:3,AuxStr);
      StrHeur := copy(AuxStr,2,2)+':';
      str(Minutes+100:3,AuxStr);
      StrHeur := StrHeur + copy(AuxStr,2,2)+':';
      str(Secondes*+100:3,AuxStr);
      StrHeur := StrHeur + copy(AuxStr,2,2);

      {cherche le type d'écran}
      Regs.ah := $0f;
      intr($10, Regs);
      if Regs.al = 7
        then begin
          ScrAddr := ecr_mono;
          TailLig := 160;
        end
```

```
        else if Regs.al <= 1
          then begin
            ScrAddr := ecr_cga;
            TailLig := 80;
          end
          else if Regs.al <= 3
            then begin
              ScrAddr := ecr_cga;
              TailLig := 160;
            end;
      {$V-}
      AfficherTemps(LigHeur,ColHeur,AttHeur,StrHeur);
      {$V+}
    end;
end;
{$F-}

procedure TempsOn;
{installe la nouvelle interruption 1cH}
begin
  if not TimeOn
    then begin
      GetIntVec($1c, SavInt1c);
      inline($fa);
      SetIntVec($1c, @Int1c);
      inline($fb);
      TimeOn := true;
    end;
end;

procedure TempsOff;
{restaure le vecteur initial}
begin
  if TimeOn
    then begin
      inline($fa);
      SetIntVec($1c, SavInt1c);
      inline($fb);
      TimeOn := false;
    end;
end;
end.
```

L'interface met à disposition :

- des variables initialisées qui définissent la position et la couleur de l'heure affichée ;
- la procédure TempsOn pour lancer la visualisation de l'heure ;
- la procédure TempsOff pour désactiver la visualisation de l'heure sur l'écran.

Parmi les 256 interruptions disponibles, l'interruption 1cH est une de celles réservées à l'utilisateur par le BIOS (l'autre est 1bH et est déjà utilisée par le DOS pour générer un ^C). La particularité de cette interruption est d'être exécutée automatiquement, en moyenne 18,2 fois par seconde, en accord avec la pulsation de l'horloge interne. Initialement, elle ne fait rien ; c'est pourquoi le programme peut lui associer une fonction comme la visualisation de l'heure, qui doit être remise à jour à chaque seconde. C'est ce que fait l'unité Temps.

Signalons encore dans l'exemple vu plus haut :

- que durant l'appel de SetIntVec (dans les procédures TempsOn et TempsOff), il est interdit d'exécuter une interruption pour ne pas risquer d'interférer dans l'échange des vecteurs d'interruption ; l'interruption redevient autorisée immédiatement après ;

- que, dans la procédure Int1c (nouvelle interruption 1cH), on utilise l'interruption 10H pour visualiser à l'écran ou déterminer le type d'écran et que l'heure est récupérée directement en mémoire, au lieu d'utiliser la fonction correspondante du DOS (2AH) ;

- un test est fait pour vérifier que le temps a varié d'au moins une seconde, pour n'afficher l'heure qu'une fois par seconde, au lieu de 18 fois ;

- il est indispensable d'exécuter TempsOff avant la fin du programme, pour restituer l'interruption originale ; si on ne le fait pas, le système continuera d'exécuter l'interruption Int1c, même si un autre programme a été chargé à la place du programme AffTemps.

```
{--------------------------cours de Pascal-Avancé--------------------------}
{                                                                           }
{                           exemple d'interruption                          }
{                    (disponible dans le fichier afftemps.pas)              }
{---------------------------------------------------------------------------}
uses crt, Temps;
var
  i, j : byte;
  SaveExit : pointer;
begin
  clrscr;
  i := 1;
  j := 1;
  TempsOn;
  {corps du programme}
  Randomize;
  repeat
    gotoxy(i, j); clreol;
    i := Random(42)+1;
    j := Random(22)+4;
    gotoxy(i, j);
    write('Appuyez sur une touche quelconque pour arrêter l''horloge');delay(100);
  until keypressed;
  TempsOff;
end.
```

Enfin, pour être complet, l'exemple ci-dessus aurait besoin d'être amélioré pour prévenir une sortie anormale du programme (Break ou erreur fatale). Il faudrait pour cela que le programme installe sa propre routine de sortie en cas d'erreur.

Chapitre VI

Problèmes rencontrés dans le développement de grands systèmes. Recommandations pour les éviter.

1 Quelques limitations rencontrées.

Les problèmes causés par le manque d'espace sur disque ou disquette ne seront pas abordés dans ce chapitre, dans la mesure où nous les considérons évidents à résoudre. L'utilisation et le développement de logiciels complexes n'a pas lieu d'être sans l'utilisation de disques durs, aussi bien pour l'implantation des logiciels utilisés pour le développement que pour l'implantation des fichiers-sources, des exécutables et des fichiers de tests du système développé. Par conséquent, nous considérons comme une condition minimum l'utilisation d'un disque dur de capacité suffisante.

La plupart des autres problèmes rencontrés sont liés à la disponibilité de mémoire pendant l'exécution du logiciel développé :

- la taille maximum du segment de données est de 64 K octets, pour un programme exécutable créé par le Turbo-Pascal ;
- la taille maximum du segment de code d'un programme ou d'une unité ne peut dépasser 64 K octets ; mais la taille totale du code exécutable (programme + unité) n'est pas limitée, sinon par la taille de la mémoire disponible ;
- la taille maximum du segment de la pile est également de 64 K octets pour un programme exécutable ;
- la taille maximum du tas, jointe à celle de la table des fragments, est limitée à la mémoire disponible ;
- la taille de la mémoire accessible au DOS est de 640 K octets ;
- la taille de la table de réadressage du fichier .EXE ne peut dépasser 64 K octets, c'est à dire 16384 éléments.

Nous allons détailler chacune de ces limitations dans les paragraphes suivants, et définir des recommandations pour les contourner.

2 Limite du segment de données.

Pour un programme écrit en Turbo-Pascal, le segment de données contient :

- toutes les variables déclarées dans un programme principal (variables globales du programme) ;
- toutes les variables définies dans la partie d'interface et dans la partie d'implémentation de chaque unité utilisée par le programme (variables globales des unités) ;
- toutes les constantes typées (qui peuvent aussi être nommées variables initialisées) : celles du programme principal, celles des unités et celles des procédures et fonctions.

La limite du segment de données est en réalité de 65520 octets (légèrement inférieure à 64 K octets). Le problème peut se rencontrer pendant la compilation ou pendant l'édition de liens.

2.1 Segment de données trop grand pendant la compilation.

Cette situation peut provoquer plusieurs messages d'erreur :

- erreur **22 Structure too large** : la taille du type d'une structure dépasse 65520 octets ; il n'y a pas d'autre solution que de la diminuer ; cette erreur se produit rarement et en général dans le cas de la définition d'une matrice de grande dimension ;
- erreur **96 Too many variables** : l'espace occupé par la totalité des variables globales ou initialisées dont la déclaration a été rencontrée au cours de la compilation d'un même module (programme ou unité) dépasse 65520 octets ; partager le module en plusieurs modules plus petits afin de répartir entre ceux-ci les déclarations de variables et, éventuellement, les diverses routines (procédures et fonctions).

2.2 Segment de données trop grand pendant l'édition de liens.

Il s'agit du cas de l'erreur **49 Data segment too large**. Pour résoudre ce problème, il suffit de déclarer les variables structurées de plus grande taille comme étant des variables dynamiques et les allouer sur le tas en utilisant les procédures *New* ou *GetMem* (voir chapitre II de ce manuel). Ainsi, seul le pointeur (4 octets) sera stocké dans le segment de données ; en contrepartie, l'espace disponible sur le tas devra être suffisant pour allouer la variable.

3 Limite du code exécutable d'un module.

Ceci peut se produire pendant la compilation d'un module et provoque une erreur **48 Code segment too large** (segment de code trop grand). Dans le cas d'un programme, il suffit d'extraire une partie des procédures et fonctions et de les rassembler dans une unité qui sera compilée séparément. Dans le cas d'une unité, il suffit de diviser celle-ci en plusieurs unités. Dans les deux cas, cette erreur traduit une mauvaise modularisation de la programmation et devrait conduire à réexaminer la structure de l'application pour la diviser en plusieurs modules.

4 Limite du segment de pile.

Comme nous l'avons vu, la pile sert pour stocker toutes les variables locales d'une routine et les paramètres passés aux routines. La taille de la pile peut être spécifiée au moment de la compilation par la directive $M, mais de toute manière, elle doit être comprise entre 1024 et 65520 octets (la valeur par défaut est de 16 K octets).

4.1 Problème de taille de pile pendant la compilation.

Il provoque une erreur **96 Too many variables**, quand l'espace total occupé par les variables locales d'une routine dépasse 65520 octets. L'espace total doit être diminué, ou alors la routine doit être divisée en plusieurs routines où les variables seront redistribuées. L'utilisation de variables dynamiques est également une bonne solution : seul le pointeur (4 octets) est alors conservé sur la pile, la variable étant allouée sur le tas.

4.2 Problème de taille de pile pendant l'exécution.

Si l'erreur se produit dans un module qui a été compilé avec la directive $S+, un message **202 Stack overflow error** est produit et l'exécution du programme est définitivement stoppée. Si l'erreur se produit dans un module compilé avec la directive $S-, aucun message ne sera émis et les conséquences sont imprévisibles et vont en général jusqu'au blocage du clavier de l'ordinateur.

Le risque est plus grand quand une procédure utilise beaucoup de variables locales ou en cas de récursivité.

Le programme ci-dessous (recurs1) qui calcule la fonction factorielle d'un entier (n) ne fonctionne plus dès que n dépasse une certaine valeur. A cause du fait que la fonction *Factorielle* définie est appelée récursivement n-1 fois et qu'à chaque fois le résultat est gardé sur la pile. La taille de la pile étant limitée à 16384 octets, l'erreur 202 se produit pour n plus grand que 792 (si on compile le programme avec $F+, l'erreur se produit pour n supérieur à 719, car l'adresse de retour occupe deux octets de plus à chaque appel).

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{                           exemple de récursivité                           }
{                    (disponible dans le fichier recurs1.pas)                }
{----------------------------------------------------------------------------}

{$N+,E+,S+}
{$M 16384, 0, 655360}
program recurs;
var
  n : word;

function Factorielle(n : word) : extended;
begin
  if n = 0
    then Factorielle := 1.0
    else Factorielle := Factorielle(n-1) * n; {récursivité}
end;

begin
  n := MaxInt;
  while n <> 0 do
    begin
      write('entrer la valeur positive de n (0 pour terminer) : ');
      readln(n);
      if n <> 0
        then writeln(' n! = ', Factorielle(n));
    end;
end.
```

Si on élimine la récursivité, le programme fonctionnera pour des valeurs très supérieures de n, même avec une pile de taille minimum, comme on peut le vérifier avec la seconde version du même programme, imprimée ci-dessous :

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{                            exemple de récursivité                          }
{                       (disponible dans le fichier recurs2.pas)             }
{----------------------------------------------------------------------------}

{$N+,E+,S+}
{$M 1024, 0, 655360}
program recurs2;
var
  n : word;

function Factorielle(n : word) : extended;
var
  i : word;
  f : extended;
begin
  f := 1.0;
  if n <> 0
    then for i := 1 to n do f := f * i;
  Factorielle := f;
end;

begin
  n := MaxInt;
  while n <> 0 do
    begin
      write('entrer la valeur positive de n (0 pour terminer) : ');
      readln(n);
      if n <> 0
        then writeln(' n! = ', Factorielle(n));
    end;
end.
```

En général, on peut adopter plusieurs attitudes afin d'optimiser la taille de la pile :

- augmenter la taille de la pile (jusqu'à un maximum de 65520), sachant que l'espace disponible pour charger le programme ou pour le tas va être diminué d'autant ;
- éviter ou optimiser les cas de routines récursives ;
- transformer certains paramètres passés par valeur en paramètres passés par adresse, notamment ceux de type *string* ou les structures de grande taille.

Ce dernier point mérite un commentaire : quand une structure ou une chaîne de caractères est passée par valeur, nous avons vu (au chapitre V.3.1) que seul un pointeur est passé en paramètre, ce qui, à première vue, paraît identique au cas des paramètres par adresse. Cependant, dans le cas du paramètre par valeur, pour empêcher que la variable originale ne soit modifiée par la routine, une variable locale est créée sur la pile et le contenu de la variable originale y est copié. Si le paramètre est transformé en paramètre par adresse, un espace de la taille de la variable sera économisé (ainsi que le temps passé à copier la structure originale vers la structure locale) mais, en contrepartie, il devient nécessaire de prendre des précautions pour que la routine n'altère pas le contenu du paramètre. Dans le cas où cela se produit, on peut parfois se contenter de copier certains champs de la structure vers des variables locales.

Le programme ci-dessous ne sert qu'à illustrer le précédent paragraphe : il produira une erreur 202 lors de son exécution, car la taille de Grille1 est de 2000 octets alors que la pile disponible est de 1024 octets.

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{                        exemple d'optimisation de la pile                   }
{                     (disponible dans le fichier parval.pas)                }
{----------------------------------------------------------------------------}

program parval;
{$S+,M 1024, 0, 655360}              {pile limitée à 1024 octets}
type
  Ecran = array[1..25]               { 25 lignes de }
            of array[1..80] of word; { 80 colonnes  }

var
  Zone1 : Ecran;

procedure UtiliseEcran(Zone : Ecran);{Zone passé par valeur}
begin  {n'importe quoi}
end;

begin
  UtiliseEcran(Zone1);
end.
```

Pour solutionner le problème, sans augmenter la taille de la pile, il suffit de transformer l'en-tête de UtiliseEcran en :

```
procedure UtiliseEcran(var : Zone : Ecran);      {Ecran passé par adresse}
```

5 Limite de mémoire disponible.

Les problèmes causés par le manque de mémoire se produisent à plusieurs niveaux.

5.1 Manque de mémoire disponible pendant la compilation ou l'édition de liens.

Cette erreur se traduit par le message **1 Out of memory** et peut être évitée de plusieurs manières :

- vérifier tout d'abord que la compilation génère le code exécutable dans la mémoire (item *Destination* du sous-menu *Compilation*) ; si c'est le cas, faire la modification nécessaire pour générer le code exécutable sur disque ;
- vérifier que le tampon de l'édition de liens a été configuré pour être placé sur le disque (Sous-item *Link Buffer* de l'item *Compiler* du menu *Options*) ; si c'est le cas, utiliser la directive /L pour placer le tampon de l'édition de liens sur le disque, ce qui libèrera la mémoire qu'il occupait ;
- si l'ordinateur dispose d'une mémoire EMS, utiliser le programme TINST.EXE d'installation pour vérifier qu'il autorise l'utilisation de la mémoire EMS pour stocker le tampon de l'éditeur du programme source ;
- retirer, de la mémoire, les programmes résidents (comme SideKick) ;
- diminuer éventuellement la valeur du paramètre BUFFERS du fichier CONFIG.SYS, s'il est très grand (supérieur à 20) ; trop le diminuer pénaliserait la vitesse des opérations d'entrée/sortie avec le disque ;
- compiler avec les directives $S- et $R-, ce qui entraîne quelques inconvénients : il n'y aura plus de contrôle de l'espace disponible sur la pile, ni des limites des indices de matrices,

ce qui peut compliquer la recherche d'une erreur de ce type dans le programme, en phase de test ;

- si vous utilisez Turbo.exe, essayez plutôt de compiler avec Tpc.exe qui occupe moins d'espace en mémoire, puisqu'il n'édite pas le fichier-source.

Si le problème n'est pas résolu en appliquant une ou plusieurs de ces recettes, il est probable qu'il faudra partager le programme en modules plus petits et les compiler séparément.

5.2 Manque de mémoire disponible pendant l'exécution.

Dans l'environnement du Turbo.exe, cette situation se traduit par une erreur **108 Not enough memory to run the program.** Le plus simple est alors de générer le code exécutable sur le disque (fichier *.EXE*) et sortir de l'environnement intégré pour exécuter le fichier *.EXE*.

Toutefois, dans certains cas, il est indispensable d'exécuter le programme dans l'environnement intégré (pour utiliser le *Debugger*, par exemple). On peut alors tenter de résoudre le problème en appliquant les recettes du paragraphe 5.1 pour diminuer l'occupation de mémoire.

Si l'erreur persiste ou si l'erreur **Insufficient memory** ou équivalente, apparaît (dans le cas d'un fichier *.EXE*), la seule solution est l'utilisation d'*overlays* (voir chapitre 1). Si le programme a été bien structuré dès le début, la transformation des unités principales en *overlays* ne doit pas poser de problèmes.

6 Limite du tas disponible durant l'exécution.

Puisque la mémoire disponible pour être utilisée comme tas est la différence entre la mémoire totale et la mémoire occupée par le programme chargé, il est évident que les causes identifiées au paragraphe 5.2, et les solutions envisagées, restent valables pour le cas de mémoire dynamique insuffisante.

Toutefois, d'autres précautions peuvent être prises. Bien qu'elles s'appliquent également pour résoudre le manque de mémoire totale, elles sont exposées ici car elles sont liées à la gestion du tas.

Le message **Insufficient memory** peut se produire, à cause du manque de place sur le tas, lors du chargement du programme, si la valeur minimale du tas est supérieure à 0 (second paramètre de la directive $M). Vérifier que la valeur indiquée est correcte.

Nous avons déjà vu au chapitre II comment contrôler le manque de place sur le tas en phase d'exécution. Il est très important de contrôler ces erreurs de manière à éviter l'abandon du programme : dans certains cas, l'impossibilité d'ouvrir une fenêtre par manque de mémoire peut être une péripétie sans importance ; il suffit d'en avertir l'utilisateur et de continuer le traitement. Dans tous les cas, on doit chercher à :

- optimiser la taille des variables dynamiques à allouer dans le programme et les unités ; un cas édifiant est la valeur du paramètre PageSize de Turbo-Access : la valeur par défaut, calculée par TaBuild, correspond à une utilisation de 64 K octets de mémoire mais, en diminuant la valeur de PageSize, on peut réduire de façon significative cet espace, ce qui peut permettre de dépasser la limite de mémoire ; il est évident, qu'en contrepartie, la fréquence des accès au disque sera augmentée ;

- libérer les variables allouées, dès qu'elles n'ont plus d'utilité ; ce qui augmentera l'espace disponible pour l'allocation d'autres variables.

7 Limite de la table de réadressage.

7.1 Description du problème.

L'erreur provoquée par cette limite se produit pendant l'édition de liens. Il s'agit de l'erreur **107 Too many relocations items** : trop d'éléments dans la table de réadressage. Pour la comprendre, il est nécessaire d'examiner la structure d'un fichier d'extension *.EXE* qui est divisé en deux parties distinctes :

- un en-tête qui contient des informations de contrôle qui permettront de charger en mémoire le programme exécutable ;
- le code exécutable lui-même et les variables initialisées du programme.

L'en-tête contient une partie de taille fixe (28 octets) qu'il est inutile de détailler ici et une partie de taille variable, que nous appellerons *table de réadressage*, où sont stockées les informations nécessaires à l'ajustement des adresses de certains éléments du programme (variables ou instructions). Cet ajustement est nécessaire car l'édition de liens prépare le programme comme si celui-ci était chargé au déplacement 0 du segment 0. Quand on l'exécute, le programme n'est jamais chargé dans le segment 0, mais dans la mémoire disponible derrière le DOS et les éventuels programmes résidents, ce qui oblige à recalculer les adresses dans la mémoire, en accord avec la table de réadressage.

Chaque élément de cette table est constitué de deux mots, le premier est l'adresse de l'élément (variable ou instruction) à l'intérieur de son propre segment, le second est le déplacement entre le segment de l'élément et le segment où commence le programme. Pour information, nous allons expliquer le fonctionnement de la table de réadressage, en utilisant l'exemple du programme PARAM.EXE du chapitre V.

Le listing en hexadécimal ci-dessous représente l'en-tête du programme PARAM.EXE, le début de la table de réadressage et le début du code exécutable. Les informations qui nous intéressent ont été soulignées :

- 58 00 représente le nombre d'éléments de la table de réadressage (dans ce cas 0058h = 88 éléments) ;
- 18 00 sert à déterminer l'adresse où commence le code exécutable, à l'intérieur du fichier *.EXE*, car la valeur représente, en nombre de paragraphes de 16 octets, la taille de l'en-tête et de la table de réadressage ; dans ce cas la valeur est 0018h, ce qui signifie que le programme commence à l'adresse 180h du fichier *.EXE.* ;
- 1C 00 donne l'adresse de la table de réadressage à l'intérieur du fichier *.EXE*, dans ce cas 001Ch ;
- 86 00 00 00, octets placés à partir de l'adresse 001Ch, constituent donc le premier élément de la table de réadressage (0000:0086h) ;
- BD 00 représente 00BDh, valeur contenue à l'adresse qui correspond au premier élément de la table (car 0000:0086 + 0180h = 0206h).

```
0000  4D 5A 40 01 0D 00 58 00 18 00 26 04 26 A4 A2 01
0010  00 40 00 00 9C 03 00 00 1C 00 00 00 86 00 00 00

.......suite de la table de réadressage.......

0180  55 89 E5 C4 7E 06 26 C6 05 04 8A 46 05 30 E4 B9

.......suite du code exécutable..............

0200  00 0E 57 9A 02 03 BD 00 8D BE 00 FE 16 57 C4 7E
```

Après le chargement du fichier PARAM.EXE par l'utilitaire Debug (voir listing plus bas), nous allons examiner le contenu de cette même adresse 0086h :

- les adresse 86h et 87h contiennent maintenant les octets 49 et 25, c'est à dire la valeur 2549h ;
- le registre CS contient la valeur 248Ch, adresse réelle où a été chargé le programme ;
- nous pouvons vérifier que 2549h est la somme de 248Ch et de l'ancienne valeur (00BDh) contenue à cette adresse ;
- de la même manière, chaque mot pointé par chaque élément de la table de réadressage sera recalculé pour adapter le code exécutable et les adresses des données globales à l'adresse physique où a été chargé le programme.

```
-r
AX=0000  BX=0000  CX=1740  DX=0000  SP=4000  BP=0000  SI=0000  DI=0000
DS=247C  ES=247C  SS=262E  CS=248C  IP=039C   NV UP EI PL NZ NA PO NC

-u 82
248C:0082 57            PUSH    DI
248C:0083 9A02034925    CALL    2549:0302
```

Quand on exécute un programme quelconque :

- le code exécutable (seconde partie du fichier *.EXE*) est placé dans la mémoire,
- la *table de réadressage* est stockée dans une zone de travail,
- chaque élément de cette table est utilisé pour recalculer la véritable adresse de l'élément correspondant,
- alors seulement, l'exécution du programme peut commencer.

Pour l'édition de liens du Turbo-Pascal, la taille de cette *table de réadressage* ne peut dépasser 64 K octets, c'est à dire 16384 éléments ; cette limite peut être atteinte avec des applications qui comptent de nombreux modules et beaucoup de code exécutable.

7.2 Quelques conseils pour éviter ce problème.

Nous avons constaté que l'utilisation d'*overlays* permet de réduire de façon significative le nombre d'éléments de la *table de réadressage*. Nous avons rencontré un exemple où la transformation en *overlays* de 8 unités diminue la table de 10280 à 3461 éléments ! Grâce à l'utilisation d'*overlays*, une partie des réadressages sont transférés vers le fichier d'extension *.OVR*.

Un autre facteur important est le nombre d'appel à des procédures ou à des fonctions d'une unité vers une autre. Si plusieurs unités utilisent souvent une petite routine implémentée dans une autre unité (ce qui va entraîner un réadressage à chaque appel), il peut être intéressant de répéter cette petite routine dans la partie d'implémentation de chaque unité. Il est évident que cette méthode ne peut (et ne doit) être utilisée que dans des cas très spéciaux : il serait très désavantageux de la généraliser car elle est contraire à l'esprit des unités.

8 Cas extrême : la mémoire est réellement insuffisante pour exécuter le programme.

Si, en dépit de toutes les techniques recommandées ci-dessus, le programme ne réussit pas à s'exécuter par manque de mémoire, ou si, en cours d'exécution, le manque de place sur le tas rend inopérantes les fonctions qu'il devrait réaliser, il ne reste qu'une issue : diviser le programme en deux programmes différents.

Si elle parait simple, cette solution présente toutefois l'inconvénient de rompre l'unité de l'application, maintenue jusqu'à maintenant :

- par l'accès aux différentes fonction par un seul système de menus et sous-menus,
- par le partage, durant l'exécution, des informations stockées dans la partie des variables globales.

En général, il est souhaitable de maintenir cette unité, même si ce n'est qu'en apparence, pour l'utilisateur final. Ce qui nous conduit à suggérer une forme de réaliser la division du programme :

- répartir les fonctions en deux programmes, si possible de tailles voisines ;
- implémenter un troisième programme, le plus petit possible (généralement il gérera les menus, mais ce n'est pas obligatoire), capable d'exécuter, par l'instruction *Exec* du Turbo-Pascal, l'un ou l'autre des deux programmes définis ci-dessus, en fonction du choix sur le menu ;
- pour communiquer entre les deux programmes (éventuellement les trois), utiliser un fichier pour stocker en fin de chaque programme, les variables dont se sert l'autre programme qui, dès le début de son exécution, ira lire ces mêmes informations ; pour accélérer cet échange d'informations, il est conseillé de déclarer ces variables proches les unes des autres afin de les lire et les écrire en bloc (*BlockRead* et *BlockWrite*) ; les déclarations doivent bien entendu être identiques dans chaque programme, ce qu'il est plus facile de garantir en utilisant une unité capable de déclarer, lire et écrire ces variables ;
- pour permettre l'exécution des deux autres programmes, le gestionnaire de menus doit occuper le moins d'espace possible : on utilisera pour cela la directive $M afin de définir une pile de petite taille (le minimum de 1024 octets doit être suffisant, à moins que la gestion des menus ne nécessite beaucoup de paramètres) et limiter la valeur maximum du tas (la valeur idéale est 0 et peut être utilisée si aucune allocation de variable dynamique n'est nécessaire) ; exemple : {$M 1024, 0, 0}.

Chapitre VII

Graphiques-unité Graph

L'unité **GRAPH**, disponible dans le Turbo-Pascal, permet la réalisation de graphiques sur l'écran. Dans un premier chapitre, nous allons expliquer certaines notions et énumérer les constantes pré-définies correspondantes dans l'unité **GRAPH**. Le second chapitre ne fait que résumer les routines disponibles dans l'unité **GRAPH** et le troisième présente un exemple qui servira également de base aux exercices du chapitre 4.

1 Terminologie, notions de base.

1.1 Equipement.

Le code exécutable qui trace des graphiques en Turbo-Pascal pourra être exécuté sur différents types de terminaux : écran CGA, EGA, VGA, Hercules, AT&T 400, PC3270 et IBM8514/A. Ceci grâce à l'utilisation, en temps d'exécution, de *pilotes* appropriés, nommés fichiers *.BGI* par Borland (Borland Graphics Interface).

La routine **InitGraph** permet de détecter automatiquement quel est le type de terminal et quel est la meilleure résolution possible sur ce terminal. Ce qui permet d'identifier quel est le fichier d'extension *.BGI* correspondant. Celui-ci sera alors chargé sur le tas. L'en-tête de la routine est :

```
procedure InitGraph(var Pilote : integer; var Mode : integer; Chemin : string);
```

où, le plus fréquemment, **Pilote** doit être initialisé à la valeur **Detect** (constante de valeur nulle) et **Chemin** doit contenir le nom du répertoire où se trouvent les fichiers *.BGI*. Après l'exécution de la routine, **Driver** et **Mode** reviennent avec les valeurs adaptées au type de matériel utilisé. Dans certains cas, l'utilisateur peut forcer le type de pilote et le mode graphique, en initialisant les deux paramètres avant l'appel de la routine.

Dans tous les cas, le fichier correspondant d'extension *.BGI* doit être accessible dans le répertoire **Chemin** ou dans le répertoire courant si **Chemin** est une chaîne vide.

La procédure **CloseGraph**, sans paramètre, permet de sortir du mode graphique et retourner au mode antérieur (généralement mode texte). Elle retire le *pilote* de la mémoire.

La procédure **RestoreCrtMode** permet de revenir au mode texte sans retirer le *pilote* de la mémoire.

La procédure :

```
procedure SetGraphMode(Mode : integer);
```

permet de passer du mode texte au mode graphique passé en paramètre.

Les pilotes et les modes suivants sont pré-définis dans l'unité **GRAPH** :

```
const
  {pilotes}
  CGA     = 1;                    IBM8514  = 6;
  MCGA    = 2;                    HercMono = 7;
```

```
  EGA     = 3;                          ATT400   = 8;
  EGA64   = 4;                          VGA      = 9;
  EGAMono = 5;                          PC3270   = 10;

  {modes graphiques pour le pilote CGA}
  CGAC1     = 0; {320x200; palette 1 : rouge, jaune, vert; 1 page}
  CGAC2     = 1; {320x200; palette 2 : cyan, magenta, blanc; 1 page}
  CGAHi     = 2; {640x200; monochrome; 1 page}
  {modes graphiques pour le pilote MCGA}
  MCGAC1    = 0; {320x200; palette 1 : rouge, jaune, vert; 1 page}
  MCGAC2    = 1; {320x200; palette 2 : cyan, magenta, blanc; 1 page}
  MCGAMed   = 2; {640x200; monochrome; 1 page}
  MCGAHi    = 3; {640x480; 2 couleurs; 1 page}
  {modes graphiques pour le pilote EGA}
  EGALo     = 0; {640x200; monochrome; 4 pages}
  EGAHi     = 1; {640x350; 16 couleurs; 2 pages}
  {modes graphiques pour le pilote EGA 64 Ko}
  EGA64Lo   = 0; {640x200; 16 couleurs; 1 page}
  EGA64Hi   = 1; {640x350; 4 couleurs; 1 page}
  EGAMonoHi = 3; {640x350; 1 page ou 4 pages}
  {modes graphiques pour le pilote Hercules}
  HercMonoHi= 0; {720x348; monochrome; 2 pages}
  {modes graphiques pour le pilote ATT400}
  ATT400C1  = 0; {320x200; palette 1 : rouge, jaune, vert}
  ATT400C2  = 1; {320x200; palette 2 : cyan, magenta, blanc}
  ATT400Med = 2; {640x200; 1 page}
  ATT400Hi  = 3; {640x400; 1 page}
  {modes graphiques pour le pilote VGA}
  VGALo     = 0; {640x200; 16 couleurs; 4 pages}
  VGAMed    = 1; {640x350; 16 couleurs; 2 pages}
  VGAHi     = 2; {640x480; 16 couleurs; 1 page}
  VGAHi2    = 3; {640x480; 256 couleurs; 1 page}
  {mode graphique pour le pilote PC3270}
  PC3270Hi  = 0; {720x350; 1 page}
  {modes graphiques pour le pilote IBM8514}
  IBM8514Lo = 0; {640x480; 256 couleurs; 1 page}
  IBM8514Hi = 1; {1024x768; 256 couleurs; 1 page}
```

1.2 Couleurs.

En fonction de l'équipement et du mode graphique en usage, on peut utiliser, sur le même graphique, une ou plusieurs couleurs. Turbo-Pascal permet d'utiliser jusqu'à 16 couleurs. Des routines permettent de choisir la couleur du fond de l'écran, la couleur des lignes, des textes, etc...

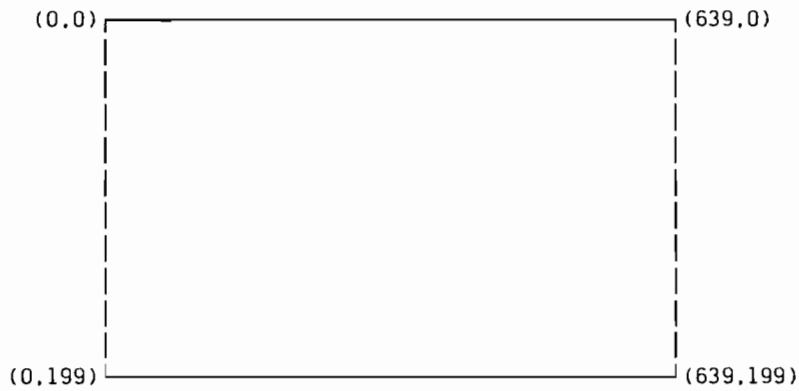
L'ensemble des couleurs utilisables en même temps est appelé *palette*. Les constantes et les types suivants sont pré-définis dans l'unité **GRAPH** :

```
const
  Black     = 0; {noir}                 LightBlue    =  9; {bleu clair}
  Blue      = 1; {bleu}                 LightGreen   = 10; {vert clair}
  Green     = 2; {vert}                 LightCyan    = 11; {cyan clair}
  Cyan      = 3; {cyan}                 LightRed     = 12; {rouge clair}
  Red       = 4; {rouge}                LightMagenta = 13; {magenta clair}
  Magenta   = 5; {magenta}              Yellow       = 14; {jaune}
  Brown     = 6; {marron}               White        = 15; {blanc}
  LightGray = 7; {gris clair}
  DarkGray  = 8; {gris foncé}           MaxColors    = 15; {nombre maximum de couleurs}
```

```
type
  PaletteType = record
                  Size   : byte; {nombre de couleurs de la palette}
                  Colors : array[0..MaxColors] of shortint; {tableau des couleurs}
                end;
```

1.3 Système de coordonnées.

Pour n'importe lequel des terminaux énumérés plus haut, on attribue au coin supérieur gauche les coordonnées (0,0). Les abscisses croissent de la gauche vers la droite, les ordonnées croissent du haut vers le bas. Le nombre de lignes ou de colonnes, en mode graphique, dépend du matériel et de la résolution utilisée. Dans le cas du mode **CGA-haute résolution** par exemple, il existe 640 colonnes, numérotées de 0 à 639, et 200 lignes numérotées de 0 à 199, ce que l'on a coutume de résumer comme étant une résolution 640x200. Le mode **Hercules** a une résolution de 720x348. L'intersection de chaque ligne avec chaque colonne est appelée point ou *pixel* (abréviation de l'anglais *picture element*). En mode **CGA-haute résolution**, il existe 128 000 *pixels* (640 fois 200). Le schéma ci-dessous montre les coordonnées des quatres coins d'un écran **CGA-haute résolution**.



On utilise également la notion de **pointeur courant** qui pointe vers la position d'un curseur graphique, semblable à un curseur en mode texte, mais invisible. Le **pointeur courant** peut se représenter par un couple (x, y) de coordonnées. Plusieurs procédures et fonctions liées au mode graphique manipulent le **pointeur courant** : par exemple MoveTo(159, 99) place le curseur au milieu d'un écran CGA en mode haute-résolution.

1.4 Textes.

Deux types de polices de caractères sont disponibles pour écrire des textes sur un écran en mode graphique :

- une police de caractères dessinés avec une matrice de 8x8 pixels ;
- plusieurs polices de caractères vectorielles.

Le premier est suffisant pour écrire des caractères de petite taille. Les autres devront être préférés pour écrire des caractères de taille plus grande car, même une fois agrandi, le caractère garde une bonne résolution, ce qui n'est pas le cas avec les matrices de pixels. On pourra exécuter le programme ORSTOM.PAS, disponible sur la disquette, pour le vérifier.

Les fichiers d'extension *.CHR* contiennent les polices vectorielles et doivent être accessibles pendant l'exécution du programme.

Les routines **SetTextJustify**, **SetUserCharSize**, **SetTextStyle**, **OutText**, **OutTextXY** et **GetTextSettings** manipulent les textes et les caractéristiques des caractères.

Les objets suivants, liés à la manipulation de textes, sont pré-définis :

```
const
  {polices de caractéres}
  DefaultFont   = 0; {police matricielle}
  TriplexFont   = 1; {police vectorielle (fichier TRIP.CHR)}
  SmallFont     = 2; {police vectorielle (fichier LITT.CHR)}
  SansSerifFont = 3; {police vectorielle (fichier SANS.CHR)}
  GothicFont    = 4; {police vectorielle (fichier GOTH.CHR)}
  {sens pour l'affichage des textes}
  HorizDir = 0; {sens horizontal}
  VertDir  = 1; {sens vertical}
  {taille des caractères}
  NormSize = 1; {taille normale}
  {type de justification horizontale}
  LeftText   = 0; {justification à gauche}
  CenterText = 1; {texte centré}
  RightText  = 2; {justification à droite}
  {type de justification verticale}
  BottomText = 0; {justification en dessous du point}
  CenterText = 1; {texte centré}
  TopText    = 2; {justification au dessus du point}

type
  {type de caractéristiques de texte}
  TextSettingsType = record
                       Font,                {police}
                       Direction : word;    {sens}
                       CharSize  : word;    {taille}
                       Horiz,               {justification horizontale}
                       Vert      : word;    {justification verticale}
                     end;
```

1.5 Types de tracés.

De nombreuses routines peuvent être utilisées pour tracer des lignes, des cercles, des ellipses, des diagrammes de barre, des "camemberts", des polygones et les remplir avec des hachures ou d'autres motifs.

Les types de lignes ou de hachures peuvent être choisis ou même définis par le programme. On utilise pour cela les routines **SetLineStyle**, **SetFillStyle** et **SetFloodPattern** avant d'appeler les routines qui remplissent les zones (**FillPoly** et **FloodFill**).

Les objets suivants, liés aux types de tracés, sont pré-définis :

```
const
  {types de lignes}
  SolidLn   = 0; {trait continu}
  DottedLn  = 1; {trait pointillé}
  CenterLn  = 2; {-.-.-}
  DashedLn  = 3; {tiretés}
  UserBitLn = 4; {défini par le programmeur}
  {épaisseur du trait}
  NormWidth = 1; {trait normal}
  ThickWidth= 3; {trait large}
```

```
type
  {type de style des traits}
  LineSettingsType = record
                       LineStyle : word; {type de trait}
                       Pattern   : word; {définition du tracé}
                       Thickness : word; {épaisseur du trait}
                     end;
  {type qui définit un arc d'ellipse}
  ArcCoordsType = record
                    x, y          : integer; {centre de l'ellipse}
                    XStart, YStart: integer; {début de l'arc}
                    XEnd, YEnd    : integer; {fin de l'arc}
                  end;
```

Les objets ci-dessous sont liés au remplissage des zones :

```
const
  {motif de remplissage}
  EmptyFill      =  0; {remplit avec la couleur du fond de l'écran}
  SolidFill      =  1; {couleur unie}
  LineFill       =  2; {tiretets horizontaux}
  LtSlashFill    =  3; {/ / / / avec un trait fin}
  SlashFill      =  4; {/ / / / avec un trait épais}
  BkSlashFill    =  5; {\ \ \ \ avec un trait épais}
  LtBkSlashFill  =  6; {\ \ \ \ avec un trait fin}
  HatchFill      =  7; {hachures fines}
  XHatchFill     =  8; {hachures épaisses}
  InterLeaveFill =  9; {hachures croisées}
  WideDotFill    = 10; {pointillé espacé}
  CloseDotFill   = 11; {pointillé peu espacé}
  UserFill       = 12; {défini par le programmeur}

  {constantes pour Bar3D (tracé de graphique de barres en trois dimensions)}
  TopOn  = true;  {dessine la partie supérieure de la barre}
  TopOff = false; {ne dessine pas la partie supérieure de la barre}

type
  FillSettingsType = record
                       Pattern : word; {type de remplissage}
                       Color   : word; {couleur}
                     end;
  {type qui définit un type de remplissage}
  FillPatternType = array[1..8] of byte;
```

1.6 Fenêtre active, pagination.

Semblable à la routine Window utilisée en mode texte, la procédure **SetViewPort** permet de définir une partie de l'écran comme fenêtre courante en mode graphique. Toutes les opérations graphiques réalisées ensuite le seront avec des coordonnées relatives à cette fenêtre.

Le type pré-défini ViewPortStyle définit une fenêtre graphique :

```
type
  ViewPortType = record
                   x1, y1, x2, y2 : integer; {coins de la fenêtre}
                   Clip           : boolean; {contrôle des limites}
                 end;
```

```
const
  {constantes de contrôle des limites de la fenêtre}
  ClipOn  = true;   {le tracé ne peut dépasser les limites de la fenêtre}
  ClipOff = false;  {le tracé peut dépasser les limites de la fenêtre}
```

Divers modes graphiques permettent de travailler avec plusieurs **pages graphiques**, une de ces pages (la page active) étant visible à l'écran alors que les autres sont stockées en mémoire. Des routines permettent de manipuler ces pages, pour réaliser une animation par exemple.

1.7 Manipulation de zones de pixels.

GetPixel et **PutPixel** peuvent être utilisées pour manipuler un *pixel*. **GetImage** et **PutImage** permettent de manipuler une zone rectangulaire de l'écran, par exemple pour la stocker en mémoire ou dans un fichier et la restituer plus tard à un autre endroit de l'écran, éventuellement après une opération logique.

Les opérateurs logiques suivants peuvent être utilisés au moment de restituer une image sur l'écran :

```
const
  NormalPut = 0; {utilise l'instruction MOV de l'assembleur}
  XORPut    = 1; {utilise l'instruction XOR de l'assembleur}
  ORPut     = 2; {utilise l'instruction OR de l'assembleur}
  ANDPut    = 3; {utilise l'instruction AND de l'assembleur}
  NOTPut    = 4; {utilise l'instruction NOT de l'assembleur}
```

1.8 Code de retour.

La fonction **GraphResult** donne la valeur du code de retour de beaucoup de routines de l'unité **GRAPH**. Elle fonctionne comme **IOResult**, étant remise à zéro après chaque appel. Voici la liste des codes de retour possibles (ce sont des constantes, déjà définies dans l'unité **GRAPH**, qui peuvent être utilisées à l'intérieur des programmes).

```
grOk             =   0 {pas d'erreur}
grNoInitGraph    =  -1 {erreur à l'initialisation}
grNoDetected     =  -2 {ne peut détecter le mode graphique}
grFileNotFound   =  -3 {pilote (.BGI) non trouvé}
grInvalidDriver  =  -4 {pilote non autorisé}
grNoLoadMem      =  -5 {mémoire insuffisante}
grNoScanMem      =  -6 {mémoire insuffisante}
grNoFloodMem     =  -7 {mémoire insuffisante}
grFontNotFound   =  -8 {fichier .CHR non trouvé}
grNoFontMem      =  -9 {mémoire insuffisante}
grInvalidMode    = -10 {mode graphique non autorisé}
grError          = -11 {autre erreur de graphique}
grIOError        = -12 {erreur d'entrée/sortie}
grInvalidFont    = -13 {format incorrect du fichier .CHR}
grInvalidFontNum = -14 {numéro invalide de police de caractère}
```

2 Description de l'unité GRAPH.

L'unité **GRAPH** met à la disposition du programmeur environ 80 procédures et fonctions, en plus de constantes et de types pré-définis. Il serait fastidieux de décrire chacune de ces routines dans ce support de cours. Nous donnerons seulement ci-dessous la liste de ces routines avec leur en-tête et leur rôle, mais sans détailler les valeurs à passer comme paramètres, ni les valeurs retournées. Pour une description plus complète, le lecteur pourra se reporter au manuel de Turbo-Pascal, ou consulter l'aide en ligne disponible pour chaque routine.

```
procedure Arc(x, y : integer; AngleDebut, AngleFin, Rayon : word);
      Trace un arc de cercle.
procedure Bar(x1, y1, x2, y2 : integer);
      Trace une barre rectangulaire en deux dimensions dans le style et la couleur du remplissage
      courants.
procedure Bar3D(x1, y1, x2, y2 : integer; Profondeur : word; Dessus : boolean);
      Trace une barre en trois dimensions dans le style et la couleur du remplissage courants.
procedure Circle(x, y : integer; Rayon : word);
      Trace un cercle.
procedure ClearDevice;
      Efface l'écran en mode graphique.
procedure ClearViewPort;
      Efface seulement la fenêtre graphique active.
procedure CloseGraph;
      Désactive le mode graphique et retourne au mode initial.
procedure DetectGraph(var Pilote, Mode : integer);
      Detecte la configuration graphique utilisée (pilote et mode).
procedure DrawPoly(NombPoints : word; var TableauPoints);
      Trace un polygone de NombPoints angles.
procedure Ellipse(x, y : integer; AngleDebut, AngleFin : word; XRayon, YRayon : word);
      Trace un arc d'ellipse.
procedure FillEllipse(x, y : integer; XRayon, YRayon : word);
      Trace une ellipse et la remplit.
procedure FillPoly(NombPoints : word; var TableauPoints);
      Trace un polygone et le remplit.
procedure FloodFill(x, y : integer; CouleurBordure : word);
      Remplit une zone limitée par CouleurBordure.
procedure GetArcCoords(var Coordonnees : ArcCoordsType);
      Retourne les coordonnées utilisées pour tracer l'arc précédent (de cercle ou d'ellipse).
procedure GetAspectRatio(var LargeurPixels, HauteurPixels : word).
      Retourne la résolution de l'écran graphique.
function GetBkColor : word;
      Retourne le numéro de la couleur du fond de l'écran.
function GetColor : word;
      Retourne le numéro de la couleur courante.
procedure GetDefaultPalette(var Palette : PaletteType);
      Retourne la palette défaut définie lors de l'initialisation du mode graphique.
function GetDriverName : string;
      Retourne le nom du pilote courant.
procedure GetFillPattern(var TypeRemplissage : FillPatternType);
      Retourne le dernier type de tracé utilisé par SetFillPattern.
procedure GetFillSettings(var InfoRemplissage : FillSettingsType);
      Retourne la couleur et le motif utilisés par SetFillStyle et SetFillPattern.
function GetGraphMode : integer;
      Retourne le mode graphique courant.
procedure GetImage(x1, y1, x2, y2 : integer; var TableauBit);
      Mémorise l'image graphique d'une zone rectangulaire de l'écran.
```

```
procedure GetLineSettings(var InfoLigne : LineSettingsType);
      Retourne le type et l'épaisseur des lignes, definis par SetLineSettings.
function GetMaxColor : word;
      Retourne la plus grande valeur possible de la couleur pour le mode graphique actif.
function GetMaxMode : integer;
      Retourne la plus grande valeur possible du mode graphique pour le pilote utilisé.
function GetMaxX : integer;
      Retourne le nombre de pixels sur la dimension horizontale pour le mode graphique actif.
function GetMaxY : integer;
      Retourne le nombre de pixels sur la dimension verticale pour le mode graphique actif.
function GetModeName(Mode : integer) : string;
      Retourne le nom du mode graphique passé en paramètre.
procedure GetModeRange(Pilote : integer; var ModeMini, ModeMaxi : integer);
      Retourne les valeurs minimum et maximum des modes autorisés pour le pilote.
procedure GetPalette(var Palette : PaletteType);
      Retourne la palette de couleurs en cours d'utilisation.
function GetPaletteSize : integer;
      Retourne seulement le nombre de couleurs de la palette en cours d'utilisation.
function GetPixel(x, y : integer) : word;
      Retourne la couleur du pixel (x y).
procedure GetTextSettings(var InfoTexte : TextSettingsType);
      Retourne les caractéristiques du mode d'écriture courant des textes.
procedure GetViewSettings(var InfoFenetre : ViewPortType);
      Retourne les caractéristiques de la fenêtre graphique active.
function GetX : integer;
      Retourne l'abscisse, dans la fenêtre active, du curseur graphique.
function GetY : integer;
      Retourne l'ordonnée, dans la fenêtre active, du curseur graphique.
procedure GraphDefaults;
      Réinitialise aux valeurs défauts définies par InitGraph.
function GraphErrorMsg(CodeErreur : integer) : string;
      Retourne le texte du message qui correspond à l'erreur passée en paramètre.
function GraphResult : integer;
      Retourne la valeur du dernier code d'erreur émis par le système graphique.
function ImageSize(x1, y1, x2, y2 : integer) : word;
      Retourne le nombre d'octets nécessaires pour mémoriser l'image graphique de la zone
      rectangulaire définie par les coordonnées passées en paramètres.
procedure InitGraph(var Pilote : integer; var Mode : integer; Path : string);
      Initialise le système graphique pour l'équipement détecté automatiquement ou passé comme
      paramètre.
function InstallUserDriver(Nom : string; PtrDetecte : pointer) : integer;
      Permet de définir un nouveau pilote.
function InstallUserFont(NomPolice : string) : integer;
      Permet d'installer une nouvelle police vectorielle de caractères.
procedure Line(x1, y1, x2, y2 : integer);
      Trace une ligne entre deux points.
procedure LineRel(Dx, Dy: integer);
      Trace une ligne entre le point courant et un autre point (coordonnées relatives).
procedure LineTo(x, y : integer);
      Trace une ligne entre le point courant et un autre point (coordonnées absolues).
procedure MoveRel(Dx, Dy : integer);
      Déplace le point courant (curseur graphique) vers une autre position (coordonnées relatives).
procedure MoveTo(x, y : integer);
      Déplace le point courant (curseur graphique) vers une autre position (coordonnées absolues).
procedure OutText(Texte : string);
      Affiche un texte à la position du curseur graphique.
procedure OutTextXY(x, y : integer; Texte : string);
      Affiche un texte à la position (x, y).
```

```
procedure PieSlice(x, y : integer; AngleDebut, AngleFin, Rayon : word);
      Trace et remplit une portion de graphique de type "camembert".
procedure PutImage(x, y : integer; var TableauBit; Logique : word);
      Affiche à l'écran une image graphique préalablement mémorisée par GetImage.
procedure PutPixel(x, y : integer; CouleurPixel : word);
      Trace un pixel d'une certaine couleur à la position (x, y).
procedure Rectangle(x1, y1, x2, y2 : integer);
      Trace un rectangle.
function RegisterBGIDriver(Pilote : pointer) : integer;
      Mémorise un pilote chargé par le programme ou par l'édition de liens.
function RegisterBGIFont(Police : pointer) : integer;
      Mémorise une police de caractères chargée par le programme ou par l'édition de liens.
procedure RestoreCrtMode;
      Restitue le mode texte utilisé avant l'activation du mode graphique.
procedure Sector(x, y : integer; AngleDebut, AngleFin, XRayon, YRayon : word);
      Trace un secteur d'ellipse et le remplit.
procedure SetActivePage(Page : word);
      Définit quelle sera la page active.
procedure SetAllPalette(var Palette);
      Change les couleurs de la palette.
procedure SetAspectRatio(x, y : word);
      Change le rapport de correction des dimensions de l'écran.
procedure SetBkColor(Couleur : word);
      Définit la couleur du fond de l'écran.
procedure SetColor(Couleur : word);
      Définit la couleur à utiliser comme couleur courante.
procedure SetFillPattern(Motif : FillPatternType; Couleur : word);
      Définit un motif de remplissage et l'indique comme étant celui à utiliser lors des prochaines
      opérations de remplissage.
procedure SetFillStyle(Motif, Couleur : word);
      Sélectionne le motif et la couleur à utiliser lors des prochaines opérations de remplissage
procedure SetGraphBufSize(Taille : word);
      Change la taille du tampon utilisé par les opérations de remplissage (défaut : 4Ko).
procedure SetGraphMode(Mode : integer);
      Active le mode graphique et efface l'écran.
procedure SetLineStyle(Style, Trace, Epaisseur : word);
      Définit les caractéristiques des traits à tracer.
procedure SetPalette(NumeroCouleur : word; Couleur : shortint);
      Change une couleur dans la palette.
procedure SetRGBPalette(NumeroCouleur : word; Rouge, Vert, Bleu : integer);
      Change la palette de couleur (seulement pour les pilotes VGA et IBM 8514).
procedure SetTextJustify(Horizontal, Vertical : word);
      Définit le type de justification à utiliser par OutText et OutTextXY.
procedure SetTextStyle(Police, Sens, Taille : word);
      Définit le type et la taille des caractères des textes et le sens de l'affichage (horizontal
      ou vertical).
procedure SetUserCharSize(MultX, DivX, MultY, DivY : word);
      Permet de modifier la hauteur et la largeur des caractères des polices vectorielles.
procedure SetViewPort(x1, y1, x2, y2 : word; Bordure : boolean);
      Définit la nouvelle fenêtre graphique active.
procedure SetVisualPage(Page : word);
      Définit la page active.
procedure SetWriteMode(ModeTrace : integer);
      Définit le mode de tracé des lignes (effacer ou superposer avec le dessin antérieur).
function TextHeight(Texte : string) : word;
      Retourne la hauteur en pixels du texte passé comme paramètre.
function TextWidth(Texte : string) : word;
      Retourne la longueur en pixels du texte passé comme paramètre.
```

3 Un exemple de diagramme de barres.

L'exemple présenté ci-dessous réalise le tracé d'un diagramme de barres du total pluviométrique mensuel. Il utilise des routines, disponibles dans l'unité **GRAFCLM1**, pour avoir accès aux données d'un fichier annuel de climatologie. Voir le chapitre 4 pour une description plus détaillée de l'unité **GRAFCLM1**.

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{                          exemple de diagramme de barres                    }
{                     (disponible dans le fichier grafclm0.pas)              }
{----------------------------------------------------------------------------}
{$V-}
uses
  crt,
  graph,
  grafclm1;

const
  NomMois : array[1..12] of string[3] =
            ('Jan', 'Fev', 'Mar', 'Avr', 'Mai', 'Jun',
             'Jul', 'Aou', 'Sep', 'Oct', 'Nov', 'Dec');
var
  TabMens : TipoTabMens;
  Station   : string[13];
  Annee     : word;
  Mois, Jour: byte;
  HautBarre, BarGauche : real;
  xe, xd, PixAxeX, PixAxeY,
  PixMaxiX, PixMaxiY          : word;
  PixDeltaY, ValDelta, PixDeltaX,
  MaxiMois, t, v, y, w          : real;
  Total : array[1..12] of real;
  Pilote, Mode, i, NombGrad : integer;
  s : string;

function ElevPuis(x, y : real) : real;
{élévation de x à la puissance y}
begin
  ElevPuis := exp(y*ln(x));
end;

begin
  if FichierOuvert(Station, Annee)
    then begin

      {calcul des totaux mensuels}
      for Mois := 1 to 12 do
        begin
          MatriceMois('PP', Mois, TabMens);
          t := 0;
          for Jour := 1 to JourMois(Annee, Mois) do
            if TabMens[Jour] <> -MaxInt
              then t := t + TabMens[Jour];
          Total[Mois] := t;
        end;
```

```
{calcul du plus grand total mensuel}
MaxiMois := 0;
for Mois := 1 to 12 do
  if Total[Mois] > MaxiMois
    then MaxiMois := Total[Mois];

{initialisation du mode graphique}
Pilote := Detect;
InitGraph(Pilote, Mode, '');

{tracé des axes}
PixAxeX := round(0.9 * GetMaxY);
PixAxeY := round(0.1 * GetMaxX);
Line(PixAxeY, 0, PixAxeY, PixAxeX);
Line(PixAxeY, PixAxeX, GetMaxX, PixAxeX);

{calcul de l'échelle en y et du nombre de graduations}
PixMaxiY := round(0.8 * GetMaxY);
w := trunc(ln(MaxiMois)/ln(10));        {nombre de chiffres de MaxiMois}
i := 0;
repeat
  inc(i)
until (i*ElevPuis(10, w) > MaxiMois);
MaxiMois := i*ElevPuis(10,w);          {multiple de 10 plus grand que MaxiMois}
NombGrad := succ(i);                   {nombre de graduations}
PixDeltaY := PixMaxiY / pred(NombGrad);{intervalle en pixels entre deux graduations}
ValDelta := ElevPuis(10, w);        {intervalle de précipitation entre deux graduations}
if NombGrad < 4
  then begin                        {si peu de graduations, diviser l'intervalle par 2}
    NombGrad := 2*NombGrad;
    PixDeltaY := PixDeltaY/2;
    ValDelta := ValDelta/2;
  end;

{dessin des graduations verticales}
SetTextStyle(DefaultFont, VertDir, 1);
SetTextJustify(LeftText, CenterText);
v := 0;
for i := 0 to NombGrad do
  begin
    y := PixAxeX-round(i*PixDeltaY);    {ordonnée de la graduation}
    Line(round(PixAxeY), round(y),
         round(PixAxeY*0.8), round(y)); {trait marquant la graduation}
    {afficher la valeur correspondant à la graduation}
    str(v:succ(round(w)):0, s);
    OutTextXY(round(0.8*PixAxeY), round(y), s);
    v := v + ValDelta;
  end;

{affichage de la légende verticale}
SetTextJustify(RightText, TopText);
s := 'mm';
OutTextXY(succ(PixAxeY+TextHeight(s)), 0, s);
```

```
      {tracé des barres}
      PixMaxiX := round(0.9 * (GetMaxX - PixAxeY)); {utiliser les 9/10 de l'axe}
      PixDeltaX := PixMaxiX / 12.0;                 {intervalle entre la fin d'une barre
                                                     et le début de la suivante}

      BarGauche := PixAxeY +                        {début de la première barre}
                (0.05 * (GetMaxX - PixAxeY));
      SetFillStyle(HatchFill, 6);                   {style des hachures des barres}
      SetColor(10);                                 {couleur des lignes}
      SetTextStyle(DefaultFont, HorizDir, 1);       {style du nom des mois}
      SetTextJustify(CenterText, CenterText);
      for Mois := 1 to 12 do
        begin
          HautBarre := Total[Mois]*PixMaxiY/MaxiMois; {hauteur de la barre}
          xe := round(BarGauche);
          xd := round(BarGauche+PixDeltaX*0.7);    {largeur de la barre = 70% de l'intervalle}
          Bar3D(xe, PixAxeX,                       {coin inférieur gauche de la barre}
                xd, round(PixAxeX - HautBarre),    {coin supérieur droit de la barre}
                (xd - xe + 1) div 4, TopOn);       {profondeur des barres}
          OutTextXY(round(BarGauche+0.7*PixDeltaX/2), {affichage du nom du mois}
                    PixAxeX+((GetMaxY - PixAxeX) div 2),
                    NomMois[Mois]);
          BarGauche := BarGauche + PixDeltaX;        {début de la barre suivante}
        end;

      {affichage du titre}
      str(Annee, s);
      s := 'Pluie mensuelle - station ' + Station + ' - Annee ' + s;
      SetTextStyle(SansSerifFont, HorizDir, 1);
      SetUserCharSize(1, 2, 1, 2);
      SetTextJustify(CenterText, CenterText);
      OutTextXY(GetMaxX div 2, round(GetMaxY*0.05), s);
    end;
  while not keypressed do;
  RestoreCrtMode;
end.
```

4 Exercices.

Les trois exercices ont pour but de compléter et généraliser l'exemple présenté au chapitre 3. Le programme **GRAFCLM.PAS** disponible sur la disquette présente un menu avec trois possibilités :

- graphique annuel d'une variable quelconque du fichier de climatologie ;
- graphique de la température maximum et de la température minimum sur le même diagramme ;
- diagramme de barres de la pluie mensuelle.

Le travail à réaliser est le suivant :

a) Implémenter une routine GraphiqueVariable, en profitant de l'exemple du chapitre 3, en particulier pour déterminer les axes, les légendes, etc.. Pour cela, il sera nécessaire de transformer en routines diverses parties de l'exemple, en les paramétrant pour les rendre utilisables par les trois parties du programme.

b) Implémenter la routine GraphiquePluie, en utilisant l'exemple du chapitre 3 et les routines de l'exercice a).

c) Implémenter la routine GraphiqueExtremes, en utilisant les routines de l'exemple a).

Pour accéder aux données, utiliser l'unité **GRAFCLM1** dont la partie d'interface est décrite ci-dessous :

```
type
  TypeTabMens = array[1..31] of real;  {tableau des valeurs mensuelles d'une variable
quelconque}
  TypeTabAnual = array[1..366] of real;{tableau des valeurs annuelles d'une variable
quelconque}

function JourMois(Annee, Mois : byte) : byte;
{retourne le nombre de jours dans un mois}

function FichierOuvert(var Station : string; var Annee : word) : boolean;
{ouvre le fichier de données climatologiques et retourne le nom de la station
 et l'année correspondantes au fichier}

function Efface(s : string) : string;
{efface les espaces de début et de fin d'une chaîne de caractères}

procedure FermerFichier;
{ferme le fichier de climatologie}

function NumeroVariable(v : string) : byte;
{retourne le numéro de la variable dans la liste des variables autorisées}

function NomVariable(n : byte) : string;
{retourne le nom de la variable identifiée par son numéro}

function UniteVariable(n : byte) : string;
{retourne l'unité de la variable}

function ValeurJour(Variable : string; Jour, Mois : byte) : real;
{retourne la valeur observée de la variable pour la date passée en paramètre}

procedure MatriceMois(Variable : string; Mois : byte; var TabMens : TypeTabMens);
{retourne le tableau mensuel des valeurs observées de la variable, pour le mois considéré}

procedure MatriceAnnee(Variable : string; var TabAnn : TypeTabAnn);
{retourne le tableau annuel des valeurs de la variable considérée}
```

Les noms usuels des variables autorisées sont :

- TX pour la température maximum,
- TN pour la température minimum,
- TM pour la température moyenne,
- UR pour l'humidité relative,
- PP pour la précipitation,
- EV pour l'évaporation,
- DE pour la durée d'ensoleillement,
- VV pour la vitesse du vent,
- DJ pour la durée théorique du jour.

Enfin, pour un type quelconque de variable, un jour sans observation retourne une valeur égale à -MaxInt.

Chapitre VIII

Compilation conditionnelle - Co-processeur arithmétique - Débogage - Son

1 Compilation conditionnelle.

La compilation conditionnelle, semblable à celle utilisée par les compilateurs assembleurs, permet de générer des fichiers exécutables différents à partir du même programme source. Elle est basée sur l'utilisation de *directives* pour définir ou tester des *variables de compilation*. Il s'agit des directives :

- {$DEFINE nom_variable}, pour définir une variable de compilation ;
- {$UNDEF nom_variable}, pour annuler une variable de compilation ;
- {$IFDEF nom_variable}, pour tester si la variable a été définie ;
- {$IFNDEF nom_variable}, pour tester si la variable n'a pas été définie ;
- {$IFOPT directive}, pour tester l'état d'une directive de compilation ;
- {$ELSE} et {$ENDIF}, pour compléter les directives de type {$IF...}.

Ces variables de compilation sont totalement indépendantes des variables du programme. Quelques unes sont déjà définies par le Turbo-Pascal :

- VER55 toujours définie dans la version 5.5 (VER40 pour la version 4.0, VER50 pour la version 5.0, etc...) ;
- MSDOS et CPU86, toujours définies dans les versions pour PC ;
- CPU87 définie quand le co-processeur arithmétique est installé sur l'ordinateur où est réalisée la compilation.

D'autres peuvent être définies par le programme. La séquence d'instructions suivantes permet de créer un programme exécutable pour afficher des messages d'erreurs en français. Il suffirait de retirer la ligne {$DEFINE Francais}, ou simplement introduire une espace entre { et $ pour afficher des messages en anglais.

```
{$DEFINE Francais}

procedure AfficherErreurStandard(n : byte);
var
  c : char;
begin
  gotoxy(1,25);
  case n of
{$IFDEF Francais}
  1 : write('Disque plein');
  2 : write('Unité de disquette non prête');
  3 : write('Mémoire insuffisante');
  {.....}
{$ELSE}
  1 : write('Disc full');
  2 : write('Drive not ready');
  3 : write('Insuficient memory');
  {.......}
```

```
{$ENDIF}
  end;
  c := readkey;
end;
```

D'autres utilisations de la compilation conditionnelle pourront être consultées dans le chapitre 2.

Une variable de compilation peut être définie soit en utilisant la directive $DEFINE, soit en utilisant la directive *Options/Compiler/Conditional Defines* dans l'environnement intégré, soit encore avec la directive /D du compilateur TPC.EXE (exemple : tpc prog /DFrancais).

2 Utilisation du co-processeur arithmétique.

Les processeurs de la famille 80x86 manipulent aisément les nombres entiers, c'est à dire les variables de type *byte*, *shortint*, *integer*, *word* et *longint*. Mais ils ne sont pas conçus pour manipuler les nombres réels comme, par exemple, les variables de type *real* en Turbo-Pascal. C'est pourquoi il existe en parallèle la famille des processeurs 80x87 appelés **co-processeurs arithmétiques**, capables d'exécuter très rapidement des opérations sur les nombres réels. La grande majorité des PC ou compatibles peuvent être équipés d'un co-processeur correspondant au processeur original (8087 pour un 8086, 80287 pour un 80286, etc...).

Le Turbo-Pascal a toujours permis l'utilisation de réels, en utilisant le 80x86 par l'intermédiaire des routines de sa bibliothèque. A partir de la version 4.0, il est devenu possible d'utiliser directement le co-processeur, en compilant les programmes avec la directive $N+, avec toutefois un inconvénient : un programme compilé pour utiliser le 80x87 ne pouvait pas fonctionner sur un micro-ordinateur non équipé du co-processeur. Heureusement, depuis la version 5.0, il est possible de compiler un programme de telle façon qu'il utilise le co-processeur, s'il est installé, ou qu'il l'émule dans le cas contraire. Dans ce dernier cas, il est évident que la rapidité du co-processeur ne pourra pas être exploitée, puisque les calculs seront faits par des routines utilisant le 80x86.

Sans utilisation du co-processeur, le seule type de variable réelle disponible est le type *real*. En utilisant le co-processeur, nous disposons des types *single* (équivalent du type *real*), *double*, *extended* et *comp*. Il faut savoir que le type *real* occupe 6 octets alors que le type *single* occupe 4 octets, ce qui implique qu'une variable réelle écrite dans un fichier créé par un programme qui n'utilise pas le 80x87 ne pourra pas être relue par le même programme (ou un autre) qui utilise le 80x87. Ce qui était une limitation de la version 4.0.

En pratique, depuis la version 5.0, si le système à développer comporte des variables réelles, on a tout intérêt à placer au début de chaque module les directives {$N+,E+} et utiliser seulement les types *single*, *double*, *extended* et *comp*. Cette méthode présente à la fois des avantages :

- une seule version du programme-source et du programme-exécutable, avec ou sans co-processeur ;
- les fichiers de réels créés sans le co-processeur pourront être utilisés avec, et inversement ;
- toutes les possibilités offertes par le co-processeur seront utilisées, lorsqu'il est installé ;

et des inconvénients :

- à cause de l'émulation, le fichier *.EXE* sera légèrement plus important, puisqu'il contiendra les routines d'émulation ;

- en l'absence du co-processeur, les calculs qui utilisent des variables de type *double*, *extended* ou *comp* pourront être plus lents que s'ils étaient réalisés avec des variables de type *real* ; en contrepartie, ils seront aussi plus précis.

Le tableau ci-dessous regroupe les temps d'exécution du programme P8087, sur différentes configurations du même ordinateur :

directives	$N-	$N+,E-	$N+,E+
avec 80x87	7"25	2"91	2"91
sans 80x87	7"25	ne fonctionne pas	8"25

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{                    test de vitesse avec ou sans co-processeur              }
{                      (disponible dans le fichier p8087.pas)                }
{----------------------------------------------------------------------------}
{$N+,E+}
uses dos;
{$IFOPT N+}
type
  real = double;
{$ENDIF}
var
  A : real;
{$IFDEF N+}
  X, Y : extended;
{$ELSE}
  X, Y : real;
{$ENDIF}
  I    : word;
  t1,
  t2 : longint;
  h, m, s, c : word;
begin
  A := Pi;
  GetTime(h, m, s, c);
  t1 := (longint(h)*3600+m*60+s)*100 + c;
  for I := 1 to 1000
    do begin
      X := exp(sin(A/2)) + exp(cos(A/2));
      Y := exp(sin(A/2)) - exp(cos(A/2));
      A := A/2;
    end;
  GetTime(h, m, s, c);
  t2 := (longint(h)*3600+m*60+s)*100 + c;
  writeln;
  writeln(' Tempo total : ', (t2-t1) div 100, '"', (t2-t1) mod 100);
end.
```

Dans le programme-source présenté ci-dessus, on peut remarquer que des directives de compilation sont utilisées pour créer plusieurs versions du même programme, en modifiant seulement le contenu de la première directive {$N+,E+}. Pour qu'un ordinateur équipé d'un co-processeur fonctionne sans, il n'est pas nécessaire de retirer ce dernier : il suffit d'utiliser

la commande *SET* du **DOS** pour modifier l'environnement :

```
SET 87=N permet de désactiver le 80x87
SET 87=Y permet de la réactiver
```

Ceci peut servir pour vérifier qu'un programme développé sur un ordinateur équipé de co-processeur fonctionne également sans co-processeur.

Enfin, la variable **Test8087** déclarée dans l'unité **SYSTEM** peut être testée à l'intérieur d'un programme pour connaître la configuration. Elle peut prendre les valeurs 0, 1, 2, 3 pour indiquer qu'il n'y a pas de co-processeur ou qu'il y a un 8087, un 80287 ou un 80387, respectivement.

3 Débogage - Utilisation du débogueur.

Nous allons nous intéresser, dans ce chapitre, aux erreurs qui se produisent lors de l'exécution d'un programme. Nous les classerons en erreurs de deux types : celles qui sont accompagnées d'un message d'erreur (voir annexe) et celles qui provoquent un arrêt du programme ou du système, sans message ou avec un message sans signification.

3.1 Erreurs d'exécution avec message.

Elles sont visualisées par un message du type :

```
Runtime error nnn at segment:déplacement
```

En consultant la liste des messages d'erreurs, il est facile d'interpréter le numéro nnn. Il reste ensuite à déterminer la ligne du programme où l'erreur s'est produite.

Si le programme est relativement petit, le plus efficace est de le compiler et l'exécuter dans l'environnement intégré, avec la directive $D+ ; ainsi la ligne où se produit l'erreur sera localisée automatiquement. Une autre solution est de compiler, avec **TPC.EXE**, le programme et toutes les unités qu'il utilise avec la directive /F pour indiquer l'adresse de l'erreur et la directive $D pour générer les informations nécessaires à la recherche de la position de l'erreur :

```
tpc prog /B /F08D4:0045 /$D+
```

L'adresse indiquée dans le message d'erreur doit obligatoirement être placée derrière /F, sans introduire d'espace. /B oblige à compiler toutes les unités dont le source est accessible. /$D+ oblige à créer les informations pour le *débogueur*, qui seront mémorisées dans les fichiers *.TPU* et dans le fichier *.EXE*.

S'il est possible de trouver l'endroit où se produit l'erreur, la ligne correspondante sera montrée à l'écran.

Si le programme est trop grand (beaucoup d'unités, message **Out of memory** pendant la compilation avec $D+) ou si la méthode précédente s'est terminée par le message **Target adress not found**, il vaut mieux tout d'abord détecter dans quel module l'erreur s'est produite. Pour ce faire :

- compiler le programme avec l'option /GS pour créer un fichier *.MAP* avec tous les segments utilisés dans le programme ;
- consulter alors le fichier *.MAP* pour trouver l'unité qui correspond au segment indiqué dans le message d'erreur ;

- une fois l'unité identifiée, la compiler avec la directive /$D+ pour créer les informations nécessaires dans le fichier *.TPU* ;

- recompiler alors le programme avec l'option /Fsegment:déplacement ; ou compiler avec l'option /GD et consulter le fichier *.MAP* où le numéro de chaque ligne de l'unité est accompagné de l'adresse correspondante, pour rechercher la ligne qui correspond au déplacement indiqué dans le message d'erreur.

Pour illustrer cette méthode, nous allons prendre pour exemple le programme PROGERR qui utilise l'unité UNITERR, tous deux imprimés ci-dessous :

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{                  exemple de recherche d'erreur d'exécution                 }
{                  (disponible dans le fichier uniterr.pas)                  }
{----------------------------------------------------------------------------}
unit uniterr;
interface
function Factorielle(n : word) : longint;
implementation
function Factorielle(n : word) : longint;
begin
  if n = 0
    then Factorielle := 1
    else Factorielle := round(Factorielle(n-1) * n);
end;
end.
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{                   exemple de recherche d'erreur d'exécution                }
{                   (disponible dans le fichier progerr.pas)                 }
{----------------------------------------------------------------------------}
uses uniterr;
begin
  write(Factorielle(170));
end.
```

Lors de son exécution, ce programme s'arrête sur un message :

```
Runtime error 207 at 0004:0041
```

Par la méthode présentée ci-dessus, la première chose à faire est de recompiler le programme avec l'option /GS, pour générer un fichier PROGERR.MAP :

```
 Start  Stop   Length Name               Class

 00000H 00033H 00034H PROGRAM            CODE
 00040H 00091H 00052H UNITERR            CODE
 000A0H 00C48H 00BA9H SYSTEM             CODE
 00C50H 00ED5H 00286H DATA               DATA
 00EE0H 04EDFH 04000H STACK              STACK
 04EE0H 04EE0H 00000H HEAP               HEAP
```

Dans la première colonne, sont indiquées les adresses absolues du début de chaque segment : ce qui permet de voir que, à l'adresse absolue 00040h (qui correspond à une valeur de segment de 0004h), commence l'unité UNITERR.

On peut alors recompiler l'unité UNITERR avec l'option /$D+ :

```
tpc uniterr /$D+
```

et le programme avec l'option /GD :

```
tpc progerr /GD
```

pour générer un autre fichier *.MAP* où, en plus des informations déjà examinées plus haut, nous trouvons également la liste des lignes de UNITERR et les adresses correspondantes :

```
Line numbers for UNITERR(UNITERR.PAS) segment UNITERR

     6 0004:0000      7 0004:000E      8 0004:0014      9 0004:0020
    10 0004:0047
```

Il devient alors facile d'identifier la ligne 9 de l'unité **UnitErr** comme étant celle où l'erreur s'est produite, puisqu'elle commence en 0004:0020 et que la ligne 10 commence en 0004:0047.

3.2 Erreurs d'exécution sans message.

Il s'agit des erreurs qui bloquent la clavier, obligeant à éteindre et rallumer l'ordinateur ou tout au moins à réinitialiser le système opérationnel (Ctrl+Alt+Del).

Dans cette situation, vérifiez tout d'abord que la compilation des modules a été faite avec les directives $S+, pour contrôler si la pile est suffisante, et $R+ pour contrôler les limites des indices. Dans le doute, recompiler (le programme entier ou une unité en particulier) avec les directives $S+,R+ et exécuter le programme à nouveau pour localiser l'erreur éventuelle.

Si la méthode précédente ne donne pas de résultat, cela devient plus laborieux, la seule alternative restant l'utilisation du **débogueur** (en plus, évidemment, d'une vérification de la logique du programme par le programmeur).

Si le programme est suffisamment court, il suffit d'utiliser le **débogueur** de l'environnement intégré. Dans le cas contraire, **Turbo-Debugger**, autre produit de **Borland**, permet de déboguer un programme exécutable.

Il est difficile de décrire par écrit le débogage en utilisant ces produits, étant donné le caractère interactif du procédé. C'est pourquoi nous nous contenterons d'énumérer ci-dessous les principales touches de fonctions qui peuvent être utilisées ; pour obtenir des informations complémentaires et la signification d'autres touches, consulter le manuel du **Turbo-Pascal** et du **Turbo-Debugger**.

- F4 exécute le programme jusqu'à la ligne où se trouve le curseur ; utile pour arriver rapidement à la partie du programme à déboguer.

- F7 exécute pas-à-pas chaque instruction ; quand on rencontre un appel de routine, elle permet de continuer à l'intérieur de la routine, toujours en exécutant pas-à-pas ; quand on l'utilise pour commencer une exécution, le curseur est placé sur la directive *begin* de la première partie d'initialisation d'unité ou du programme principal.

- F8 exécute également pas-à-pas mais, quand on rencontre un appel de routine, celle-ci est exécutée en une seule fois ; elle peut également être utilisée pour commencer l'exécution.

- Alt F5 visualise l'écran correspondant au résultat de l'exécution du programme, sans sortir du programme.

- Ctrl F4 permet de visualiser et de modifier des variables ; dans le cas d'une structure, il est nécessaire d'identifier les variables par leur nom complet ; elle peut également être utilisée à tout moment comme calculatrice.

- Ctrl F7 permet de définir une liste de symboles du programme dont les valeurs seront visualisées et mises à jour en permanence dans une fenêtre de l'écran (fenêtre de *watch*).

- Ctrl F9 exécute le programme à partir de l'instruction courante jusqu'à la fin ; peut être utilisée pour exécuter le programme sans débogage.

- Ctrl F2 interrompt l'exécution du programme et le réinitialise, ce qui permet de recommencer une nouvelle exécution.

Pour un fonctionnement correct du débogage, il est indispensable de bien utiliser deux directives :

- la directive $D+ pour que les informations nécessaires à la dépuration soient créées à la compilation,

- la directive $L+ pour que les variables locales d'une unité soient inclues dans les informations de débogage (avec $L-, seules les variables publiques seront documentées).

En pratique, dans le cas d'un programme relativement important, il devient assez difficile d'utiliser le débogage, à cause de la quantité de mémoire nécessaire. Avec le **Turbo-Debugger**, la disponibilité d'une extension de mémoire EMS permettra le débogage de programmes plus importants, car le **Turbo-Debugger** est capable d'utiliser cette mémoire pour stocker des informations. Dans tous les cas, tentez de réduire au minimum la taille du programme en ne compilant avec les directives $D+,L+ que les unités où l'erreur a le plus de chance de se trouver.

Enfin, il est conseillé de compiler la version définitive du programme avec les directives $S-,R-,D-,L- et de les activer uniquement en phase de développement.

4 Son.

Deux procédures sont disponibles en Turbo-Pascal pour utiliser le haut-parleur du PC. Il s'agit de **Sound** et **NoSound**.

```
procedure Sound(f : word);
procedure NoSound;
```

La procédure **Sound** fait vibrer la membrane du haut-parleur avec une fréquence f exprimée en hertz. La vibration (et par conséquent le son) continue jusqu'à l'appel à la procédure **NoSound** qui immobilise la membrane. Généralement, on utilise la procédure :

```
procedure Delay(t : word);
```

pour laisser le son pendant t millisecondes, avant de faire appel à **NoSound**.

Le programme ci-dessous est un exemple qui pourrait être réutilisé, par exemple pour accompagner l'affichage d'un message d'erreur.

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                            }
{                      exemple d'utilisation du haut parleur                 }
{                      (disponible dans le fichier biperr.pas)               }
{----------------------------------------------------------------------------}
uses crt;
var
  c : char;

procedure Son(f , t : word);
{emet le son de fréquence f durante t millisecondes}
begin
  sound(f);
  delay(t);
  nosound;
end;
procedure BipErr;
var
  m : word;
begin
  m := 600;
  while m > 100 do
    begin
      Son(m,100);
      m := m - 100;
    end;
end;
begin
  clrscr;
  writeln('taper F pour finir ou une autre touche pour écouter le signal d''erreur');
  c := #0;
  while c <> 'F' do
    begin
      c := upcase(readkey);
      if c <> 'F'
        then BipErr;
    end;
end.
```

5 Exercice.

Transformez votre clavier en piano (jouet). De telle manière qu'une ligne de touches représente les touches blanches et la ligne de touches au-dessus représente les touches noires (notes altérées). Voici le tableau des fréquences des notes à partir du troisième octave :

```
do3  = 264                    do4  = 528
do#3 = 281                    do#4 = 565
re3  = 297                    re4  = 594
re#3 = 313                    re#4 = 625
mi3  = 330                    mi4  = 660
fa3  = 352                    fa4  = 704
fa#3 = 374                    fa#4 = 748
sol3 = 396                    etc...
sol#3= 415
la3  = 440
la#3 = 468
si3  = 495
```

Chapitre IX

Une approche de la programmation orientée objets

Dans sa version 5.5, Turbo-Pascal possède les caractéristiques d'un *Langage Orienté Objets* (LOO). Ce qui n'empêche pas de l'utiliser pour ce qu'on pourrait appeler la *programmation traditionnelle*, comme nous l'avons vu dans les chapitres précédents. Ce dernier chapitre a seulement pour but d'initier le lecteur aux concepts de la *Programmation Orientée Objets* (POO), appliquée au Turbo-Pascal 5.5.

1 Premières définitions.

La POO est une nouvelle méthode de programmation, déjà disponible dans plusieurs langages de programmation (C++, SmallTalk, TP5.5, MicroSoft-Pascal 6.0). Par rapport à la programmation structurée, elle introduit de nouveaux concepts et, par conséquent, une nouvelle terminologie. En Turbo-Pascal, cela conduit à de nouveaux types de données et de nouvelles routines.

1.1 Finalités de la POO.

Elles sont ambitieuses :

- aboutir à une nouvelle méthode de programmation plus proche de la perception naturelle de la réalité ;
- renforcer l'abstraction des données ;
- augmenter la modularité des applications ;
- favoriser la réutilisation de code source.

1.2 Objets, encapsulation, méthodes.

Considérons une table de codification dans un programme de mise à jour de ces tables. On peut la décomposer en plusieurs éléments :

- le code de la table,
- le nom (titre) de la table,
- le nom de la grille de saisie utilisée pour mettre à jour les éléments de la table.

Ce qui pourrait être implémenté de la manière suivante, en définissant un type *record* (TypeTable) et une variable (Table) :

```
type
  TypeTable = record
                CodeTab : string[3];
                NomTab  : string[40]
                Grille  : string[8];
              end;

var
  Table : TypeTable;
```

Pour manipuler cette table, on utiliserait plusieurs routines pour afficher la grille de saisie à l'écran, afficher et/ou saisir les éléments de la table, etc...

En conclusion, en programmation traditionnelle, nous définissons une variable qui décrit les caractéristiques de la table, et des routines qui manipulent ces caractéristiques.

Il est probable que quelqu'un qui n'a jamais pratiqué la programmation structurée aurait une vision différente d'une table de codification : pour lui, il s'agirait d'une fenêtre sur l'écran, avec un titre, où sont visualisés ou peuvent être définis des éléments de la table. Autrement dit, une table est un ensemble de caractéristiques et de possibilités pour manipuler ces caractéristiques.

La POO permet justement d'implémenter cette vision plus naturelle. Un **objet** dans la terminologie de la POO, rassemble les caractéristiques d'un objet déterminé (sous forme de données) et le code exécutable nécessaire pour les manipuler. Cette dernière propriété est appelée **encapsulation**. L'implémentation de l'exemple ci-dessus aurait cet aspect :

```
type
  TypeTable = object
                CodeTab : string[3];
                NomTab  : string[40];
                Grille  : string[8];
                procedure AfficherGrille(Grille : string);
                procedure EffacerTable;
                procedure VisualiserElement(Element : TypeElement);
                etc...
              end;
var
  Table : TypeTable;
```

Les routines qui manipulent les données des objets sont appelées **méthodes**. Il faut remarquer qu'une variable de type ***object*** (appelée également **instance**) est définie de la même façon qu'une variable quelconque. On peut également définir des pointeurs vers des objets et les utiliser de la même manière que des pointeurs vers d'autres types.

1.3 Héritage.

Les propriétés des objets ne s'arrêtent pas là. Il est probable que, pour en revenir à notre exemple, des grilles sont également utilisées pour manipuler d'autres données que les éléments d'une table. Par conséquent, pourquoi ne pas considérer une grille comme un objet en soi ? La routine d'affichage ou la routine d'effacement de la grille seront les mêmes pour toutes les grilles. L'exemple ci-dessus pourrait alors devenir :

```
type
  TypeGrille = object
                 Grille : string[8];
                 procedure AfficherGrille;
                 procedure EffacerGrille;
               end;

TypeTable = object(TypeGrille)
              CodeTab : string[3];
              NomTab  : string[40];
              procedure AfficherElement(Element : TypeElement);
              etc...
            end;
```

```
var
  Table : TypeTable;
```

TypeTable est défini comme étant object(TypeGrille). Ce qui signifie que TypeTable est un **type descendant** de TypeGrille ; par conséquent, il **hérite de toutes les caractéristiques et de toutes les méthodes de son ancêtre (ou type ascendant)**. Cette caractéristique, appelée héritage, constitue une des différences essentielles entre l'utilisation du type *record* dans la programmation traditionnelle et l'utilisation du type *object* dans la POO.

La décomposition de l'exemple ci-dessus pourrait continuer : une grille est une fenêtre avec des champs, l'objet TypeGrille pourrait donc descendre d'un objet fenêtre, etc...

1.4 Champs des objets.

La propriété qu'ont les objets d'hériter se traduit par le fait que le champ Grille, bien qu'il ne soit pas explicitement défini dans l'objet TypeTable, peut être utilisé comme s'il était un champ de TypeTable : la notation **Table.Grille** est tout à fait valide. De même pour les méthodes : l'instruction **Table.AfficherGrille;** exécute la méthode AfficherGrille de l'objet Table, c'est à dire qu'elle affiche la grille associée à Table.

L'exemple ci-dessus montre que les champs et les méthodes d'un objet peuvent être utilisés de la même manière que les champs d'une structure de type *record*. L'instruction *with* peut également être utilisée avec des objets.

Autre point important : au contraire d'autres LOO, Turbo-Pascal permet d'avoir accès directement aux champs d'un objet. Toutefois il est conseillé d'éviter d'utiliser cette facilité, afin d'augmenter l'abstraction des objets. Tous les champs d'un objet devraient être manipulés exclusivement par ses propres méthodes. Quand elle est respectée, cette règle rend plus facile l'adaptation d'une application : pour changer l'implémentation d'un objet, il suffit alors de se préoccuper des données et des méthodes de l'objet, sans se préoccuper du reste du programme ; la seule limitation est de conserver le même en-tête pour les méthodes.

1.5 Compatibilité entre objets.

Nous avons déjà vu que, par suite de l'héritage, un type objet descendant hérite de tous les champs et toutes les méthodes de son ancêtre. Il hérite également de la compatibilité avec tous ses ancêtres. Dans l'exemple des tables déjà vu plus haut, cela signifie que les instructions suivantes sont correctes :

```
var
  GrilAux : TypeGrille;
  GrilPtr : ^TypeGrille;
  TablePtr : ^TypeTable;
.....
  GrilAux := Table;
  GrilPtr := TablePtr;
```

Comme on le voit, la compatibilité est également valable pour les pointeurs. Les types descendants peuvent être utilisés en lieu et place d'un ancêtre quelconque. La réciproque n'est pas vraie (Table := GrilAux provoquerait une erreur de compilation).

2 Exemples.

Un premier exemple utilise Turbo-Screen, gestionnaire de grilles d'écran utilisable en Turbo-Pascal. Pour saisir un champ sur l'écran avec Turbo-Screen, il suffit d'envoyer une commande par l'instruction *write* du Turbo-Pascal et, si le code de retour vaut 'RET' (=retour chariot), de récupérer la valeur saisie par l'instruction *readln* :

```
write(Deb, 'SAISIE,nom_champ, CR', Fin);
if Cr = 'RET' then readln(Variable);
```

Dans le cas d'une grille à plusieurs champs, cette séquence de code se répète souvent et il serait intéressant d'implémenter une manière de réduire le code source nécessaire à l'exécution de cette commande.

En Turbo-Pascal traditionnel, la solution est de définir une routine de saisie, ayant comme paramètres : le nom du champ sur l'écran, le type de variable à récupérer, et la variable (sans type) où sera placée la valeur saisie. Cette manière de faire est disponible sur la disquette d'exemples dans le fichier SAISIE.PAS.

Une autre solution, implémentée en POO, est disponible dans le fichier OSAISIE.PAS et imprimée ci-dessous. Les deux programmes utilisent la grille OSAISIE.MAP et supposent que AFFTURBO.COM (partie résidente de Turbo-Screen) a été chargée.

```
{----------------------------cours de Pascal-Avancé---------------------------}
{         exemple de déclaration et utilisation d'objets pour paramétrer       }
{                     la saisie de champs avec Turbo-Screen                    }
{                   (disponible dans le fichier osaisie.pas)                   }
{------------------------------------------------------------------------------}
uses tscreen;
type
  ochamp = object
             function Saisie(NomChamp : string): string;
           end;
  oreal = object(ochamp)
            r : real;
            procedure Saisie(NomChamp : string);
          end;
  obyte = object(ochamp)
            b : byte;
            procedure Saisie(NomChamp : string);
          end;
  ointeger = object(ochamp)
            i : integer;
            procedure Saisie(NomChamp : string);
          end;

  TypeEnr = record
              vreal    : oreal;
              vinteger : ointeger;
              vbyte    : obyte;
              vstring  : string[25];
            end;
var
  Enr : TypeEnr;
  vchamp : ochamp;
```

```
function ochamp.Saisie;
var
  s : string;
begin
  write(Deb, 'SAISIE,'+NomChamp+',CR', Fin);
  readln(Cr);
  if Cr = 'RET'
    then readln(s)
    else s := '';
  Saisie := s;
end;
procedure oreal.Saisie(NomChamp : string);
var
  s : string;
  c : integer;
begin
  s := ochamp.Saisie(NomChamp);
  val(s, r, c);
end;
procedure obyte.Saisie(NomChamp : string);
var
  s : string;
  r : integer;
begin
  s := ochamp.Saisie(NomChamp);
  val(s, b, r);
end;
procedure ointeger.Saisie(NomChamp : string);
var
  s : string;
  r : integer;
begin
  s := ochamp.Saisie(NomChamp);
  val(s, i, r);
end;
begin
  write(Deb, 'UTILISE,OSAISIE.MAP',Fin);
  with Enr do
    begin
      vbyte.Saisie('TELBYTE');
      vinteger.Saisie('TELINTEGER');
      vreal.Saisie('TELREAL');
      vstring := vchamp.Saisie('TELSTRING');
    end;
end.
```

Un type *object* **ochamp** a été défini (avec la particularité de ne posséder aucun champ mais seulement une méthode) ainsi que trois autres types *object* (**oreal**, **obyte** et **ointeger**) descendants de **ochamp**, chacun redéfinissant la méthode de **ochamp**. Il n'a pas été possible de définir un type **ostring** pour les champs de type *string* car ils sont de taille variable. Une **instance** vchamp (de type **ochamp**) sera pourtant utilisée pour traiter les champs de type *string*.

La variable Enr, de type TypeEnr, va recevoir les données saisies. Remarquons que chaque champ de cette structure est de type *object*, à l'exception de vstring.

La méthode **Saisie** de chaque objet utilise la méthode **Saisie** du type **ochamp** pour récupérer les caractères saisis au clavier et les transformer dans le type de la variable correct.

Il suffit alors, dans le programme principal, d'appeler la méthode de chaque objet pour saisir les données. Ces objets pourraient être déclarés dans une *unit* et être réutilisés sans travail supplémentaire dans d'autres programmes qui utilisent Turbo-Screen.

Il est parfois utile de définir un **objet abstrait** auquel ne correspondra aucune **instance** mais qui servira d'ancêtre à d'autres objets. Le cas d'**ochamp** est comparable : bien qu'il ne possède aucun champ de données, une **instance** en a été déclarée pour saisir les champs de type *string* ; remarquons que la variable **ochamp**, bien qu'elle soit indispensable, a une taille nulle (ceci peut être vérifié en utilisant l'opérateur *sizeof*).

Un autre exemple est présenté ci-dessous et disponible dans le fichier OFENETR.PAS :

```
{---------------------------cours de Pascal-Avancé---------------------------}
{                                                                             }
{                    exemple de déclaration et utilisation d'objets           }
{                     (disponible dans le fichier ofenetr.pas)                }
{-----------------------------------------------------------------------------}
uses crt;
type
  opoint = object
             Ligne,
             Colonne    : byte;
             procedure Init(x, y : byte);
             function LignPos : byte;
             function ColoPos : byte;
           end;
  orectangle = object(opoint)
                 Largeur,
                 Hauteur : byte;
                 Sauve   : pointer;
                 procedure Init(x, y, l, h : byte);
                 procedure Afficher;
                 procedure Effacer;
                 procedure Deplacer(dx, dy : integer);
                 procedure Liberer;
               end;
  ofenetre = object(orectangle)
              Titre : string[80];
              procedure Init(x, y, l, h : byte; t : string);
              procedure Afficher;
              procedure Deplacer(dx, dy : integer);
            end;
var
  Fenetre : ofenetre;

procedure opoint.Init(x, y : byte);
begin
  Colonne := x;
  Ligne := y;
end;

function opoint.LignPos : byte;
begin
  LignPos := Ligne;
end;
```

```
function opoint.ColoPos : byte;
begin
  ColoPos := Colonne;
end;

procedure orectangle.Init(x, y, l, h : byte);
begin
  opoint.Init(x, y);
  Largeur := l;
  Hauteur := h;
  GetMem(Sauve, h*l*2);
end;

procedure orectangle.Afficher;
var
  i, j : byte;
  p : pointer;
begin
  p := Sauve;
  for i := Ligne to pred(Ligne+Hauteur) do
    begin
      Move(Mem[$b800:2*(pred(i)*80+pred(Colonne))], p^, Largeur*2);
      p := ptr(seg(p^), ofs(p^)+Largeur*2);
    end;
  gotoxy(Colonne, Ligne);
  write('┌');
  for i := succ(Colonne) to Colonne+Largeur-2 do write('-');
  write('┐');
  for i := 2 to pred(Hauteur) do
    begin
      gotoxy(Colonne, pred(Ligne+i));
      write('|');
      for j := 2 to pred(Largeur) do write(' ');
      gotoxy(pred(Colonne+Largeur), pred(Ligne+i));
      write('|');
    end;
  gotoxy(Colonne, pred(Ligne+Hauteur));
  write('└');
  for i := succ(Colonne) to Colonne+Largeur-2 do write('-');
  write('┘');
end;

procedure orectangle.Effacer;
var
  i : byte;
  p : pointer;
begin
  p := Sauve;
  for i := Ligne to pred(Ligne+Hauteur) do
    begin
      Move(p^, Mem[$b800:2*(pred(i)*80+pred(Colonne))],  Largeur*2);
      p := ptr(seg(p^), ofs(p^)+Largeur*2);
    end;
end;
```

```
procedure orectangle.Deplacer(dx, dy : integer);
begin
  if (Ligne+dy >= 1) and (Ligne+dy+Hauteur <= 25)
     and (Colonne + dx >= 1) and (Colonne+Largeur+dx <= 80)
    then begin
      Effacer;
      Ligne := Ligne+dx;
      Colonne := Colonne+dy;
      Afficher;
    end;
end;

procedure orectangle.Liberer;
begin
  freemem(Sauve, Largeur*Hauteur*2);
end;

procedure ofenetre.Init(x, y, l, h : byte; t : string);
begin
  orectangle.Init(x, y, l, h);
  Titre := t;
end;

procedure ofenetre.Afficher;
var
  l : byte;
begin
  orectangle.Afficher;
  l := (Largeur-length(Titre)) div 2;
  gotoxy(Colonne+pred(l), Ligne);
  write(Titre);
end;

procedure ofenetre.Deplacer(dx, dy : integer);
begin
  if (Ligne+dy >= 1) and (Ligne+dy+Hauteur <= 25)
     and (Colonne + dx >= 1) and (Colonne+Largeur+dx <= 80)
    then begin
      Effacer;
      Ligne := Ligne+dy;
      Colonne := Colonne+dx;
      Afficher;
    end;
end;

begin
  Fenetre.Init(5, 10, 40, 10, 'Fenêtre d''exemple');
  Fenetre.Afficher;
  while not keypressed do
    begin
      Fenetre.Deplacer(integer(Random(40)-20), integer(Random(10)-5));
      delay(200);
    end;
  Fenetre.Effacer;
  Fenetre.Liberer;
end.
```

Dans cet exemple artificiel :

- **opoint** est un objet abstrait qui définit une position sur la grille (Ligne et Colonne) et possède une méthode pour s'initialiser ;

- **orectangle** est un objet qui descend de **opoint**, et qui définit un rectangle sur l'écran (coin supérieur gauche, largeur, hauteur et pointeur vers une zone où sera sauvé le contenu de l'écran tout de suite avant l'affichage du rectangle) ; il possède plusieurs méthodes pour s'initialiser, pour s'afficher et se déplacer sur l'écran, pour s'effacer et pour se détruire ;

- **ofenetre** est un objet qui descend de **orectangle**, et qui définit une fenêtre comme étant un rectangle et un titre ; il possède également des méthodes pour s'initialiser, s'afficher et se déplacer sur l'écran.

Chaque objet hérite des champs et des méthodes de son ancêtre : par exemple, **ofenetre** n'a pas de méthode pour se détruire ou s'effacer car elles seraient identiques à celles de **orectangle** ; il a le droit d'utiliser directement les méthodes équivalentes de son ancêtre. Au contraire, il est nécessaire de redéfinir la méthode **Afficher** car elle est plus complète dans le cas d'une fenêtre (affichage du titre).

Alors pourquoi est-il nécessaire de redéfinir la méthode **Deplacer** pour **ofenetre** puisque son code source est identique à celui de **orectangle** ? En réalité, bien que le code source semble le même, dans chacune des méthodes **Deplacer** la routine **Afficher** utilisée sera celle de l'objet considéré et, par conséquent, le code exécuté sera différent.

Le programme principal ne fait que déplacer la fenêtre sur l'écran de manière aléatoire, jusqu'à ce une touche quelconque du clavier soit appuyée.

3 Méthodes virtuelles, polymorphisme.

Les méthodes rencontrées jusqu'à maintenant sont des **méthodes statiques**, dans la mesure où elles sont connues dès la compilation. Cela est souvent insuffisant pour gérer correctement les méthodes. Comme nous l'avons vu plus haut, dans le second exemple, il paraît superflu de redéfinir la méthode **Deplacer** pour l'objet **ofenetre**, puisque son code source est identique à celui de la méthode correspondante de l'objet **orectangle**. Cet inconvénient disparait avec l'usage de **méthodes virtuelles**.

Les variables peuvent être divisées en variables statiques, dont l'emplacement est connu dès la compilation et en variables dynamiques allouées durant l'exécution. La distinction entre méthodes **statiques** et méthodes **virtuelles** est comparable : les premières sont reliées aux objets dès l'édition de liens, alors que les secondes sont reliées aux objets, le plus tard possible, pendant l'exécution.

3.1 Méthodes virtuelles.

Pour déclarer une méthode virtuelle, il suffit d'ajouter le mot réservé *virtual* immédiatement après la déclaration de la méthode. Par exemple :

```
procedure Afficher; virtual;
```

Dans un type descendant, toute méthode qui redéfinit une méthode virtuelle doit être virtuelle.

La procédure d'initialisation d'un objet qui comporte au moins une méthode virtuelle n'est plus une *procedure* mais un **constructor** (nouveau mot réservé du Turbo-Pascal). Le **constructor** d'un objet réalise automatiquement des opérations en rapport avec la **table des méthodes virtuelles** (TMV). C'est pourquoi, il doit être exécuté avant tout appel à une

méthode virtuelle de l'objet. Dans l'exemple précédent, une fois que **Deplacer** a été déclarée comme *virtual*, les méthodes **Init** de orectangle et ofenetre doivent être déclarées ainsi :

```
constructor Init(x, y, l, h : byte);
constructor Init(x, y, l, h : byte; t : string);
```

Le nom Init n'est pas obligatoire. Il est cependant pratique de l'utiliser systématiquement car il est devenu un standard de Borland et des bibliothèques d'objets disponibles. Un objet peut posséder plusieurs **constructors** (de noms différents). Un **constructor** ne peut pas être *virtual*.

3.2 Polymorphisme.

Pour en revenir à notre exemple de fenêtres, nous avons déjà noté que la méthode **Deplacer**, bien qu'ayant le même code source, doit être redéfinie pour les deux objets (orectangle et ofenetre) car la méthode **Afficher** est propre à chacun des objets.

Ce problème peut être résolu avec l'utilisation de méthodes virtuelles. Après avoir défini les méthodes utilisées par **Deplacer** comme virtuelles, il devient inutile de déclarer **Deplacer** pour l'objet **ofenetre**. Puisque **ofenetre** descend de **orectangle**, il hérite de la méthode **Deplacer** et comme **Afficher** est déclaré comme *virtual*, la liaison entre **Deplacer** et **Afficher** sera réalisée au moment de l'exécution :

- s'il s'agit de déplacer une variable de type **orectangle**, le programme exécutera **Afficher** et **Effacer** de **orectangle** ;
- s'il s'agit de déplacer une variable de type **ofenetre**, le programme exécutera **Afficher** de **ofenetre** et **Effacer** de **orectangle** (puisque cette dernière méthode n'est pas redéfinie pour l'objet **ofenetre**).

Ce concept est nommé **polymorphisme**, auquel sont associés les objets **polymorphes**. Le **polymorphisme** est une des caractéristiques les plus puissantes de la POO. Si on suppose que les objets ci-dessus ont été déclarés dans une *unit* distribuée sous forme de fichier *.TPU*, le programmeur, connaissant la partie interface de l'*unit*, peut créer de nouveaux objets descendants de **orectangle** et **ofenetre** et leur appliquer la méthode **Deplacer** de **orectangle**, sans avoir à la redéfinir. On en conclut que la méthode **Deplacer** sera utilisée pour des opérations sur des variables dont le type n'était pas connu au moment de la compilation.

4 Objets dynamiques.

4.1 Allocation d'objets dynamiques.

De même qu'une variable, une instance d'objet peut être allouée dans la mémoire dynamique, grâce à la procédure **New** (voir chapitre II). Exemple :

```
var
  GrillePtr : ^TypeGrille;

New(GrillePtr);
```

Les objets qui comportent des méthodes virtuelles doivent être initialisés par leur **constructor** en même temps qu'ils sont alloués. C'est pourquoi la procédure **New** accepte un second paramètre et peut fonctionner comme une fonction qui renvoie un pointeur. Voici

l'en-tête qui correspond à cette nouvelle syntaxe :

```
function New(Type de l'objet, Routine d'initialisation) : pointer;

Exemple :

var
  PtrFenetre : ^ofenetre;

PtrFenetre := New(ofenetre, Init(12, 3, 40, 15, 'Fenêtre d'exemple'));
```

4.2 Libération d'objets dynamiques.

De la même façon, la procédure **Dispose** a été améliorée pour permettre la libération d'objets dynamiques :

```
procedure Dispose(Pointeur vers l'objet, Routine de libération);
```

Un nouveau mot réservé permet de déclarer les routines de libération des objets dynamiques. Il s'agit de **destructor** qui remplace le mot *procedure* dans l'en-tête.

Il est conseillé d'utiliser le nom *Done* (standard de Borland) pour un **destructor**. Un objet peut posséder plusieurs **destructors**, de noms différents. Un **destructor** peut être vide :

```
destructor ofenetre.done;
begin
end;
```

Des instructions seront automatiquement insérées par le compilateur, notamment pour détruire la table des méthodes virtuelles.

4.3 Exemple.

Le programme présenté ci-dessous est une nouvelle version de l'exemple du paragraphe 2, avec utilisation de méthodes virtuelles et allocation d'objets dynamiques.

```
{--------------------------cours de Pascal-Avancé--------------------------}
{                                                                           }
{  exemple de déclaration et utilisation d'objets polymorphes et dynamiques }
{                    (disponible dans le fichier ofenetrl.pas)              }
{---------------------------------------------------------------------------}
uses crt;
type
  opoint = object
             Ligne,
             Colonne    : byte;
             procedure Init(x, y : byte);
             function LignPos : byte;
             function ColoPos : byte;
           end;
  orectangle = object(opoint)
                 Largeur,
                 Hauteur : byte;
                 Sauve   : pointer;
                 constructor Init(x, y, l, h : byte);
                 procedure Afficher; virtual;
```

```
                procedure Effacer; virtual;
                procedure Deplacer(dx, dy : integer);
                destructor Done;
              end;
  ofenetre = object(orectangle)
              Titre : string[80];
              constructor Init(x, y, l, h : byte; t : string);
              procedure Afficher; virtual;
              destructor Done;
            end;

  Ptrofenetre = ^ofenetre;

var
  Fenetre : Ptrofenetre;

procedure opoint.Init(x, y : byte);
begin
  Colonne := x;
  Ligne := y;
end;

function opoint.LignPos : byte;
begin
  LignPos := Ligne;
end;

function opoint.ColoPos : byte;
begin
  ColoPos := Colonne;
end;

constructor orectangle.Init(x, y, l, h : byte);
begin
  opoint.Init(x, y);
  Largeur := l;
  Hauteur := h;
  GetMem(Sauve, h*l*2);
end;

procedure orectangle.Afficher;
var
  i, j : byte;
  p : pointer;
begin
  p := Sauve;
  for i := Ligne to pred(Ligne+Hauteur) do
    begin
      Move(Mem[$b800:2*(pred(i)*80+pred(Colonne))], p^, Largeur*2);
      p := ptr(seg(p^), ofs(p^)+Largeur*2);
    end;
  gotoxy(Colonne, Ligne);
  write('┌');
  for i := succ(Colonne) to Colonne+Largeur-2 do write('-');
  write('┐');
  for i := 2 to pred(Hauteur) do
    begin
      gotoxy(Colonne, pred(Ligne+i));
      write('|');
```

```
      for j := 2 to pred(Largeur) do write(' ');
      gotoxy(pred(Colonne+Largeur), pred(Ligne+i));
      write('|');
    end;
  gotoxy(Colonne, pred(Ligne+Hauteur));
  write('└');
  for i := succ(Colonne) to Colonne+Largeur-2 do write('-');
  write('┘');
end;

procedure orectangle.Effacer;
var
  i : byte;
  p : pointer;
begin
  p := Sauve;
  for i := Ligne to pred(Ligne+Hauteur) do
    begin
      Move(p^, Mem[$b800:2*(pred(i)*80+pred(Colonne))],  Largeur*2);
      p := ptr(seg(p^), ofs(p^)+Largeur*2);
    end;
end;

procedure orectangle.Deplacer(dx, dy : integer);
begin
  if (Ligne+dy >= 1) and (Ligne+dy+Hauteur <= 25)
     and (Colonne + dx >= 1) and (Colonne+Largeur+dx <= 80)
    then begin
      Effacer;
      Ligne := Ligne+dy;
      Colonne := Colonne+dx;
      Afficher;
    end;
end;

destructor orectangle.Done;
begin
  freemem(Sauve, Largeur*Hauteur*2);
end;

constructor ofenetre.Init(x, y, l, h : byte; t : string);
begin
  orectangle.Init(x, y, l, h);
  Titre := t;
end;

procedure ofenetre.Afficher;
var
  l : byte;
begin
  orectangle.Afficher;
  l := (Largeur-length(Titre)) div 2;
  gotoxy(Colonne+pred(l), Ligne);
  write(Titre);
end;
```

```
destructor ofenetre.Done;
begin
  orectangle.Done;
end;

begin
  Fenetre := new(Ptrofenetre, Init(5, 10, 40, 10, 'Fenêtre d''exemple'));
  Fenetre^.Afficher;
  while not keypressed do
    begin
      Fenetre^.Deplacer(integer(Random(40)-20), integer(Random(10)-5));
      delay(200);
    end;
  Fenetre^.Effacer;
  Dispose(Fenetre, Done);
end.
```

5 Compléments.

L'objectif de ce chapitre IX étant de donner une première notion de la POO ; nous allons, dans cette dernière partie, en énumérer d'autres aspects, sans les détailler.

Une constante de type *object* peut être déclarée sous une forme identique à une constante de type *record*. Une valeur initiale est attribuée, aux seuls champs de l'objet.

Un type *object* ne peut pas être défini dans la partie déclaration d'une routine.

Un objet ne peut pas être un élément d'un fichier de type *file*, *file of* ou *text*.

L'unité **OBJECTS** disponible dans le Turbo-Pascal 5.5 implémente plusieurs types d'objets :

- le type **Base**, objet abstrait, ascendant ultime de tous les autres,
- le type **Stream** (flux) qui implémente des fonctions équivalentes à celles du type *file*, pour permettre l'utilisation de fichiers d'objets polymorphes,
- les types **DosStream** et **BufStream** qui implémentent des fichiers d'objets reconnus par le DOS,
- les types **Node** et **List** qui implémentent le traitement de listes d'objets.

Sont également disponibles plusieurs programmes d'exemples d'utilisation des objets énumérés ci-dessus.

Annexe I

Partie interface des unités standards.

Les unités **Graph** et **Overlay** ayant déjà été étudiées, respectivement, aux chapitres I et VII, il n'en sera pas question dans cette annexe.

Unité SYSTEM

```
const
  OvrCodeList  : word = 0;     {liste des segments du tampon d'overlay}
  OvrHeapSize  : word = 0;     {taille initiale du tampon d'overlay}
  OvrDebugPtr  : pointer = nil;{pointeur pour le débogage d'overlays}
  OvrHeapOrg   : word = 0;     {origine du tampon d'overlay}
  OvrHeapPtr   : word = 0;     {pointeur du tampon d'overlay}
  OvrHeapEnd   : word = 0;     {fin du tampon d'overlay}
  OvrLoadList  : word = 0;     {liste des overlays chargés en mémoire}
  OvrDosHandle : word = 0;     {gestionnaire DOS d'overlay}
  OvrEMSHandle : word = 0;     {gestionnaire EMS d'overlay}
  HeapOrg      : pointer = nil;{début du tas (mémoire dynamique)}
  HeapPtr      : pointer = nil;{pointeur vers la fin de la partie occupée du tas}
  FreePtr      : pointer = nil;{pointeur vers la table des fragments}
  FreeMin      : word = 0;     {taille minimum de la table des fragments}
  HeapError    : pointer = nil;{adresse de la fonction de gestion des erreurs du tas}
  ExitProc     : pointer = nil;{adresse de la procédure de sortie du programme}
  ExitCode     : integer = 0;  {code de retour du programme}
  ErrorAddr    : pointer = nil;{adresse de la procédure de traitement d'erreur}
  PrefixSeg    : word = 0;     {préfixe de segment de programme (PSP)}
  StackLimit   : word = 0;     {limite inférieure du pointeur de la pile}
  InOutRes     : integer = 0;  {code de retour des opérations d'entrée/sortie}
  RandSeed     : longint = 0;  {valeur aléatoire initiale}
  FileMode     : byte = 2;     {mode d'ouverture des fichiers}
  Test8087     : byte = 0;     {témoin d'installation ou non du co-processeur 80x87}

var
  Input        : text;         {fichier d'entrée standard (clavier)}
  Output       : text;         {fichier de sortie standard (écran)}
  SaveInt00    : pointer;      {sauvegarde du vecteur d'interruption 00}
  SaveInt02, SaveInt1b, SaveInt23, SaveInt24, SaveInt34, SaveInt35, SaveInt36, SaveInt37, SaveInt38,
  SaveInt39, SaveInt3a, SaveInt3b, SaveInt3c, SaveInt3d, SaveInt3e, SaveInt3f, SaveInt75 : pointer;
                               {sauvegarde du vecteur d'interruption correspondant}

function Abs(x : entier ou réel) : entier ou réel;
     Renvoie la valeur absolue de x.
function Addr(x : variable, fonction ou procédure) : pointer;
     Renvoie l'adresse (pointeur) de x.
procedure Append(var Fichier : text);
     Ouvre un fichier de type "text" en mode ajout à la fin du fichier.
function ArcTan(x : real) : real;
     Renvoie la valeur en radians de l'arc-tangente de x.
procedure Assign(Fichier : file, file of ou text; NomFichier : string);
     Associe un nom de fichier physique à la variable Fichier.
```

```
procedure BlockRead(var Fichier : file; var Tampon; N : word[; L : word]);
      Essaye de lire N enregistrements de Fichier et les stocke dans Tampon. Renvoie, dans la
      variable L, le nombre d'enregistrements effectivement lus.
procedure BlockWrite(var Fichier: file; var Tampon; N : word [; E : word]);
      Essaye d'écrire dans Fichier, N enregistrements stockés dans Tampon. Renvoie dans la variable
      E le nombre d'enregistrements effectivement écrits.
procedure ChDir(Rep : string);
      Permet de spécifier Rep comme répertoire courant.
function Chr(B : byte) : char;
      Renvoie le caractère qui correspond à la valeur numérique B dans la table ASCII.
procedure Close(var Fichier : file, file of ou text);
      Ferme un fichier ouvert.
function Concat(Str1 [, Str2, ...., Strn]) : string;
      Met bout à bout plusieurs chaines de caractères pour en constituer une seule.
function Copy(Str : string; Debut, Taille : integer) : string;
      Renvoie une partie de chaine de caractères (sous-chaine).
function Cos(Angle : real) : real;
      Renvoie le cosinus d'Angle. Angle étant mesuré en radians.
function CSeg : word;
      Renvoie la valeur du registre CS (segment de code).
procedure Dec(var X : type scalaire [; N : longint]);
      Diminue de N la valeur de X. Equivalent à X := X - N; mais plus efficace.
procedure Delete(var Str : string; Debut, Taille : byte);
      Supprime Taille caractères de Str, à partir de la position Debut.
procedure Dispose(var P : pointer);
      Libère la mémoire dynamique occupée par la variable P^.
function DSeg : word;
      Renvoie la valeur du registre DS (segment de données).
function Eof(var Fichier : file, file of ou text) : boolean;
      Renvoie la valeur (true ou false) de l'indicateur de fin de fichier.
function Eoln(var Fichier : text) : boolean;
      Renvoie la valeur (true ou false) de l'indicateur de fin de ligne d'un fichier de type "text".
procedure Erase(var : Fichier : file, file of ou text);
      Supprime un fichier. Le fichier doit être fermé.
procedure Exit;
      Provoque la sortie immédiate d'une routine ou du programme principal.
function Exp(X : real) : real;
      Renvoie la valeur de l'exponentielle de x.
function FilePos(var Fichier : file ou file of) : longint;
      Renvoie la position courante (numéro de l'enregistrement) dans le fichier.
function FileSize(var Fichier : file ou file of) : longint;
      Renvoie le nombre d'enregistrements d'un fichier.
procedure FillChar(var V : type quelconque; N : word; Car : char);
      Remplit avec le caractère Car, N positions à partir du début de V.
procedure Flush(var Fichier : text);
      Ecrit dans le fichier le contenu du tampon en mémoire associé à ce fichier.
function Frac(X : real) : real;
      Renvoie la partie décimale de X.
procedure FreeMem(var P : pointer; T : word);
      Libère T octets de la mémoire dynamique à partir de l'adresse pointée par P.
procedure GetMem(var P : pointer; T : word);
      Crée une nouvelle variable dynamique et lui alloue un espace de T octets.
procedure Halt[(Code : word]);
      Stoppe l'exécution du programme et retourne éventuellement un code d'erreur.
function Hi(I : integer ou word) : byte;
      Renvoie la valeur de l'octet le plus significatif de I.
procedure Inc(var X : type scalaire [; N : longint]);
      Augmente de n la valeur de X. Equivalent à X := X + N; mais plus rapide.
```

```
procedure Insert(Str1 : string; var Str2 : string; I : byte);
      Insère une chaîne de caractères Str1, à partir de la position I de la chaîne Str2.
function Int(X : real) : real;
      Renvoie la partie entière de X.
function IOResult : word;
      Renvoie le code de retour de la dernière opération d'entrée-sortie.
function Length(Str : string) : byte;
      Renvoie le nombre de caractères de la chaîne Str.
function Ln(X : real) : real;
      Renvoie le logarithme naturel de X.
function Lo(I : integer ou word) : byte;
      Renvoie la valeur de l'octet le moins significatif de I.
procedure Mark(var P : pointer);
      Mémorise, dans le pointeur P, la valeur actuelle du pointeur de tas (HeapPtr).
function MaxAvail : longint;
      Renvoie la taille du plus grand bloc de mémoire dynamique disponible.
function MemAvail : longint;
      Renvoie la taille totale de tous les blocs de mémoire dynamique disponibles.
procedure MkDir(Rep : string);
      Crée un nouveau répertoire nommé Rep.
procedure Move(var Source, Destin; T : word);
      Copie T octets de Source vers Destin.
procedure New(var P : pointer);
      Crée une nouvelle variable dynamique et lui alloue la mémoire dynamique nécessaire.
function Odd(X : byte, integer, word ou longint) : boolean;
      Renvoie true si X est impair.
procedure Ofs(X : variable, fonction ou procédure) : word;
      Renvoie la partie déplacement de l'adresse de X.
function Ord(X : scalaire) : longint;
      Renvoie le numéro d'ordre d'une valeur scalaire.
function ParamCount : word;
      Renvoie le nombre de paramètres de la ligne de commande.
function ParamStr(I : word) : string;
      Renvoie le Ième paramètre de la ligne de commande.
function Pi : real;
      Renvoie une valeur approchée de π.
function Pos(Str1, Str2 : string) : byte;
      Renvoie la position où commence la sous-chaîne Str1 à l'intérieur de la chaîne Str2.
function Pred(X : scalaire) : scalaire;
      Renvoie la valeur de l'élément qui précède X.
function Ptr(Segm, Ofs : word) : pointer;
      Renvoie un pointeur construit avec le segment Segm et le déplacement Ofs.
function Random [(Limite : word ou real)] : real ou word;
      Renvoie un nombre aléatoire positif compris entre 0 et 1 ou entre 0 et Limite.
procedure Randomize;
      Initialise le générateur de nombres aléatoires.
procedure Read(Fichier : file, file of ou text; Var1[, Var2, ...]);
      Lit un enregistrement d'un fichier ou des variables d'un type quelconque dans un fichier
      texte.
procedure ReadLn(Fichier : text; Var1[, Var2, ...]);
      Exécute la procédure Read et passe à la ligne suivante d'un fichier texte.
procedure Release(var P : pointer);
      Restaure une configuration du tas sauvée précédemment par Mark.
procedure Rename(var Fichier : file, file of ou text; NouveauNom : string);
      Renomme un fichier fermé.
procedure Reset(var Fichier : file, file of ou text[; T : word]);
      Ouvre un fichier déjà existant, en indiquant éventuellement la taille des enregistrements.
procedure Rewrite(var Fichier : file, file of ou text[; T : word]);
      Crée et ouvre un nouveau fichier, en indiquant éventuellement la taille des enregistrements.
```

```
procedure RmDir(Rep : string);
      Supprime un répertoire vide.
function Round(x : real) : longint;
      Arrondit la valeur de X pour renvoyer un entier.
procedure RunError[(Code : word)];
      Stoppe un programme en générant une erreur d'exécution.
procedure Seek(var Fichier : file ou file of; N : longint);
      Définit l'enregistrement N comme enregistrement courant du fichier.
function SeekEof[(var Fichier : test)] : boolean;
      Renvoie la valeur de l'indicateur de fin de fichier texte, en ignorant les espaces.
function SeekEoln([Fichier : text)] : boolean;
      Renvoie la valeur de l'indicateur de fin de ligne d'un fichier texte.
function Seg(X : variable, fonction ou procédure) : word;
      Renvoie la partie "segment" de l'adresse de X.
procedure SetTextBuf(var Fichier : text; var Buffer [; T : word]);
      Associe un nouveau tampon d'entrée-sortie à un fichier texte.
function Sin(Angle : real) : real;
      Renvoie le sinus de Angle. Angle étant exprimé en radians.
function SizeOf(X : variable ou type) : word;
      Renvoie la taille en octets du paramètre X.
function SPtr : word;
      Renvoie la valeur du registre SP (pointeur de la pile).
function Sqr(x : entier ou réel) : entier ou réel;
      Renvoie la valeur de x élevé au carré.
function Sqrt(x : real) : real;
      Renvoie la racine carrée de x.
function SSeg : word;
      Renvoie la valeur du registre SS (segment de la pile).
procedure Str(X[T : byte [ : N : byte] ], var Str1 : string);
      Convertit une valeur numérique en sa représentation sous forme de chaîne de caractères de T
      positions et N décimales.
function Succ(X : scalaire) : scalaire;
      Renvoie la valeur de l'élément qui suit X.
procedure Swap(i : integer ou word);
      Echange l'octet le moins significatif de I avec l'octet le plus significatif.
function Trunc(x : real) : longint;
      Renvoie la partie entière de x.
procedure Truncate(var Fichier : file, fileof ou text);
      Tronque le fichier à la position courante du pointeur dans celui-ci.
function UpCase(C : char) : char;
      Renvoie la valeur du caractère C après transformation en majuscule.
procedure Val(Str : string; var x : integer, byte, word, longint ou real; var Erreur : integer);
      Convertit une chaîne de caractères en valeur numérique.
procedure Write(Fichier : file, file of ou text; Var1[, Var2, ...]);
      Ecrit un enregistrement dans un fichier ou des variables de type quelconque dans un fichier
      texte .
procedure WriteLn(Fichier : text; Var1[, Var2, ...]);
      Exécute la procédure Write puis passe à la ligne suivante d'un fichier texte.
```

Unité CRT

```
const
  {modes d'affichage à l'écran, utilisés par TextMode}
  BW40    = 0;      {40 colonnes x 25 lignes, noir et blanc}
  CO40    = 1;      {40 colonnes x 25 lignes, couleur}
  BW80    = 2;      {80 colonnes x 25 lignes, noir et blanc}
  CO80    = 3;      {80 colonnes x 25 lignes, couleur}
  Mono    = 7;      {80 colonnes x 25 lignes sur écran monochrome}
  Font8x8 = 256;    {mode EGA 43 lignes ou VGA 50 lignes}
  {couleurs de textes}
  Black        = 0;  {noir}                 Blue         = 1;  {bleu}
  Green        = 2;  {vert}                 Cyan         = 3;  {cyan}
  Red          = 4;  {rouge}                Magenta      = 5;  {magenta}
  Brown        = 6;  {marron}               LightGray    = 7;  {gris clair}
  DarkGray     = 8;  {gris foncé}           LightBlue    = 9;  {bleu clair}
  LightGreen   = 10; {vert clair}           LightCyan    = 11; {cyan clair}
  LightRed     = 12; {rouge clair}          LightMagenta = 13; {magenta clair}
  Yellow       = 14; {jaune}                White        = 15; {blanc}
  Blink        = 128;{clignotant}

var
  CheckBreak   : boolean;   {indicateur de Ctrl Break}
  CheckEof     : boolean;   {indicateur de Ctrl Z quand on lit un fichier a partir du clavier}
  CheckSnow    : boolean;   {indicateur de contrôle d'effet de neige sur les écrans CGA}
  DirectVideo  : boolean;   {indicateur d'accès direct à la mémoire de l'écran}
  LastMode     : word;      {valeur du mode d'affichage au début du programme}
  TextAttr     : byte;      {mémorise les attributs (couleurs) du mode texte}
  WindMin,
  WindMax      : word;      {coordonnées de la fenêtre courante}

procedure AssignCrt(var Fichier : text);
     Associe l'écran à un fichier de type "text".
procedure ClrEol;
     Efface la fin de la ligne courante à partir de la position du curseur.
procedure ClrScr;
     Efface l'écran et place le curseur en position (1, 1).
procedure Delay(T : word);
     Provoque une temporisation de T millièmes de seconde.
procedure DelLine;
     Efface la ligne où se trouve le curseur.
procedure GoToXY(c, l : byte);
     Place le curseur sur la colonne c de la ligne l.
procedure HighVideo;
     Initialise le mode d'affichage en forte intensité.
procedure InsLine;
     Insère une ligne vide à la position du curseur.
function KeyPressed : boolean;
     Renvoie la valeur true si une touche a été appuyée, false dans le cas contraire.
procedure LowVideo;
     Initialise le mode d'affichage en intensité normale.
procedure NormVideo;
     Réinitialise le mode d'affichage à la valeur qu'il avait au début du programme.
procedure NoSound;
     Interrompt le fonctionnement du haut-parleur.
function ReadKey : char;
     Lit un caractère en provenance du clavier.
procedure Sound(Freq : word);
     Active le haut-parleur avec une fréquence de vibration Freq exprimée en hertz.
```

```
procedure TextBackGround(Couleur : byte);
      Définit la couleur du fond de l'écran (entre 0 e 7).
procedure TextColor(Couleur : byte);
      Définit la couleur d'affichage des caractères.
procedure TextMode(Mode : integer);
      Définit le mode d'affichage à utiliser (voir constantes pré-définies dans l'unité Crt).
function WhereX : byte;
      Renvoie le numéro de la colonne sur laquelle est placé le curseur.
function WhereY : byte;
      Renvoie le numéro de la ligne sur laquelle est positionné le curseur.
procedure Window(x1, y1, x2, y2 : byte);
      Définit une nouvelle fenêtre d'affichage sur l'écran.
```

Unité DOS

```
const
  {valeurs des indicateurs de flag}
  FCarry     = $0001; {retenue}
  FParity    = $0004; {parité}
  FAuxiliary = $0010; {retenue intermédiaire}
  FZero      = $0040; {égal à zéro}
  FSign      = $0080; {signe}
  FOverflow  = $0800; {overflow}
  {modes des fichiers}
  fmClosed = $D7B0;
  fmInput  = $D7B1;
  fmOutput = $D7B2;
  fmInOut  = $D7B3;
  {attributs de fichiers}
  ReadOnly  = $01;    {attribut pour lecture seule}
  Hidden    = $02;    {fichier caché}
  SysFile   = $04;    {fichier du système opérationnel}
  VolumeId  = $08;    {nom de volume}
  Directory = $10;    {répertoire}
  Archive   = $20;    {fichier normal}
  AnyFile   = $3F;    {tous les fichiers}

type
  {type pour fichier "file" ou "file of"}
  FileRec = record
              Handle   : word;
              Mode     : word;
              RecSize  : word;
              Private  : array[1..26] of byte;
              UserData : array[1..16] of byte;
              Name     : array[0..79] of char;
            end;
  {types pour fichiers de type "text"}
  TextBuf = array[0..127] of char;
  TextRec = record
              Handle    : word;
              Mode      : word;
              BufSize   : word;
              Private   : word;
              BufPos    : word;
              BufEnd    : word;
              BufPtr    : ^TextBuf;
```

```
              OpenFunc  : pointer;
              InOutFunc : pointer;
              FlushFunc : pointer;
              CloseFunc : pointer;
              UserData  : array[1..16] of byte;
              Name      : array[0..79] of char;
              Buffer    : TextBuf;
            end;

  Registers = record {utilisé par Intr et MsDos}
                case integer of
                0 : (ax, bx, cx, dx, bp, si, di, ds, es, flags : word);
                1 : (al, ah, bl, bh, cl, ch, dl, dh : byte);
              end;
  DateTime = record {utilisé par UnPackTime, PackTime, etc...}
               Year, Month, Day, Hour, Min, Sec : word;
             end;
  SearchRec = record {utilisé par FindFirst et FindNext}
                Fill : array[1..21] of byte;
                Attr : byte;         {attribut de fichier}
                Time : longint;      {date et heure de la dernière mise à jour}
                Size : longint;      {taille du fichier}
                Name : string[12];   {nom et extension du fichier}
              end;
  DirStr  = string[67]; {nom du répertoire}
  NameStr = string[8];  {nom du fichier}
  ExtStr  = string[4];  {extension du nom de fichier}

var
  DosError : integer; {code d'erreur du DOS}

function DiskFree(D : word) : longint;
      Renvoie le nombre total d'octets disponibles sur le disque D. D peut prendre la valeur 0 pour
      indiquer le disque courant, 1 pour A:, 2 pour B:, 3 pour C:, etc..
function DiskSize(D : word) : longint;
      Renvoie la taille totale du disque D. Se reporter à la fonction précédente pour la valeur à
      donner à D.
function DosExitCode : word;
      Renvoie le code de sortie d'une routine (0 pour fin normale, 1 pour Ctrl-C, etc..).
function DosVersion : word;
      Renvoie le numéro de la version du système opérationnel; Lo(DosVersion) étant la version et
      Hi(DosVersion) la sous-version.
function EnvCount : word;
      Renvoie le nombre de chaînes de caractères comprises dans la zone d'environnement du DOS.
function EnvStr(I : word) : string;
      Renvoie une chaîne de caractères extraite de l'environnement du DOS.
procedure Exec(Prog, Param : string);
      Exécute le programme Prog, éventuellement avec des paramètres de ligne de commande, sans
      sortir du programme en cours.
function FExpand(NomFichier : PathStr) : PathStr;
      Renvoie le nom complet du fichier : unité:\répertoires\nom_fichier.
procedure FindFirst(Path : string; Attribut : word; var Fic : SearchRec);
      Renvoie le premier fichier du répertoire Path qui satisfait aux attributs de recherche. Voir
      la description des constantes d'attributs et le type SearchRec de l'unité DOS.
procedure FindNext(var Fic : SearchRec);
      Renvoie le prochain fichier qui satisfait aux conditions utilisées lors de l'appel de
      FindFirst.
function FSearch(NomFichier : PathStr; ListeRep : string) : PathStr;
      Recherche un fichier dans une liste de répertoires.
```

```
procedure FSplit(NomFichier : PathStr; var Rep : DirStr; var Nome : NameStr; var Ext : ExtStr);
      Divise le nom d'un fichier en nom de répertoire, nom et extension du fichier.
procedure GetCBreak(var Break : boolean);
      Renvoie l'état actuel (true ou false) du détecteur de Ctrl Break.
procedure GetDate(var Annee, Mois, Jour, JourSemaine : word);
      Renvoie la date courante du système opérationnel.
procedure GetDir(D : byte; var Rep : string);
      Renvoie le nom du répertoire courant du disque D.
function GetEnv(VarEnv : string) : string;
      Renvoie la valeur de la variable d'environnement du DOS passée comme paramètre.
procedure GetFAttr(var Fichier : file, file of ou text; var Attrib : word);
      Renvoie les attributs du fichier (voir constantes d'attributs de l'unité DOS).
procedure GetFTime(var : Fichier : file, file of ou text; Date : longint);
      Renvoie la date et l'heure de dernière mise à jour du fichier. Faire appel à UnPackTime pour
      décodifier la variable Date.
procedure GetIntVec(Int : byte; var Vecteur : pointer);
      Renvoie la valeur du vecteur de l'interruption Int.
procedure GetTime(var Heure, Minutes, Secondes, Centièmes : word);
      Renvoie l'heure courante utilisée par le système opérationnel.
procedure GetVerify(var Ver : boolean);
      Renvoie la valeur (true ou false) de l'indicateur VERIFY du DOS.
procedure Intr(Int : byte; var Regs : registers);
      Exécute l'interruption Int. Voir la définition du type "registers" dans l'unité DOS.
procedure Keep(Code : word);
      Stoppe l'exécution du programme et retourne au système opérationnel, mais en laissant le
      programme résident en mémoire.
procedure MsDos(var Regs : Registers);
      Exécute une fonction du DOS. Voir la définition du type "registers" dans l'unité DOS.
procedure PackTime(var T : DateTime; var DateHeure : longint);
      Convertit un temps de type DateTime en longint.
procedure SetCBreak(var Break : boolean);
      Initialise à true ou false l'état du détecteur de Ctrl-Break.
procedure SetDate(Annee, Mois, Jour : word);
      Initialise la date utilisée par le système opérationnel.
procedure SetFAttr(var Fichier :  file, file of ou text; Attribut : word);
      Définit les attributs d'un fichier. Voir les constantes d'attributs définies dans l'unité DOS.
procedure SetFTime(var Fichier : file, file of ou text; DateHeure : longint);
      Définit la date et l'heure de la dernière mise à jour du fichier.
procedure SetIntVec(Int : byte; Vecteur : pointer);
      Définit la nouvelle valeur du vecteur de l'interruption Int.
procedure SetTime(Heure, Minutes, Secondes, Centiemes : word);
      Initialise l'heure utilisée par le système opérationnel.
procedure SetVerify(Ver : boolean);
      Définit la valeur (true ou false) de l'indicateur VERIFY du DOS.
procedure SwapVectors;
      Echange le contenu des pointeurs SaveIntxx de l'unité System avec le contenu actuel des
      vecteurs d'interruption.
procedure UnPackTime(DateHeure : longint; var T : DateTime);
      Convertit un temps de type longint en temps de type DateTime.
```

Unité PRINTER

```
var
  lst : text;
```

Annexe II

Messages d'erreur à l'exécution

2 File not found (Fichier non trouvé)

Retourné par *Reset, Append, Rename* ou *Erase* si le nom de fichier ne correspond pas à un fichier existant.

3 Path not found (Chemin non trouvé)

Retourné par *Reset, Append, Rename* ou *Erase* si le nom de fichier contient un nom de répertoire invalide ou si le répertoire n'existe pas.

Retourné par *ChDir, MkDir* ou *RmDir* si le répertoire n'est pas valide ou n'existe pas.

4 Too many open files (Trop de fichiers ouverts)

Retourné par *Reset, Rewrite* ou *Append,* si trop de fichiers sont déjà ouverts dans le programme.

5 File access denied (Accès interdit au fichier)

Retourné en plusieurs occasions :

- en essayant d'écrire dans un fichier ouvert seulement pour la lecture, ou protégé ;
- en essayant de lire dans un fichier ouvert seulement pour l'écriture ;
- en essayant de créer un répertoire qui existe déjà ;
- en essayant d'effacer un répertoire non vide, ou qui n'existe pas ou qui est la racine d'un volume ;
- dans d'autres situations moins fréquentes.

6 Invalid handle file (Identification interne du fichier invalide)

Signifie que la variable de type *file* a été corrompue.

12 Invalid file access code (Type d'accès invalide pour le fichier)

Retourné par *Reset* ou *Append* si le type d'accès est invalide.

15 Invalid drive number (Numéro d'unité invalide)

Retourné par *GetDir* si le numéro de l'unité n'est pas correct.

16 Cannot remove current diretory (Le répertoire courant ne peut pas être détruit)

Retourné par *RmDir* quand on tente de supprimer le répertoire courant.

17 Cannot rename across drives (Impossible de renommer en changeant d'unité de disque).

Retourné par *Rename* quand on essaie de déplacer le fichier d'une unité de disque vers une autre.

100 Disk read error (Erreur en lecture du disque)

Retourné par *Read* quand on essaie de lire au-delà de la fin du fichier.

101 Disk write error (Erreur en écriture du disque)

Retourné par *Close, Write, WriteLn, Flush* quand le disque est plein.

102 File not assigned (Fichier non associé)

Le fichier n'a pas été associé à un nom de fichier physique par la procédure Assign.

103 File not open (Fichier fermé)

Tentative d'écriture sur un fichier non ouvert.

104 File not open for input (Le fichier n'est pas ouvert en lecture)

Tentative de lecture sur un fichier qui n'a pas été ouvert en lecture.

105 File not open for output (Le fichier n'est pas ouvert pour écriture)

Tentative d'écriture sur un fichier ouvert pour lecture seule.

106 Invalid numeric format (Nombre invalide)

Retourné par *Read* ou *ReadLn* quand la valeur à lire dans le fichier ne correspond pas au format numérique espéré.

Les erreurs 150 à 162 correspondent aux erreurs critiques du DOS :

150 Disk is write-protected (Disque protégé en écriture)
151 Unknown unit (Unité de disque inconnue)
152 Drive not ready (Unité de disque non prête)
153 Unknown command (Commande invalide)
154 CRC error in data (Erreur de lecture des données)
155 Bad drive request structure length (Taille invalide de la structure de communication avec l'unité de disque)
156 Disk seek error (Erreur de positionnement des têtes de lecture)
157 Unknown media type (Type d'unité inconnu)
158 Sector not found (Secteur de données non trouvé)
159 Printer out of paper (Imprimante sans papier)
160 Device write fault (Erreur d'écriture sur le disque)
161 Device read fault (Erreur de lecture sur le disque)
162 Hardware failure (Erreur de hardware)

200 Division by zero (Division par zéro)
201 Range check error (Donnée hors intervalle)

Retourné, si le programme a été compilé avec la directive $R+, quand :

- l'indice d'un tableau est hors des limites du tableau ;
- on tente d'attribuer à une variable une valeur incompatible avec les valeurs autorisées ;
- on tente de transmettre à une routine une valeur incompatible avec le type de paramètre.

202 Stack overflow error (Débordement de la pile)

Retourné, quand le programme a été compilé avec la directive $S+, quand l'espace disponible sur la pile est insuffisant pour allouer les variables locales d'une routine.

203 Heap overflow error (Débordement du tas)

Retourné par *New* ou *GetMem* quand il n'y a pas suffisamment d'espace sur le tas pour allouer une nouvelle variable.

204 Invalid pointer operation (Opération invalide sur un pointeur)

Retourné par *Dispose* ou *Freemem* si le pointeur passé en paramètre vaut nil ou s'il pointe vers une adresse non comprise dans le tas, ou si la table des fragments ne peut être agrandie.

205 Floating point overflow (Valeur réelle trop grande)

Un calcul sur des nombres réels conduit à un résultat plus grand que la limite autorisée pour le type de réel considéré.

206 Floating point underflow (Valeur réelle trop petite)

Un calcul sur des nombres réels conduit à un résultat plus petit que la précision autorisée pour le type de réel considéré. La valeur zéro est donnée comme résultat.

207 Invalid floating point operation (Calcul invalide sur des réels)

Retourné si :

- la valeur du paramètre de *Trunc* ou *Round* est en dehors de l'intervalle autorisé pour le type *longint* (-2147483648 et +2147483647) ;
- le paramètre de *Sqrt* est négatif ;
- le paramètre de *Ln* est négatif ;
- un débordement s'est produit sur la pile du 80x87.

208 Overlay manager not installed (Le gestionnaire d'overlay n'a pas été installé)

L'appel à *OvrInit* n'a pas été fait ou a échoué.

209 Overlay file error (Erreur de lecture dans le fichier des overlays)

L'extraction d'un module placé en *overlay* n'a pas pu être réalisée.

210 Object not initialized (Variable de type objet non initialisée)

Dans un programme qui a été compilé avec l'option $R+, on tente d'appeler une méthode virtuelle avant que l'instance de l'objet ait été initialisée par son constructeur.

Les erreurs de numéros 1000 à 1005 sont retournées par le Turbo-Access.

1000 Record size is greater than MaxDataRecSize (Taille d'enregistrement plus grande que MaxDataRecSize)

Pour un fichier séquentiel indexé de votre application, la taille de l'enregistrement de données est supérieure à la valeur de MaxDataRecSize déclarée lors de la configuration de l'unité TACCESS par TaBuild. Diminuer la taille de l'enregistrement ou reconfigurer TACCESS, ce qui peut obliger à réorganiser les autres fichiers séquentiels indexés (voir chapitre III).

1001 Record size is too small (Taille de l'enregistrement trop petite)

La taille de l'enregistrement est inférieure à 18 octets.

1002 Key length is greater then MaxKeyLen (La taille de la clé est supérieure à MaxKeyLen)

Pour un fichier séquentiel indexé de votre application, la taille de la clé est supérieure à la valeur de MaxKeyLen déclarée lors de la configuration de l'unité TACCESS par TaBuild. Diminuer la taille de la clé ou reconfigurer TACCESS, ce qui peut obliger à reconfigurer les autres fichiers séquentiels indexés (voir chapitre III).

1003 Data file created with different record size (Fichier de données créé avec une taille d'enregistrement différent)

Le fichier existe déjà et vous essayez d'y accéder avec une taille d'enregistrement différente, soit parce que la description du fichier a changé, soit parce que la version de TACCESS n'est plus la bonne. Corriger le programme ou détruire ou réorganiser le fichier.

1004 Index file created with different key or page size (Fichier index créé avec une taille de clé, ou de page, différente)

L'index existe déjà et vous essayez d'y accéder avec une clé (ou une page) de taille différente, soit parce que la description de la clé a changé, soit parce que la version de TACCESS n'est plus la bonne. Corriger le programme ou détruire ou réorganiser le fichier.

1005 Not enough memory for page stack (Mémoire insuffisante pour la page de clés)

Augmenter la taille du tas (avec la directive $M) ou reconfigurer l'unité TACCESS avec une taille de page plus petite (moins de clés dans chaque page).

Cochonneau Gérard (1990)

Cours de programmation avançée en Pascal

Paris : ORSTOM, 129 p. multigr.