P.S.I.

LANGAGES

TURBO PASCAL SUR AMSTRAD

LANGAGES

TURBO PASCAL SUR AMSTRAD

CONNAISSEZ-VOUS TOUTE LA COLLECTION AMSTRAD CHEZ P.S.I. ?

Pour les Amstrad CPC 464, 664 et 6128 :

Initiation :

— La découverte de l'Amstrad - Daniel-Jean David
— Exercices en Basic pour Amstrad - Maurice Charbit

Programmation BASIC :

— 102 programmes pour Amstrad - Jacques Deconchat
— Amstrad en famille - Jean-François Sehan
— Super jeux Amstrad - Jean-François Sehan
— Super générateur de caractères sur Amstrad - Jean-François Sehan
— Photographie sur Amstrad et Apple II - Pierrick Moigneau et Xavier de la Tullaye

Maîtrise du BASIC :

— Basic Amstrad, 1. Méthodes pratiques - Jacques Boisgontier et Bruno Césard
— Basic Amstrad, 2. Programmes et fichiers - Jacques Boisgontier
— Basic Plus, 80 routines sur Amstrad - Michel Martin
— Périphériques et fichiers sur Amstrad - Daniel-Jean David
— Amstrad en musique - Daniel Lemahieu

Assembleur :

— Assembleur de l'Amstrad - Marcel Henrot

Système :

— Clefs pour Amstrad, 1. Système de base - Daniel Martin
— CP/M Plus sur Amstrad 6128 et 8256 - Yvon Dargery
— Clefs pour Amstrad, 2. Système disque - Daniel Martin et Philippe Jadoul

A paraître :

— Intelligence artificielle : langage et formes sur Amstrad - Thierry Lévy-Abegnoli et Olivier Magnan
— Animation graphique sur Amstrad - Gilles Fouchard et Jean-Yves Corre
— Simulation et intelligence artificielle sur Amstrad - René Descamps
— Clefs pour Amstrad 8256 - Eric Baumarti

CONNAISSEZ-VOUS LA COLLECTION LANGAGE PASCAL AUX ÉDITIONS DU P.S.I. ?

— La programmation des jeux de réflexion - Louis Jardonnet
— Bibliothèque scientifique en Pascal - Hervé Haut
— Pascal UCSD sur Apple II, tomes 1 et 2 - Jacques Rouault et Patrice Girard
— Méthodes de calcul numérique, tomes 1 et 2 - Claude Nowakowski
— Turbo Pascal sur IBM/PC - Pierre Brandeis et Frédéric Blanc
— Programmer en Pascal - Daniel-Jean David et Jean-Luc Deschamps

A paraître :

— Introduction à l'informatique de gestion pour le turbo Pascal - Xavier Comtesse et Augusto Cosatti

Pour tout problème rencontré dans les ouvrages P.S.I.
vous pouvez nous contacter au numéro ci-dessous :

Numéro Vert/Appel Gratuit en France

05 21 22 01

(Composer tous les chiffres, même en région parisienne)

ISBN 2-86595-310-6

LANGAGES

TURBO PASCAL SUR AMSTRAD

PIERRE BRANDEIS
ET FRÉDERIC BLANC

ÉDITIONS DU P.S.I.
1986

Frédéric Blanc prépare une maîtrise d'informatique (M.I.A.G). Après avoir écrit plusieurs ouvrages sur le langage Assembleur de divers ordinateurs familiaux (Oric, MO5-TO7/70, MSX), il se consacre à présent à des ordinateurs plus professionnels (Macintosh, IBM-PC) ainsi qu'à des langages plus "évolués" (Pascal, C, Lisp, Basic). Il collabore également à plusieurs revues du groupe Tests et s'apprête à developper un logiciel à tendance ludique sur Macintosh.

Pierre Brandeis est musicien professionnel. Il est venu à l'informatique au début de l'année 83. Il s'intéresse d'abord à l'Oric-1 pour lequel il écrit un ouvrage sur le MSX. Puis, il franchit un échelon avec Macintosh et Pascal, sans pour autant abandonner son métier d'origine. Il collabore à la revue *L'Ordinateur Individuel*.

SOMMAIRE

INTRODUCTION

Pascal a longtemps été considéré comme un langage difficile ou du moins long à mettre en œuvre. C'était assez vrai jusqu'à l'arrivée de Turbo-Pascal. L'éditeur, le compilateur et le debogueur étaient rarement présents en mémoire ensemble, faute de place. On perdait beaucoup de temps avec les nombreux rechargements de ces éléments, nécessités par les incessantes allées et venues entre l'éditeur et le debogueur, (sans parler du temps parfois très long de la compilation elle même), lors de la mise au point des programmes. De plus, dans la majorité des cas, la compilation produisait un "pseudo-code" qui devait lui-même être interprété au moment de l'exécution, ce qui ralentissait forcément quelque peu celle-ci.

Cette lourdeur était donc préjudiciable à l'essor de Pascal. Pourtant ce langage est intéressant à plus d'un titre : il est d'un usage général, s'adresse à tous les types d'applications (calculs numériques, manipulation des chaînes de caractères et souvent même le graphisme), tout en étant beaucoup plus puissant et rapide que d'autres langages évolués comme le Basic (grâce à la récursivité par exemple) ou beaucoup plus lisible et facile à apprendre (il suffit de voir un programme écrit en Forth pour s'en convaincre).

L'arrivée de Turbo-Pascal remet tout ceci radicalement en cause! La plupart des critiques formulées à l'encontre de Pascal tombent en effet d'elles-mêmes face à Turbo-Pascal : L'éditeur, le compilateur et le débogueur sont intégrés en un seul programme. Il suffit de taper une seule lettre au clavier pour passer instantanément de l'un dans l'autre. Le compilateur est d'une rapidité surprenante : il compile environ 5000 lignes à la minute ! Le confort du programmeur est donc pratiquement aussi grand qu'en Basic : il ne manque guère que le mode direct, évidemment impossible dans un langage compilé.

Mais ce n'est pas tout ! Le produit de la compilation est du véritable code machine Z80 : cela signifie que le compilateur traduit votre programme Pascal en codes machine ! Côté rapidité, on peut difficilement faire mieux.

Cela n'empêche pourtant pas que toutes les spécificités originelles de Pascal aient été gardées : types définis par l'utilisateur, fonctions, procédures, données simples ou structurées, enregistrements, fichiers, pointeurs, et j'en passe, ont non seulement été conservés, mais ont été très souvent étendus par rapport au Pascal standard, en augmentant encore la puissance du langage.

Quant à l'implantation du programme terminé, elle est on ne peut plus simple: Turbo-Pascal produit des fichiers .COM directement éxécutables sous CP/M, sans même que Turbo ne soit présent sur la disquette d'exécution....

Confort du programmeur, rapidité d'exécution, puissance énorme, facilité de mise en route, que demander de plus à un langage ? Turbo-Pascal est la voie royale du développement, et on peut sans crainte envisager d'écrire avec lui des applications professionnelles.

CHAPITRE 1

QU'EST-CE QUE TURBO-PASCAL?

Pour pouvoir répondre à cette question, il faut d'abord présenter le langage Pascal de manière générale. Pascal dérive du langage Algol, en service sur de gros ordinateurs. Il a été développé dans le but de permettre un apprentissage facile de la programmation. A cette fin, on a défini un nombre restreint de commandes de base, faciles à retenir et à comprendre, qu'on pourra combiner de multiples façons pour effectuer des tâches très diverses.

BASIC ET PASCAL

Il est difficile de comparer Basic et Pascal, car ce sont deux langages très différents l'un de l'autre. Basic est un langage **interprété**, tandis que Pascal est **compilé**. Qu'est-ce que cela signifie?

Le microprocesseur, cœur de l'ordinateur, ne sait de toute façon se servir que des codes binaires (suites de 0 et de 1) pour lesquels il a été construit. Il est incapable de comprendre quoi que ce soit à un langage évolué, proche du nôtre. Comme nous autres humains sommes peu enclins à utiliser le binaire, et que nous préférons de loin nous servir justement de notre langage naturel, il faut bien qu'il y ait un trait d'union pour que la machine puisse comprendre l'homme. Ce trait d'union peut prendre deux formes: interpréteur ou compilateur. Dans les deux cas, on écrit le programme sous la forme d'un texte qu'on appelle programme source ou texte source ou simplement source. Le source obéit aux règles syntaxiques propres à chaque langage, mais demeure assez proche du nôtre (ou plutôt de celui de nos amis anglo-saxons) pour qu'il soit facile à écrire. La différence entre interprétation et compilation intervient ensuite.

Basic, langage interprété

Lorsqu'un langage est interprété (prenons l'exemple du Basic que tout le monde connaît), il est traduit au fur et à mesure de son exécution en codes machine que le microprocesseur peut exécuter. C'est, bien sûr, le travail de l'interpréteur de faire cette traduction simultanée. En fait, l'interpréteur est lui-même un programme exécuté par le microprocesseur, et la traduction n'est pas vraiment simultanée: il examine une commande dans le texte source, puis, une fois qu'il l'a reconnue en la cherchant dans une table contenant toutes les commandes permises et leur adresse d'exécution, il en vérifie la syntaxe. S'il n'y a pas de faute, il exécute la routine correspondante (non sans lui avoir le plus souvent transmis des paramètres de façon plus ou moins simple). Il lui faut revenir ensuite au source pour recommencer ensuite tout ce processus pour la commande suivante, et ainsi de suite jusqu'à la fin du programme: il a donc besoin de numéros de lignes pour ne pas perdre la trace de l'endroit où il doit continuer.

De même, il existe en Basic une table des variables ; à chaque variable qu'il rencontre, l'interpréteur parcourt la table en recherchant le nom de cette variable pour trouver sa valeur. Si le nom n'est pas trouvé, il s'agit d'une nouvelle variable à ajouter à la table : elle y est stockée avec sa valeur. On voit que si on veut limiter le temps de recherche et la taille mémoire de la table, on est obligé de limiter aussi la longueur des noms de variables.

Si l'on réfléchit au fait que l'ordinateur est particulièrement utile dans les tâches répétitives, on s'aperçoit que finalement un temps énorme est perdu inutilement à rechercher et à vérifier un grand nombre de fois les instructions (puiqu'elles viennent déjà d'être trouvées et vérifiées). Prenons l'exemple d'une boucle du type :

```
1010 FOR I = 1 TO 30000
1020 NEXT I
```

L'interpréteur, après avoir "compris" FOR et assigné la valeur 1 à la variable I, devra **à chaque tour de boucle** rechercher à nouveau la variable I dans la table pour l'incrémenter, avant de vérifier si elle a atteint la valeur 30000. Cela représente beaucoup de travail, donc de temps : ce programme tourne sur Amstrad en 34 secondes environ.

Pascal, langage compilé

On peut se demander s'il ne serait pas possible de faire toutes ces recherches une fois pour toutes. On gagnerait ainsi beaucoup de temps ensuite. C'est précisément ce qui se passe dans un langage compilé : le programme est traduit en entier avant son exécution. Toutes les recherches se font une seule fois: les commandes sont directement traduites en l'adresse de leur routine; les variables sont changées en un "pointeur" qui contient leur adresse. C'est ce qu'on appelle

compilation, et cette fois c'est un compilateur qui se charge de ce travail. Reprenons le même petit programme que ci-dessus, en Turbo-Pascal cette fois :

```
var i:integer;
begin
for i := 1 to 30000 do;
end.
```

Une fois compilé, il tient sur 57 octets de codes machine : à l'exécution, plus de table, plus de recherche, simplement 30 000 boucles.

Plus besoin non plus de numéros de lignes: la compilation suit le texte source du début à la fin; quant à l'exécution, elle ignore purement celui-ci et ne se sert que du code produit par la compilation, qu'on appelle code objet. Notre petit exemple tourne maintenant en 1,3 secondes environ, soit plus de 26 fois plus vite !

Le bonheur des uns...

Il doit pourtant bien y avoir un revers à la médaille, nul n'étant parfait ici bas! Ce revers se présente sous plusieurs formes : en premier lieu, le mode direct, où l'on peut taper une commande au clavier et la voir aussitôt exécutée, n'est pas possible, puisqu'il faut toujours passer par la compilation pour produire le code exécutable ; en second lieu, la compilation elle-même prend un certain temps, surtout si le programme est long, et à la moindre modification, il est nécessaire de recompiler le programme entier. De plus, dans de nombreuses versions de Pascal, le code produit par la compilation n'est qu'un pseudo-code, le "P-code", qui doit lui-même être interprété à l'exécution; cela pour faciliter l'adaptation de Pascal à différentes machines. On perd ainsi une partie des avantages de la compilation. En troisième lieu, il faudra déclarer toutes les variables et tous les "objets" utilisés dans le programme avant leur utilisation; cela pour permettre au compilateur de savoir quelle place il doit leur réserver en mémoire. C'est une grande contrainte par rapport aux habitudes d'écriture prises avec l'interpréteur Basic ; elle va cependant dans le sens de l'organisation structurée de la pensée, favorable à une programmation efficace, voulue par les concepteurs de Pascal.

Un autre argument était souvent opposé à Pascal avant l'arrivée de Turbo-Pascal: le Basic est souvent résident sur les micro-ordinateurs, et même lorsqu'il ne l'est pas, il tient en un seul programme : l'éditeur qui permet d'écrire le programme, l'interpréteur, le vérificateur de syntaxe, les routines qui forment les commandes, le lanceur de programme (RUN), les messages d'erreurs, tout cela est présent en mémoire, souvent en ROM. Le passage de l'un à l'autre de ces éléments se fait de manière transparente, sans même qu'on s'en rende compte. Le cas de Pascal est en général différent. L'absence de mode direct rendant difficile son implantation en tant que langage résident, il doit être chargé en RAM depuis une disquette. Or, jusqu'à une date récente, la place mémoire en RAM était comptée, car les Ko étaient très chers. On était, par conséquent, obligé de séparer les différents constituants du langage et de ne charger que l'élément dont on avait besoin : éditeur, compilateur,

lanceur, débogueur (aide à la mise au point). Cela permettait d'ailleurs d'utiliser n'importe quel traitement de texte comme éditeur, pourvu qu'il fournisse un fichier au standard ASCII accepté par le compilateur. Lors de l'écriture et de la mise au point des programmes, on devait faire de nombreux va-et-vient entre ces divers éléments, en perdant beaucoup de temps en chargements, sauvegardes et rechargements. Tout cela donnait une réputation de lourdeur à Pascal, et le programmeur amateur hésitait à s'y frotter. C'est alors...

...ne fait pas toujours le malheur des autres!

C'est alors qu'apparaît non pas un cavalier surgi de la nuit, mais plus simplement Turbo-Pascal. Celui-ci, tout en étant une version améliorée du Pascal standard, lève d'un coup tous ces obstacles. Il offre tous les éléments nécessaires à une mise au point facile des applications, regroupés en un seul programme chargé en une seule fois: plus d'allées et venues fastidieuses.

Son compilateur est si rapide que, dans le cas de programmes simples, la compilation est quasi immédiate, et qu'on retrouve pratiquement le mode direct: en frappant la lettre **R**, on provoque la compilation, suivie aussitôt de l'exécution. La mise au point, elle aussi, est extrêmement simple: en cas d'erreur dans le programme, celui-ci s'arrête, l'éditeur est rappelé, et l'endroit où l'erreur s'est produite est retrouvé automatiquement dans le texte source! Enfin, suprême supériorité, le code objet produit est réellement du code machine Z80, ce qui fait qu'il tourne pratiquement aussi vite que s'il était écrit en Assembleur...

Turbo-Pascal offre donc presque tous les avantages combinés des langages compilés et interprétés, sans en avoir les inconvénients. C'est un joli tour de force!

TURBO-PASCAL, LANGAGE TYPE, MODULAIRE, STRUCTURE

Pascal (et donc Turbo-Pascal) a été conçu en utilisant le principe de la pensée structurée : pour faire quelque chose de compliqué, on le décompose successivement en modules de plus en plus simples, jusqu'à ce que l'écriture d'une tâche élémentaire devienne évidente. La structure du langage est telle qu'il est parfaitement possible d'écrire un programme à plusieurs. On se répartit les tâches selon les affinités de chacun, et il suffit de se mettre d'accord sur les paramètres à échanger ou à passer entre les modules (que l'on appelle procédures et fonctions) pour pouvoir écrire ceux-ci indépendamment les uns des autres. Il ne restera plus, ensuite, qu'à écrire le programme principal, généralement très simple (sinon on l'aurait encore décomposé en modules plus petits) pour ordonner l'appel de ces modules. La structure du programme est alors très claire et encore renforcée par la manière dont on en présente le texte source : chaque nouveau module ou sous-module est décalé vers la droite par une marge plus importante (on dit qu'on indente chaque nouveau niveau).

Une autre caractéristique de Turbo-Pascal est d'être un langage typé. Cela signifie que le programmeur pourra déclarer (définir) ses propres types de variables, en plus des types préexistants. Les variables ainsi typées **ne pourront prendre que les valeurs admises par leur type**. Cela contribue à écrire sans difficulté des programmes clairs, lisibles, faciles à comprendre et à modifier. Bien entendu, toutes les explications nécessaires seront données à ce sujet dans cet ouvrage.

Enfin, avant d'entrer dans le vif du sujet, citons encore une caractéristique très puissante de Turbo-Pascal : il autorise la **récursivité**. Sans entrer ici dans le détail de ce qui fera l'objet plus loin d'une étude approfondie, disons simplement qu'un objet est récursif s'il fait appel à lui-même dans sa propre définition. C'est le cas des procédures et des fonctions (ces modules dont nous venons de parler): elles peuvent s'appeler elles-mêmes. C'est une grande supériorité sur le Basic, qui, lui, ne supporte jamais la recursivité.

Pascal est aussi récursif dans le sens où la structure même d'un programme se retrouve à l'intérieur d'une procédure ou d'une fonction. Ce sont en fait des programmes dans un programme.

Vous voyez que les limites sont repoussées très loin, et qu'on peut faire beaucoup de choses avec Turbo-Pascal.

CHAPITRE 2

TURBO-PASCAL EN ENVIRONNEMENT AMSTRAD

Turbo-Pascal a été écrit sous plusieurs systèmes d'exploitation : CP/M 86, MS-DOS (qui tourne sur IBM-PC), CP/M 2.2 et CP/M Plus (également appelé CP/M 3.0). Ces deux dernières versions sont souvent référencées comme CP/M 80 (pour Z80). Ce sont elles qui tournent sur Amstrad.

La première chose à faire est une copie de la disquette originale. C'est très important, **car il est possible par maladresse ou méconnaissance du système de perdre certaines possibilités**. Utilisez pour cela un utilitaire CP/M (DISCKIT ou PIP par exemple). Protégez auparavant en écriture votre disquette originale en découvrant le trou en haut à gauche de celle-ci (une erreur est si vite arrivée!). Rangez votre original en lieu sûr.

LES DIFFERENTS FICHIERS DE LA DISQUETTE

Sur la disquette que vous avez achetée se trouvent différents fichiers :

- **TURBO.COM** est l'éditeur/compilateur de Turbo-Pascal; c'est bien sûr le fichier principal.

- **TURBO.MSG** contient les messages d'erreurs. Vous pouvez ou non les inclure au démarrage de Turbo. Ils peuvent être traduits en français (ou dans une autre langue).

- **TURBO.OVR** sert quand on veut utiliser les fichiers Overlay (vous saurez plus tard ce que c'est).

- **CPCINST.COM** sert à installer les couleurs de l'écran pour l'éditeur en CP/M 2.2.

- **TINST.COM** sert à installer l'éditeur sur divers types de terminaux (terminal = clavier+écran). Vous ne l'utiliserez que si vous avez un Amstrad 6128.

- **TINST.DTA** contient les données relatives au terminal (seulement utile sur 6128).

- **TINST.MSG** contient les messages de TINST. Peut être traduit.

- **A LIRE** ou **READ.ME** sont des informations intéressantes à lire : modes d'emploi ou dernières améliorations.

- **INSTALL2.DOC** est un fichier à lire qui explique comment installer un terminal différent. Normalement, vous n'avez pas à vous en préoccuper.

- **MC.PAS** est le source du programme exemple MicroCalc. Il s'agit d'un petit tableur.

- **MC-MODØØ** à **MC-MODØ7** sont des fichiers qui seront inclus lors de la compilation de MC.PAS.

- **MC.HLP** est un fichier à lire contenant le mode d'emploi de MC.PAS.

- **MCDEMO.MCS** est un fichier utilisé par MC.

- **WINDOW.PAS** est un exemple de fenêtrage en Pascal (ne fonctionne qu'en CP/M2.2).

- **LISTER.PAS** est le source d'utilitaire de listage sur imprimante.

- **CMDLIN.PAS** est le source d'un exemple passage de paramètres directement depuis la ligne de commande CP/M au programme Turbo-Pascal.

COMMENT CHARGER TURBO-PASCAL

Deux cas peuvent se présenter :

Sur Amstrad 464 ou 664 avec lecteur de disquettes

Ces appareils ont 64 Ko de RAM, et n'acceptent que CP/M 2.2. Pas de problème dans ce cas : Turbo-Pascal est livré sur sa disquette préinstallé pour cette version. La marche à suivre pour entrer sous Turbo-Pascal est simple :

- démarrez CP/M sur une disquette système (pour cela tapez **|cpm** et **<RETURN>** sous Basic) ;
- une fois sous CP/M sortez la disquette système et insérez la disquette Turbo ;
- tapez **Ctrl-C** et **<RETURN>** pour indiquer à CP/M que vous avez changé de disquette ;
- tapez **TURBO** pour entrer sous Turbo-Pascal.

Vous n'avez donc pas à suivre les indications de l'appendice L du manuel Turbo. Vous n'avez pas, non plus, à utiliser le fichier TINST. Vous ne le pourriez d'ailleurs pas, car TINST ne fonctionne que sous CP/M Pus.

Pour simplifier les manipulations, il est plus pratique de copier CP/M, TURBO.COM et TURBO.MSG sur une même disquette qui pourra donc servir pour le démarrage. Vous y ajouterez les fichiers qui vous semblent utiles: CPCINST.COM et TURBO.OVR par exemple. Vous disposez d'environ 7,5 Ko pour le source et pour le code objet (moins 1,3 Ko si vous chargez les messages d'erreurs). Cela peut sembler peu, mais diverses méthodes vous permettront de produire des programmes plus longs (les fichiers inclus), d'autant plus que Turbo-Pascal n'est pas nécessaire en mémoire pour l'exécution d'un programme compilé. La technique des fichiers inclus est expliquée, d'une part dans le fichier ALIRE (ou READ.ME), et d'autre part dans un des chapitres de cet ouvrage.

Le programme CPCINST.COM de la disquette Turbo sert à changer les couleurs de l'affichage sous Turbo-Pascal. Pour le faire fonctionner, il suffit de taper **CPCINST** lorsqu'on est sous CP/M 2.2. Suivez ensuite les indications qu'il vous donne.

Sur Amstrad 6128

Vous pouvez le faire fonctionner soit sous CP/M 2.2, soit sous CP/MPlus, car vous disposez de 128 Ko de RAM. L'avantage de CP/M Plus est qu'il se charge dans le deuxième banc de RAM et vous laisse cette fois un peu plus de 30 Ko au total

pour le source et pour l'objet. Son inconvénient est qu'il est nécessaire d'installer l'éditeur avec TINST et qu'une fois cela effectué, Turbo-Pascal ne pourra plus fonctionner sous CP/M2.2, donc sur Amstrad 464 ou 664. Cela provient en partie de la manière différente qu'ont les deux versions de CP/M d'interpréter les codes de contrôle. Un affichage perturbé et/ou un "plantage" de la machine proviennent sans doute du fait que vous exécutez sous une version des programmes écrits sous l'autre.

Pour installer Turbo-Pascal sur Amstrad 6128, vous devez :

- soit suivre les indications ci-dessus, en démarrant sous CP/M 2.2 qui se trouve sur l'une de vos disquettes système. Dans ce cas, vous renoncez à utiliser la place mémoire supplémentaire que vous offre CP/M Plus ;

- soit utiliser CP/M Plus ;

  - assurez-vous qu'une copie de la disquette originale a bien été faite ;

  - démarrez CP/M Plus sur une disquette système (pour cela taper |**cpm** et **<RETURN>** sous Basic) ;

  - une fois sous CP/M sortez la disquette système et insérez la disquette Turbo;

  - tapez **TINST** pour lancer l'installateur. Cette opération n'est à faire que la première fois qu'on utilise la disquette, mais attention, cette disquette ne pourra plus fonctionner sous CP/M 2.2, donc sur Amstrad 464/664. Or, à l'heure où nous écrivons ces lignes, des extensions graphiques sont prévues pour Turbo-Pascal, et il se pourrait qu'elles ne fonctionnent que sous CP/M2.2. Il faut donc absolument conserver une disquette Turbo fonctionnant sous cette version.Une fois dans TINST, choisissez l'option **S**, puis le numéro **30** (l'Amstrad émule le terminal Zénith), puis **Q** pour quitter ;

  - tapez **TURBO** pour entrer sous Turbo-Pascal.

Vous devez maintenant avoir **deux** versions de Turbo-Pascal, en plus de votre original: une qui fonctionne sous CP/M Plus, et une qui tourne sous CP/M 2.2.

LES MESSAGES D'ERREURS

Une fois entré sous Turbo-Pascal, vous devez répondre à la question :

```
Include error messages?
```

Répondez **Y** (pour Yes) ou **N** (pour No). Sachez que les messages d'erreurs occupent 1,3 Ko de plus en RAM. Si vous les chargez, Turbo vous répond `loading TURBO.MSG`. Lorsqu'une erreur se produira, Turbo vous affichera le message

correspondant. Si vous ne les chargez pas, Turbo vous donnera seulement le numéro de l'erreur qui s'est produite ; vous devrez rechercher ce numéro dans la liste des erreurs qui se trouve en annexe 3.

Traduction des messages d'erreurs

A l'origine, les messages sont en anglais dans le fichier TURBO.MSG. Il est possible de les traduire en français. Utilisez l'éditeur Turbo pour modifier le fichier des messages. Les vingt-quatre premières lignes du fichier définissent des constantes de texte qui sont ensuite incluses dans les messages. Ces constantes sont des caractères de contrôle qui apparaissent en video inverse.

Pour traduire le fichier TURBO.MSG, commencez par traduire chaque message en entier (il n'est pas nécessaire que la traduction ait la même longueur que l'original); essayez ensuite de trouver le plus possible de mots communs entre eux. Redéfinissez les constantes en début de texte et incluez leurs identificateurs dans les messages. La seule chose qui ne peut être changée est le nombre maximal de constantes et, bien sûr, la relation entre le numéro et la signification de l'erreur. Les caractères de contrôle, c'est-à-dire les identificateurs sont entrés en tapant **Ctrl-P-Ctrl-*caractère***.

LE MENU PRINCIPAL DE TURBO-PASCAL

Quand vous entrez sous Turbo, le menu principal apparaît. Il prend la forme suivante :

```
Logged drive: A

Work file:

Main file:

Edit              Compile      Run        Save

eXecute           Dir          Quit       compiler Options

Text:             0 bytes

Free:             XXXX bytes

>
```

XXXX est le nombre d'octets libres en mémoire et varie suivant la version CP/M et la présence ou l'absence des messages d'erreurs.

Les commandes du menu principal

Les lettres en gras ci-dessus, qui sont en video inverse sur l'Amstrad sont les commandes disponibles sous Turbo. Pour exécuter une commande, il suffit de taper la lettre correspondante, lorsque vous avez la main, c'est-à-dire quand le signe "**>**" est affiché. Il peut arriver que le menu disparaisse quand vous travaillez ; tapez alors une lettre différente d'une commande pour le rappeler.

Examinons maintenant les divers choix disponibles :

- **L**: permet de changer de disque par défaut. Cette commande vous sert à changer de lecteur de disquettes, si vous en utilisez plusieurs, ou à changer de disquette si vous n'en avez qu'un. Après avoir tapé **L**, vient le message :

  ```
  New drive:
  ```

  Entrez le nom du nouveau disque, précédé éventuellement d'une lettre entre A et P référençant son lecteur. Cette opération est nécessaire si vous changez de disque en cours de travail, car il faut "prévenir" Turbo qu'une nouvelle disquette a été insérée ; si vous ne le faisiez pas, vous risqueriez le message d'erreur de CP/M 'Bdos error on A", et vous perdriez tout ce qui est présent en mémoire à ce moment.
  Si vous n'avez qu'un seul lecteur et que vous voulez changer de disquette en cours de travail, retirez l'ancienne disquette après l'affichage du message `New drive:`, puis insérez la nouvelle et tapez **<RETURN>**. Vous pouvez alors voir que le disque tourne un instant: le système "reconnaît" la nouvelle disquette: il n'y aura pas d'erreur à la prochaine écriture. Sous CP/M plus, on peut simuler la présence de plusieurs lecteurs de disquettes en faisant précéder le nom du fichier d'une lettre entre A et P, qui identifiera le disque. Le système demande alors lui-même les échanges de disquettes nécessaires.

- **W**: choix du fichier de travail. Le fichier de travail est celui sur lequel fonctionneront les commandes **E**dit, **C**ompile, e**X**ecute, et **S**ave. C'est la première commande à taper quand vous entrez sous Turbo. Le message suivant est alors affiché :

  ```
  Work fike name:
  ```

  Entrez alors un nom légal de fichier: de 1 à 8 caractères, éventuellement suivi d'un point et d'une extension de 1 à 3 caractères. Si vous ne mettez pas de point ni d'extension, l'extension **.PAS** sera prise par défaut. Méfiez vous, certains fichiers peuvent avoir une extension .<espace> qui n'apparaît pas dans la commande **DIR**, mais qui existe quand même :

`FICHIER`	devient `FICHIER.PAS;`
`FICHIER.`	reste inchangé;
`FICHIER.PRG`	reste inchangé.

Les extensions **.BAK**, **.CHN**, **.CMD** et **.COM** et tous les nombres sont réservés au système (fichiers Overlay). Si vous avez auparavant édité un autre fichier qui n'a pas été sauvegardé, Turbo vous demande :

```
Workfile XXXX not saved. Save (Y/N)?
```

Répondez **Y** pour le sauver ou **N** pour le perdre.Une fois le nom entré, il est recherché sur le disque; s'il est présent, il est chargé en mémoire et peut être édité. S'il est absent, le message New File est affiché, et l'éditeur est appelé pour que vous puissiez l'écrire (CP/M2.2 seulement; en CP/M Plus il faut taper **E)**. Le nom que vous avez donné sera celui sous lequel il sera ensuite enregistré sur la disquette.

C'est avec cette commande que vous chargez également les fichiers documentaires à lire. Tapez ensuite **E** pour entrer dans l'éditeur (et donc les lire).

- **M**: définit le fichier principal (Main File). Lorsque vous travaillez avec des fichiers inclus (vous saurez plus tard ce que c'est), vous pouvez définir un fichier principal dans lequel les autres seront inclus. Le fichier de travail (Workfile) pourra alors être l'un des fichiers inclus, ce qui permettra de l'éditer, tout en gardant le même fichier principal.

  Lorsque vous lancez la compilation et que le fichier de travail et le fichier principal sont différents, ce dernier est chargé en mémoire; si une erreur survient, le fichier où l'erreur s'est produite est chargé, et l'erreur est trouvée dans le source. Une fois la correction faite et la compilation relancée, le fichier corrigé est automatiquement sauvegardé et le fichier principal rechargé.

- **E**: active l'éditeur. Le fichier de travail est affiché, et vous pouvez l'éditer. Les nombreuses commandes de l'éditeur sont expliquées ci-après.

- **C**: lance la compilation. Si vous avez spécifié un fichier principal, il est compilé, sinon ce sera le fichier de travail. La compilation s'effectue d'après l'option de compilation choisie (commande **O**): en mémoire ou sur disque (fichiers .COM ou .CHN). Par défaut, la compilation s'effectue en mémoire.

- **R**: provoque l'exécution d'un programme. Si le programme n'est pas encore compilé, la compilation s'effectue comme ci-dessus, puis le programme est lancé. C'est cette commande qui vous permet de faire tourner le programme que vous venez d'écrire ou de charger.

- **S**: sauvegarde le fichier de travail sur le disque. S'il y avait déjà une version de celui-ci sur le disque, elle sera renommée avec l'extension **.BAK**.

- **X**: permet l'exécution d'un autre programme depuis Turbo. Ce programme n'est pas forcément un programme créé par Turbo-Pascal. Le message :

  ```
  Program:
  ```

  vous invite à entrer le nom du programme à exécuter. A la fin de celui-ci, le contrôle est redonné à Turbo.

- **D**: affiche le contenu de la disquette. Le message :

  ```
  Dir mask:
  ```

  vous permet d'entrer le nom d'un lecteur de disque, le nom d'un lecteur suivi d'un nom de fichier, ou un masque contenant les caractères de remplacement * et **?**. Si vous tapez simplement **<RETURN>**, vous obtenez le catalogue complet de la disquette et la place restant sur le disque, sous réserve d'avoir informé le système du changement de disquette.

- **Q**: quitte Turbo et revient à CP/M. Si le fichier de travail a été édité, on vous demande si vous voulez le sauver avant de quitter.

- **O**: appelle le menu des options de compilation.

Les options de compilation

Lorsqu'on appelle les options de compilation depuis le menu principal, celui-ci fait place au menu de compilation. Les options sont au nombre de 3, et celle qui est en service est désignée par une flèche. On sélectionne une option en tapant une seule lettre :

- **M**: provoquera la compilation en mémoire. C'est l'option par défaut.

- **C**: provoquera la compilation sur disque. Le fichier résultant aura l'extension **.COM** (command). Il pourra être lancé depuis CP/M sans que la présence de Turbo soit nécessaire sur la disquette. Les programmes compilés de cette façon n'occupent pas de place en mémoire durant la compilation. Le code produit contient le programme et la librairie Turbo-Pascal.

- **H**: provoquera la compilation sur disque. Le fichier résultant aura l'extension **.CHN** (Chain). Le code produit contient seulement le code du programme, mais pas la librairie Turbo-Pascal. Il ne peut donc être exécuté que depuis un autre programme Turbo-Pascal, en utilisant la procédure **Chain** ou **Execute**. Ceci sera expliqué au chapitre 17.

- Lorsque vous choisissez **C** ou **H**, deux lignes supplémentaires vous indiquent l'adresse minimale de début et maximale de fin pour le code. Vous n'avez

normalement pas à les changer. Cependant, l'adresse de début peut être remontée si nécessaire pour réserver un espace libre (par exemple pour des variables absolues à partager entre plusieurs programmes chaînés ou en Overlay). C'est seulement utile pour de gros programmes, ne vous en inquiétez donc pas pour le moment. L'adresse de fin ne doit être baissée que si vous envisagez de faire tourner votre programme sur d'autre machines que l'Amstrad (ce qui est parfaitement possible, du moment que la machine envisagée supporte CP/M). Il peut alors être prudent de baisser cette adresse jusque vers C100 ou A100 si vous voulez tourner sous MP/M. Pour modifier ces adresses, tapez **S** pour Start (début)ou **E** pour End (fin), et entrez la nouvelle adresse. L'appui de **<RETURN>** conserve l'ancienne adresse.

- **Q**: quitte les options de compilation et revient au menu principal. L'option désignée par la flèche est prise en compte.

- **F**: recherche des erreurs à l'exécution des fichiers .COM ou .CHN (voir débogueur).

L'EDITEUR

L'éditeur est activé en tapant **E** dans le menu principal. On revient au menu principal après l'édition en tapant **Ctrl-K-D.** Il s'agit d'un éditeur plein écran très semblable au traitement de texte Wordstar, dont il reprend d'ailleurs les commandes d'édition (45 au total).

Entrez votre texte source normalement, en tapant **<RETURN>** pour aller à la ligne. La longueur maximale d'une ligne est de 127 caractères, mais il est rarement nécessaire d'aller jusque là : en Turbo-Pascal, le caractère saut de ligne est simplement un séparateur comme le blanc ou le caractère de tabulation. Vous pouvez donc aller à la ligne n'importe quand, sauf à l'intérieur d'une chaîne de caractères.

L'éditeur Turbo accepte les fichiers ASCII écrits avec d'autres traitements de texte. Si une ligne dépasse 127 caractères, Turbo insère un retour chariot et affiche un message.

Les programmes Pascal ont une présentation très claire, à base d'indentation (décalage des marges vers la droite) ; cette présentation est facilitée par l'éditeur qui est conçu pour elle.

Sur la ligne supérieure de l'écran, la ligne d'état, apparaissent des informations :

```
Line n        Col  m       Insert       Indent     X:FICHIER.FFF
```

- **Line** et **Col** donnent le numéro de la ligne et celui de la colonne où se trouve le curseur. Vous les verrez changer au fur et à mesure de vos déplacements sur l'écran.

- **Insert** indique que les caractères tapés au clavier viendront s'insérer dans le texte à l'endroit du curseur. Ce mode peut être changé par la touche **Copy** (ou commande **Ctrl-V**) en OverWrite, où les caractères tapés recouvrent ceux présents à l'écran. Lors de la première entrée de texte, le mode doit être **Insert** : OverWrite doit être réservé aux corrections.

- **Indent** indique que le mode d'indentation automatique est actif. Vous pouvez le désactiver par la commande **Ctrl-Q-I** (bascule). Lorsque ce mode est actif, le curseur ne revient pas en colonne 1 après un saut de ligne, mais juste en dessous du premier mot de la ligne au dessus. Si vous voulez commencer la ligne ailleurs, utilisez une commande de déplacement du curseur ou la touche <**TAB**>.

- **X:FICHIER.FFF** indique le disque, le nom et l'extension du fichier en cours d'édition.

Les commandes de l'éditeur

La plupart des commandes sous éditeur s'obtiennent en frappant un caractère de contrôle, c'est-à-dire en maintenant la touche **Control** appuyée tout en frappant une lettre. Cependant, certaines d'entre elles nécessitent deux frappes au clavier: on tape d'abord **Ctrl-Q** ou **Ctrl-K** ; le curseur disparaît, et le signe **^Q** ou **^K** s'affiche en haut à gauche de l'écran. On tape ensuite la deuxième lettre. Si vous changez d'avis après avoir frappé **Ctrl-Q** ou **Ctrl-K**, appuyez simplement sur <**RETURN**> et le curseur reprendra sa place.

Pour sortir

Ctrl-K-D	ramène au menu principal.

Les commandes de déplacement du curseur

L'éditeur est déjà installé sur Amstrad pour reconnaître les touches de déplacement curseur (flèches). De plus on a les commandes suivantes :

Ctrl-E	comme flèche en haut.
Ctrl-X	comme flèche en bas.
Ctrl-S	comme flèche à gauche.
Ctrl-D	comme flèche à droite.
Ctrl-flèche en haut (ou **Ctrl-R**)	remonte l'affichage d'une page.
Ctrl-flèche en bas (ou **Ctrl-C**)	descend l'affichage d'une page.

Ctrl-flèche à droite (ou **Ctrl-F**)	saute au début du mot suivant.
Ctrl-flèche à gauche (ou **Ctrl-A**)	saute au début du mot précédent.
Ctrl-W	déroule l'écran d'une ligne vers le bas.
Ctrl-Z	déroule l'écran d'une ligne vers le haut.

Pour mémoriser ces commandes, comparez l'emplacement des touches sur le clavier avec leur action : vous constaterez des similitudes.

Afin de se déplacer encore plus vite dans le texte, six autres commandes sont prévues ; elles ressemblent aux précédentes, mais sont faites de deux caractères successifs. Comparez-les avec les commandes ci-dessus, regardez le clavier, et vous arriverez vite à vous en souvenir.

Ctrl-Q-S	déplace le curseur à la première colonne.
Ctrl-Q-D	déplace le curseur à la fin de la ligne.
Ctrl-Q-E	déplace le curseur en haut de l'écran.
Ctrl-Q-X	déplace le curseur en bas de l'écran.
Ctrl-Q-R	déplace le curseur au début du fichier.
Ctrl-Q-C	déplace le curseur à la fin du fichier.
Ctrl-Q-P	remet le curseur à sa dernière position.

Il existe enfin deux commandes de déplacement du curseur au sein d'un bloc marqué (voir ci-dessous) :

Crtl-Q-B	déplace le curseur au début du bloc marqué.
Crtl-Q-K	déplace le curseur à la fin du bloc marqué.

Commandes d'insertion et de suppression

En premier lieu, la commande de rattrapage :

Ctrl-Q-L	annule les modifications d'une ligne tant que le curseur n'a pas quitté celle-ci.

Ensuite, l'insertion :

<Copy> ou **Ctrl-V**	bascule Insertion/Recouvrement (Insert/Overwrite).
Ctrl-N	insère une ligne à la position du curseur.
<TAB>	déplace le curseur d'une position d'une tabulation.Les positions de tabulation ne sont pas fixes, elles sont automatiquement fixées au dessous de chaque mot de la ligne

immédiatement au dessus du curseur. Cela permet une écriture facile des programmes Pascal, dans lesquels on a l'habitude d'aligner verticalement les débuts de phrases de même niveau.

Ctrl-Q-I bascule Indent actif/inactif. Quand il est actif, **<RETURN>** fait venir le curseur sous le premier caractère de la ligne immédiatement supérieure et non pas en début de ligne. Utilisez une commande de déplacement ou la touche **<TAB>** pour modifier cette position si nécessaire.

Puis les suppressions :

<DEL> supprime le caractère à gauche du curseur.
Ctrl-G supprime le caractère sous le curseur.
Ctrl-T supprime le mot à droite du curseur.
Ctrl-Y supprime la ligne où se trouve le curseur (**attention!...**).
Ctrl-Q-Y supprime définitivement la fin de la ligne du curseur.

Commandes de bloc

Un bloc est une quantité de texte quelconque, long de un caractère à plusieurs pages. Pour pouvoir le copier, le déplacer, le supprimer, le montrer, il faut d'abord définir un bloc (il ne peut y avoir qu'un seul bloc marqué à la fois) :

Ctrl-K-B marque le début d'un bloc.
Ctrl-K-K marque la fin d'un bloc.
Ctrl-K-T marque un mot seul comme un bloc.

Une fois le bloc marqué, il apparaît en vidéo inverse. Vous pouvez le remettre en vidéo normale sans pour autant qu'il cesse d'être marqué:

Ctrl-K-H bascule Montrer/Cacher le bloc marqué.

On peut ensuite manipuler le bloc :

Ctrl-K-C recopie à la position du curseur le bloc marqué.

Ctrl-K-V enlève le bloc marqué et l'insère à la position du curseur (déplacement).

Ctrl-K-Y supprime le bloc marqué sans possibilité de le restaurer (**attention!...**).

Ctrl-K-R lit un fichier sur la disquette et l'insère à la position du curseur, comme un bloc. La commande vous demande le nom du fichier à lire. En cas d'erreur, presser **<RETURN>** pour revenir dans le texte. Le bloc lu est marqué comme un bloc.

Ctrl-K-W écrit un bloc marqué sur la disquette. La commande vous demande quel nom donner au fichier créé; si ce nom existe déjà, vous êtes prévenu que vous allez l'écraser. L'extension **.PAS** est donnée par défaut.

Commandes de recherche d'occurrences

Ctrl-Q-F recherche une chaîne de caractères dans le texte. Entrez la chaîne à trouver, terminée par **<RETURN>** ou tapez **Ctrl-U** pour sortir. Si une recherche a déjà été faite, vous pouvez rappeler la chaîne avec **Ctrl-F**. La chaîne de recherche peut avoir jusqu'à 30 caractères, tous quelconques; si vous voulez y inclure un caractère de contrôle, il faut le faire précéder de **Ctrl-P**: **Ctrl-P-Ctrl-A** insèrera un **Ctrl-A**, qui peut d'ailleurs servir de caractère générique pour remplacer n'importe quel caractère à trouver. Après avoir entré votre chaîne, vous pouvez spécifier des options de recherche (**<RETURN>** si vous n'en voulez pas) **:**

B : rechercher en arrière vers le début du texte ;
U : ignorer majuscules et minuscules ;
W: rechercher des mots entiers seulement, c'est-à-dire ignorer les occurrences à l'intérieur d'un mot ;
n : (nombre) rechercher la n^e occurrence de la chaîne.

Les options sont tapées à la suite les unes des autres sans séparateur, par exemple W5U, et terminées par **<RETURN>**. Si la chaîne est trouvée, la recherche peut être répétée par **Ctrl-L**. La chaîne précédemment recherchée peut être rappelée avec la commande **Ctrl-F.**

Ctrl-Q-A recherche et remplace une chaîne par une autre. Vous devez entrer successivement la chaîne de recherche et celle de remplacement. Par défaut, une confirmation (`replace Y/N`) vous sera demandée à chaque occurrence de la chaîne de recherche. La commande fonctionne comme la précédente, à l'exception de **Ctrl-A** qui n'a pas de signification particulière dans la chaîne de remplacement. Vous disposez en plus de celles ci-dessus des options suivantes :

G : recherche et remplacement dans tout le texte, quelle que soit la position du curseur ;
N : remplacement sans confirmation ;
n : (nombre) représente ici le nombre d'occurrences à rechercher et à remplacer.

Ctrl-L recommence la dernière opération de recherche ou de recherche et remplacement avec les mêmes paramètres.

Enfin, les dernières commandes

Ctrl-U annule une opération en cours quand une entrée est demandée.

Ctrl-P préfixe pour entrer un caractère de contrôle dans le texte.

Modification des commandes de l'éditeur (CP/M Plus)

Vous venez de le voir, les commandes de l'éditeur sont nombreuses et puissantes. Mais ce n'est pas tout : sur Amstrad 6128, ou plus précisément sous CP/M Plus, vous pouvez les changer à votre convenance! Pour cela, revenez sous CP/M et lancez TINST. Choisissez l'option **C** (Commandes). Chacune des 45 commandes apparaît tour à tour, avec la commande installée, suivie d'une flèche. Tapez **<RETURN>** pour la laisser telle quelle ou tapez la nouvelle commande pour la changer. Si vous vous trompez, entrez simplement **B** à la ligne suivante; vous reviendrez alors à la ligne précédente, car il n'y a pas ici de possibilité de corriger une faute de frappe. Les flèches curseur et la touche **Copy** sont affichées comme des caractères accentués suivis de (+128) (cela correspond au code qu'elles produisent).

La plupart des commandes sont bien pensées à l'origine, mais la touche **<CLR>** n'est pas active. Vous pouvez l'installer en regard de la commande 26 "supprimer le caractère sous le curseur" (appuyez simplement dessus, puis sur **<RETURN>**). Il y a cependant un petit problème : cette touche produit en fait le code de contrôle **Ctrl-P**, et ce code est utilisé par la commande 45 "préfixe des codes de contrôle". Vous devez donc aussi changer cette commande, car vous ne pouvez évidemment pas laisser d'ambiguïté. Une bonne idée est d'installer la commande 45 avec le code **Ctrl-^**, puisque **"^"** est l'abréviation usuelle de **Ctrl**.

Il n'y a pas de porte de sortie à TINST, mais si vous vous perdez en cours de route, vous pouvez toujours faire un **Ctrl-Shift-ESC** qui vous ramènera sous Basic sans que rien ne soit changé. Vous aurez d'ailleurs toujours la possibilité de retaper toutes les commandes décrites ci-dessus.

LE COMPILATEUR

La compilation est l'opération qui traduit le texte source en code machine exécutable.

On lance la compilation (depuis le menu principal) soit en tapant **C**, soit en tapant **R**. Dans ce dernier cas, le programme est exécuté dès la fin de la compilation.

Le compilateur Turbo-Pascal est particulièrement rapide (plus de 5000 lignes à la minute...). Vous pourrez d'ailleurs le vérifier, car, la compilation lancée, il affiche le message :

```
Compiling
```

et les numéros des lignes sur lesquelles il passe, au fur et à mesure de sa progression à travers le texte source. La compilation s'effectue en une seule passe, ce qui accélère encore le processus. Pour finir, Turbo vous donne la longueur et les adresses du code et des données, ainsi que la place encore disponible en mémoire :

```
Code:          N bytes          (début-fin)
Free           N bytes          (début-fin)
Data           N bytes          (début-fin)
```

Si une erreur survient, la compilation s'arrête, le numéro de l'erreur est donné, ainsi que le message correspondant si les messages d'erreurs ont été chargés. Vous devez alors appuyer sur **<ESC>** ; l'éditeur est appelé et l'erreur est retrouvée dans le texte source: le curseur se positionne juste après le mot erroné. Corrigez, sortez de l'éditeur par **Ctrl-K-D**, et relancez la compilation. Lorsque celle-ci est terminée, le signe **">"** vous redonne la main (même s'il n'est pas visible à l'écran, vous êtes alors dans le menu principal; frappez une touche différente d'une commande pour le faire apparaître).

Les directives de compilation

Nous avons déjà étudié les différentes options de compilation, nous ne reviendrons pas dessus. Mais en plus de celles-ci, il existe des directives de compilation qui s'insèrent dans le texte source sous la forme **"{$*directive*}"**. Si vous ne connaissez pas encore Turbo-Pascal, vous ne pouvez pas comprendre à quoi elles servent. Pour cette raison, nous avons préféré en donner la liste en annexe. N'oubliez donc pas de lire celle-ci quand vous aurez progressé un peu, car les directives de compilation sont importantes : si un programme semble ne pas fonctionner, cela peut provenir de l'oubli d'une directive indispensable.

LE LANCEUR

On lance le programme par la commande **R**. S'il n'était pas déjà compilé, ou s'il a été édité depuis, la compilation s'effectue. Turbo affiche ensuite :

```
Running
```

et le programme démarre. A la fin de celui-ci, la main vous est rendue (le signe **">"** est affiché).

LE DEBOGUEUR

Un débogueur est un utilitaire d'aide à la mise au point des programmes. Il est intégré dans Turbo-Pascal : quand vous exécutez un programme compilé en mémoire et qu'une erreur survient à l'exécution, l'éditeur est rappelé automatiquement, et l'erreur retrouvée dans le fichier source (on peut donner au passage un coup de chapeau au concepteur de Turbo-Pascal!). Cependant, ceci n'est pas possible si le programme est un fichier .COM ou .CHN. Dans ce cas, seul un code d'erreur est affiché, avec la valeur du compteur programme (PC) au moment de l'erreur. Pour retrouver dans le source l'endroit de l'erreur, tapez **F** (pour Find=chercher) dans le menu Option de compilation. Entrez ensuite la valeur de PC donnée par le message d'erreur. L'endroit de l'erreur est alors recherché, et le curseur s'y trouvera quand vous passerez dans l'éditeur (taper **E**). Une liste des messages d'erreur figure en annexe 3.

LES GRAPHISMES

La version de base de Turbo-Pascal sur Amstrad ne comprend pas d'instructions graphiques. Une extension sera bientôt commercialisée; à l'heure où vous lirez ces lignes, elle sera sans doute disponible. En attendant, il est possible, si vous connaissez bien votre machine, d'écrire des procédures graphiques. Ceci n'est relativemant simple que sous CP/M 2.2, car il faut faire des sauts aux points d'entrée des routines graphiques dans le premier bank de RAM. Vous devrez utiliser les procédures **InLine**, qui sont en fait des sous-programmes en codes machine.

Les graphismes en Turbo-Pascal feront l'objet d'un deuxième tome au présent ouvrage.

CHAPITRE 3

VOCABULAIRE DE BASE

L'ALPHABET

L'alphabet d'un langage est constitué de tous les signes typographiques qui peuvent être utilisés au cours de l'écriture d'un programme. En dehors des lettres A à Z (majuscules ou minuscules sont équivalentes) et des dix chiffres 0 à 9, certains signes ont en Turbo-Pascal une signification particulière :

+, -, /, * sont les opérateurs arithmétiques et ensemblistes ;

<,>, =, <=,>=,<> sont les opérateurs relationnels ;

(,),[,],{,} et le caractère blanc sont des séparateurs ;

' ; : . sont les signes de ponctuation du langage ;

:= est le signe de l'affectation (nous y reviendrons).

_ (souligné) est utilisé à la place du caractère blanc là où celui-ci est interdit.

LES MOTS

Les mots qui forment le langage sont constitués d'une combinaison de signes, séparés par des délimiteurs (mots réservés, opérateurs, ponctuation) ou des séparateurs (blancs, retours chariot ou commentaires). Ils appartiennent tous à l'une des catégories suivantes :

- **Mots clefs (ou réservés)**: ces mots ont une signification particulière pour Turbo-Pascal et font partie intégrante du langage ; ce sont en général des

instructions ou des en-têtes de déclaration. Nous verrons leur signification en détail plus loin. Ils sont considérés comme des délimiteurs. En voici la liste :

absolute	*external	nil	*shr
and	file	not	*string
array	for	of	then
begin	forward	or	to
case	function	packed	type
const	goto	procedure	until
div	if	program	var
do	in	record	while
downto	*inline	repeat	with
else	label	set	*xor
rnd	mod	*shl	

Les astérisques précèdent les mots non définis dans le Pascal standard.

- **Nombres** : les nombres peuvent apparaître dans un programme Turbo-Pascal sous la forme entière (ex. 67), décimale (ex. 114.9), scientifique (ex. 25E6) ou hexadécimale (ex. $6EAF). Turbo-Pascal offre la possibilité de définir des constantes numériques et de les utiliser ensuite au sein du programme. Les valeurs maximales admises pour chacune de ces formes seront vues au chapitre 5.

- **Identificateurs** : ce sont des noms que l'utilisateur donne aux programmes, constantes, types, variables, procédures ou fonctions. Les règles sont simples : ce sont des chaînes de lettres ou de chiffres commençant obligatoirement par une lettre, et pouvant comporter un nombre quelconque de caractères, tous significatifs (souvenez-vous quand même qu'une ligne a au plus 127 caractères). Il ne doit pas y avoir d'espace dans un identificateur, mais on peut le remplacer par le caractère souligné (_). Turbo-Pascal ne fait pas la différence entre majuscule et minuscule; on peut donc s'en servir pour rendre les identificateurs plus lisibles :

  `NombreDePassages` ou `nombre_de_passages` est plus facile à lire que `nombredepassages` ou `NOMBREDEPASSAGES`.

  N'hésitez pas à employer des mots explicites pour nommer vos variables; le programme en sera plus lisible (même s'il est un peu plus long à taper!). N'ayez pas peur de consommer de la place en mémoire en utilisant des identificateurs longs : ils ne sont réellement lus qu'à la compilation ; dans le code généré par le compilateur, ils sont transformés en un pointeur qui, de toute façon, occupe deux octets, quelle que soit la longueur de l'identificateur.

  Un identificateur ne doit pas être un mot clef. Cependant, il existe des identificateurs prédéfinis pour les types, fonctions et procédures prédéfinis. Ces

identificateurs standard peuvent être "recouverts" par vos propres définitions ; vous perdez alors la définition prédéfinie. Attention donc dans le choix de vos identificateurs.

Les identificateurs suivants sont corrects :

```
nombre_de_lignes Jours toto AMSTRAD CalculerLaMoyenne reste
B6  Mach3 Machin
```

mais ceux-ci ne le sont pas :

`3X  @5  253  mon nombre` (contient un espace) `aujourd'hui` (contient une apostrophe).

- **Les chaînes de caractères** : on est amené dans le courant d'un programme à spécifier des chaînes de caractères qui doivent être prises telles quelles, par exemple pour imprimer des messages, affecter une valeur à une variable chaîne, etc. En Basic, les chaînes sont notées entre guillemets ; en Turbo-Pascal, elles sont notées entre apostrophes :

```
'Oh, la belle bleue!'
```

est une chaîne valide. N'importe quel caractère peut apparaître dans la chaîne ; seule l'apostrophe doit être redoublée afin d'éviter toute ambiguïté :

```
'J''ai pris le train aujourd''hui.'
```

"" est une chaîne comprenant UNE apostrophe.

Turbo-Pascal permet également la définition de constantes chaînes, qui seront ensuite utilisées dans le courant du programme.

- **Les commentaires** : on peut insérer des commentaires dans un programme Turbo-Pascal. Ils peuvent être encadrés soit par des accolades { } soit à gauche par (*, et à droite par *):

```
(* ceci est un commentaire qui sera ignoré par le pro-
gramme*)
{ Ceci est un autre commentaire valide}
```

Les commentaires sont utiles pour clarifier la lecture d'un programme; bien que Turbo-Pascal soit un langage très lisible, comparé à d'autres (FORTH par exemple), la pensée d'un programmeur est toujours difficile à suivre. Pensez donc à commenter votre programme aux endroits "stratégiques".

- **Les directives de compilation** : elles indiquent au compilateur quelles options le programmeur désire. Elles peuvent aussi servir à inclure un morceau de

programme présent sur la disquette dans le programme que vous écrivez. Elles se présentent sous la forme d'un commentaire comprenant le signe **$** et une directive, ou plusieurs directives séparées par les virgules :

```
{$R+}
{$I-,A+}
(*$I grafik.inc*)
```

Les espaces ne sont pas autorisés avant ou après le signe **$** ni après le nom d'un fichier inclus. Une liste complète des directives de compilation se trouve en annexe 4.

LA SYNTAXE GENERALE

Un programme Turbo-Pascal est fait de deux sections bien distinctes : une section de **déclarations** et une **section exécutable**. Chaque section est faite de phrases, toutes séparées par des points-virgules. Ces phrases sont elles-mêmes composées d'une ou plusieurs instructions, toujours séparées par des points-virgules. Nous allons maintenant étudier en détail ces deux sections.

CHAPITRE 4

L'ARCHITECTURE D'UN PROGRAMME PASCAL

Un programme Pascal a une architecture très différente de celle d'un programme Basic. Il se divise en deux parties :

- une section de déclarations ;
- une section exécutable.

L'ensemble de ces deux sections forme ce qu'on appelle un bloc. Cette notion est importante, car la structure de bloc se retrouve dans le corps des fonctions et procédures.

LES DECLARATIONS

La section déclarations est quelquefois plus longue que la section exécutable; en effet, en Pascal, tout "objet" (étiquette, type, variable, fonction, procédure) invoqué dans le corps du programme doit au préalable être déclaré dans la section déclarations. C'est, bien sûr, une contrainte par rapport au Basic, mais cela est rendu nécessaire par le processus de compilation. De plus, cela impose de bien penser son programme avant de l'écrire, ce qui va tout à fait dans le sens de la programmation structurée.

La section déclarations peut commencer (mais en Turbo-Pascal ce n'est pas obligatoire) par le mot réservé **Program**, suivi obligatoirement d'un nom (identificateur à votre choix). Ce nom caractérisera alors le programme. En voici un exemple :

```
Program Premieressai;
```

Notez le point-virgule. C'est un séparateur ; il indique au compilateur la fin d'une instruction. Il est présent à la fin de presque toutes les lignes d'un programme Pascal, à quelques exceptions près, et son oubli est la cause la plus fréquente d'erreurs de syntaxe.

La section déclarations se poursuit avec les déclarations d'étiquettes, de constantes, de types, de variables, de procédures et fonctions. Toutes ne sont pas toujours présentes dans tous les programmes, cela dépendra des besoins propres à chaque application.

Note pour les habitués du Pascal standard : en Turbo-Pascal, l'ordre des déclarations est indifférent, et chaque déclaration peut apparaître plusieurs fois, pourvu que tout objet utilisé soit déclaré auparavant (à l'exception des pointeurs).

La déclaration d'étiquettes (label)

En Pascal, langage structuré, on évite les instructions de saut à une autre partie du programme (**Goto**). Cependant, il y a des cas où son emploi rend l'écriture du programme plus simple. Les labels servent donc à référencer les sauts. A la différence du Pascal standard, les labels peuvent être en Turbo-Pascal, soit des nombres, soit des identificateurs ; comme tout objet non prédéfini, ils doivent être déclarés avant d'être utilisés. On emploie pour cela le mot réservé **label**, suivi du ou des labels déclarés, séparés par des virgules :

```
Label 1,ici,fin,sortie;
```

Notez encore le point-virgule : il indique la fin de la déclaration de label.

Dans le corps du programme, le label sera suivi de deux points, et identifiera l'instruction qui le suit :

```
ici: Write('Vive le Pascal');
```

Quand, à l'exécution, le programme rencontrera une instruction **Goto** `ici`, il se branchera au label `ici` et écrira le message : `'Vive le Pascal'`.

Write est une procédure prédéfinie que nous étudierons en détail plus loin ; cependant sachez dès maintenant qu'elle affiche sur l'écran des variables et/ou des chaînes de caractères :

`Write(i);` affiche le contenu de la variable `i` (`i` doit être d'un type simple ou intervalle).

`Write('Ceci est une phrase');` les chaînes de caractères doivent être mises entre apostrophes.

On peut mélanger variables et chaînes en les séparant par des virgules; par exemple, si la variable `j` vaut 7 :

`Write('Il y a ',j,' jours dans la semaine');` affichera à l'écran `'Il y a 7 jours dans la semaine'`.

WriteLn est identique à **Write** à ceci près que la prochaine impression se fera sur la ligne suivante au lieu de se faire à la suite, sans espace, sur la même ligne.

La déclaration de constantes

En Pascal, il est souvent pratique de donner un nom à certaines constantes du programme. De cette façon, en cas de besoin, une seule modification au début du programme permettra d'adapter celui-ci à d'autres cas.

Prenons un exemple : vous écrivez un programme qui travaille sur une classe de 30 enfants. Vous calculez la moyenne de leurs notes, leur taux d'absence, le nombre de cahiers nécessaires... A chaque calcul, le nombre 30 apparaîtra. Si vous apprenez ensuite que finalement les enfants seront 32 au lieu de 30, ou si vous voulez appliquer le programme à une autre classe, il faudra aller changer le nombre 30 dans tous les calculs. Un bon moyen d'éviter cette recherche, fastidieuse et source d'erreurs, est de déclarer le nombre d'élèves comme constante :

```
const   enfants = 30;
```

et de faire tous vos calculs sur "`enfants`" au lieu de "`30`". Lors de l'adaptation du programme, il suffira de modifier la constante pour obtenir les résultats sur tout autre nombre d'enfants :

```
const   enfants = 32;
```

Les constantes peuvent être de plusieurs sortes : numériques, chaîne de caractères ou structurées, de plus elles peuvent être typées ou non; elles sont déclarées par le mot réservé **const** et sont séparées dans la déclaration par des points-virgules :

```
const   taux = 11.5;
        PI = 3.1416;
        Titre = 'Maitrisez Turbo-Pascal sur AMSTRAD';
```

Nous reviendrons sur les constantes structurées et typées dans les chapitres 5 et 12.

La déclaration de types

Une partie de la puissance de Pascal vient de la notion de type. Dans presque tous les langages évolués, les variables peuvent avoir différents types. En général ces types sont au nombre de trois : entier, réel, chaîne de caractères (cas de la

plupart des Basic). En Turbo-Pascal, non seulement les types prédéfinis sont plus nombreux : **integer** (entier), **real** (réel), **boolean** (booléen), **byte** (octet), **char** (caractère) et **string** (chaîne), mais de plus on peut définir soi-même des types particuliers à l'application que l'on veut faire. Les variables déclarées avec un certain type ne pourront prendre que les valeurs admises par ce type.

Les types prédéfinis n'ont pas à être déclarés, ils sont intrinsèques au Turbo-Pascal. Les différents types et leur composition seront étudiés en détail dans le prochain chapitre. Sachez pour l'instant que la déclaration de type commence par le mot réservé **type** suivi de la définition de chaque type, séparée par des point-virgules :

```
Type    assaisonnement = [huile, vinaigre, sel, poivre,
                          citron,ail];
        sauce = set of assaisonnement;
        jours = (lundi, mardi, mercredi, jeudi, vendredi,
                 samedi, dimanche);
        hexadécimal = (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F);
        vieux = 0..120;
        personne = record
                       nom : string;
                       age : vieux;
                   end;
```

Vous n'avez pas besoin de comprendre complètement dès à présent toutes ces déclarations, retenez simplement qu'elles peuvent être très éclectiques !

Turbo-Pascal admet plusieurs déclarations de type dans un même programme ; la seule règle est que tout identificateur doit être déclaré avant emploi (sauf bien sûr les identificateurs prédéfinis et les pointeurs).

La déclaration de variables

En Pascal, toute variable doit être déclarée avant son utilisation (avec une petite exception pour les pointeurs que nous verrons au chapitre 14). Les variables déclarées dans le programme principal seront dites variables globales, par opposition aux variables locales, déclarées dans les fonctions ou procédures (voir chapitres 5 et 9).

La déclaration associe une ou plusieurs variables (séparées par des virgules) avec un type qui peut être soit prédéfini, soit déclaré précédemment. L'affectation d'un type à une variable se fait à l'aide des deux points (**:**), et les déclarations des

variables de types différents sont séparées par les points-virgules. Soit par exemple, si on suppose définis les types ci-dessus :

```
var     condition : boolean;
        i,j,premier : integer;
        mayonnaise : sauce;
        chiffre_hexa : hexadecimal;
        untel : personne;
```

Attention : la déclaration d'une variable ne donne pas de valeur initiale à celle-ci. Les variables sont indéterminées jusqu'à ce qu'on leur affecte une valeur dans la section exécutable.

Turbo-Pascal admettant plusieurs déclarations de types et de variables, c'est une bonne idée de grouper dans la mesure du possible les déclarations de types et celles de variables de type correspondant : on déclarera ainsi une variable juste après son type. Cela facilitera la relecture.

Les déclarations de fonctions et de procédures

Les fonctions et les procédures sont des sortes de sous-programmes qui retournent des valeurs dans certaines variables ou qui effectuent une tâche bien déterminée. Il en existe de deux sortes: prédéfinies et utilisateur. Les procédures et fonctions prédéfinies n'ont évidemment pas à être déclarées ; il en existe de toute sorte en Turbo-Pascal et nous les étudierons au chapitre 9.

Par contre, les fonctions ou procédures de l'utilisateur seront donc déclarées et définies dans cette partie. Pascal est récursif. Cela signifie qu'une structure du langage peut être composée d'une structure semblable à elle-même. Ainsi, dans la section déclaration du programme, certaines fonctions ou procédures pourront être déclarées; or ces fonctions et procédures ont la même architecture qu'un programme, c'est-à-dire qu'elle sont faites d'un bloc : section déclarations (contenant éventuellement des déclarations de labels, constantes, types et variables locaux, et pouvant elle-même déclarer de nouvelles fonctions et procédures plus internes, et ainsi de suite sur autant de niveaux que nécessaire), et une section exécutable qui indique ce que fait la procédure ou la fonction.

Le nom de la procédure ou de la fonction devient dès qu'il est déclaré un **mot du langage** ; la mention de ce nom dans le programme suffira à provoquer son exécution. Nous reviendrons bien sûr sur les fonctions et procédures de l'utilisateur dans les chapitres 8 et 9.

Exercice : relevez les erreurs éventuelles dans les déclarations suivantes :

```
Program erreurs
const   UN = 5
        titre = LES CONSTANTES;
label 1;3;toto;
var     nombre = UN;
        mot : string[3];
```

(Le corrigé est en fin de chapitre)

LA SECTION EXECUTABLE

La section exécutable d'un bloc est une suite d'instructions, exécutées les unes après les autres. Elle doit commencer par **begin** et se terminer par **end.** dans le cas du programme ou par **end;** dans le cas d'une procédure ou fonction. Les mots réservés **begin** et **end** encadrent donc un groupe d'instructions (séparées à l'intérieur du groupe par des points-virgules sauf pour la dernière instruction précédant le **end**, pour laquelle il peut être omis). Les instructions qui composent un groupe sont soit des instructions de base, soit des appels à des fonctions ou procédures (prédéfinies ou utilisateur).

Certaines de ces instructions peuvent être réunies pour former des instructions composées (sous-groupes), qui commenceront elles aussi par **begin** et se termineront par **end;**. Chaque sous-groupe peut alors être considéré comme une macro-instruction. Certains sous-groupes ne seront utiles qu'à la lisibilité du programme.

Le programme se termine par le mot-clef **end** suivi d'un point **(.)**, en relation avec le **begin** qui ouvre la section exécutable.

Voici un exemple d'architecture d'un programme Turbo-Pascal :

```
Program  demo;
{declarations possibles de label, const, type, var valables
dans tout le programme}
procédure  p1;
{declarations possibles de label, const, type, var locales à p1}
  procedure  p2;
  {declarations possibles de label,const,type,var locales à p2}
  begin {section executable de p2}
        instruction1;
  end; {fin de p2}
```

```
begin {section executable de p1}
        instruction3;
        instruction4;
        begin
              instruction5;
              p2; {appel a la procedure p2}
        end;
end; {fin de p1}

{declaration possible d'autres procedures ou fonctions}

begin {section executable du programme}
  instruction6;
  p1; {appel a la procedure p1}
end. {fin du programme}
```

Pour rendre la lecture plus facile, on a l'habitude d'indenter (décaler vers la droite) chaque nouveau sous-groupe. On voit ainsi facilement quel **end** se rapporte à quel **begin**. On ne met en général qu'une ligne par instruction, toujours pour améliorer la lisibilité. Toutefois ce n'est pas obligatoire, et on peut mettre autant d'instructions sur une ligne qu'il peut en tenir (maximum 127 caractères). Voyez la différence :

```
Program illisible; const a=10; var i,j:integer; begin for
i := 1 to a do begin for j := 1 to i do write(j);writeln;
end; end.
```

```
Program  Tres_Bien_Presente;{il  n'y a pas de caracteres
                              accentues}
const a = 10;
var i,j : integer;
begin
  for i := 1 to a do
        begin
                for j := 1 to i do Write(j);
                WriteLn;
        end;
end.
```

Ces deux programmes sont semblables, mais avouez que le second est plus agréable et surtout plus clair que le premier.

QUELQUES PETITS EXEMPLES.

Nous en savons assez maintenant pour construire quelques petits programmes.

```
Program présentation;
var nom : string[10]; {il faut specifier le nombre maximum de
                       caracteres}
Begin
  WriteLn('Quel est votre nom?');
  ReadLn(nom);
  Write('Bonjour ', nom);
End.
```

Read et **ReadLn** sont des procédures prédéfinies symétriques de **Write** et **WriteLn**, qui acceptent une entrée depuis le clavier (terminée par **<RETURN>**), en allant à la ligne ensuite dans le cas de **ReadLn**. Le programme se comprend tout seul.

```
Program cercle;
var rayon, surface : real;
begin
  WriteLn('Entrez le rayon du cercle');
  Readln(rayon);
  surface := pi*rayon*rayon;
  writeln(surface);
end.
```

Ce programme appelle quelques précisions : **pi** est une constante prédéfinie ; inutile donc de la déclarer. Le signe **":="** est l'opérateur d'affectation. Nous l'avons déjà rencontré dans les exemples. Il permet de donner une valeur à une variable préalablement déclarée. Le signe **"*"** est l'opérateur de multiplication ; `rayon*rayon` donne donc le carré de `rayon`. Il existe aussi une fonction prédéfinie pour calculer le carré d'un nombre (**Sqr**), nous la verrons plus loin.

```
Program cube; {calcule le cube du nombre entre au clavier}
var nombre, resultat : integer;
Begin
  Write('Entrez un nombre? ');
  ReadLn(nombre);
  resultat := nombre*nombre*nombre;
  WriteLn('Le cube de ',nombre,' est ' ,resultat);
end.
```

Clair, n'est-ce-pas ?

Corrigé des exercices

1- déclaration de programme : il manque un ";" après erreurs ;

2- déclaration de label : les labels doivent être séparés par des virgules. Cette ligne doit donc s'écrire : `label 1,3,toto;`

3- déclaration de constantes: UN = 5 est correct (drôle d'idée, mais enfin...). Par contre, il manque aussi le point-virgule après 5. La chaîne formant la constante titre doit être mise entre apostrophes: `titre = 'LES CONSTANTES';`

4- déclaration de variables : on ne peut pas assigner de valeur initiale à une variable dans sa déclaration. On déclare seulement son type. nombre = UN est donc incorrect (le signe = n'aurait de toute façon pas été approprié) puisque UN n'est pas un type, mais une constante.. Il fallait déclarer : `nombre : integer; mot : string[3]` est correct (parce que **string[]** est un type prédéfini, dans lequel il faut indiquer la longueur maximale que la chaîne peut prendre).

CHAPITRE 5

LES TYPES SIMPLES ET LEURS OPERATEURS

Un type est l'ensemble des valeurs que peut prendre une variable. Il y a trois grandes classes de types : les types scalaires, les types structurés, et le type pointeur. Nous n'étudierons dans ce chapitre que les types scalaires prédéfinis et leurs opérateurs ; les autres types seront vus plus loin.

LES TYPES SCALAIRES PREDEFINIS (OU TYPES SIMPLES)

On appelle type scalaire un type dans lequel une variable ne peut prendre qu'une seule valeur à la fois, par opposition aux types ensemble ou structurés (tableaux) où une variable peut prendre plusieurs valeurs en même temps.

Les types scalaires prédéfinis ou types simples sont les suivants : integer (entier), real (réel), boolean (booléen) et char (caractère).

- **Le type integer** (entier). Un nombre est entier en Turbo-Pascal s'il est compris entre -32768 et +32767, sans décimale. Les calculs sur les entiers sont les plus rapides, mais aucun résultat, même intermédiaire, ne doit sortir des limites, sous peine d'un message d'erreur (Overflow).

- **Le type real** (réel). Les nombres réels doivent être en valeur absolue plus petits que 1E38 et plus grands que 1E-38. Ils ont onze chiffres significatifs. Une valeur entière peut être affectée à une variable réelle (mais non l'inverse).

- **Les variables de type boolean** (booléen) peuvent prendre une des deux valeurs **False** (faux) ou **True** (vrai) qui sont des constantes prédéfinies. Elles sont utilisées par les opérations logiques.

- **Le type byte** (octet) est une partie du type integer. Une variable de type byte peut prendre une valeur entière inférieure ou égale à 255. Une valeur de type byte peut être affectée à une variable réelle ou entière ; par contre, une valeur réelle ne peut jamais être donnée à une variable de type byte ; quant aux entiers, ils doivent être compris dans les limites 0-255, sans quoi ils seront rendus modulo 255. Le type byte accepte les nombres négatifs, mais les retourne comme s'ils étaient positifs.

- **Le type char** représente un caractère, pris dans le groupe des caractères disponibles sous CP/M. Les caractères sont rangés dans l'ordre ASCII **(A'<'Z'<'z')**. Les caractères semi-graphiques habituels de l'Amstrad (codes 129 à 255) sont accessibles sous CP/M 2.2, tandis qu'ils sont remplacés par les caractères internationaux sous CP/M Plus. On peut également accéder aux caractères de contrôle, soit en faisant précéder leur numéro (décimal ou hexa avec **$**) d'un signe **#**, soit en faisant précéder leur lettre de contrôle du signe **^**. Dans les deux cas, il ne faut pas mettre d'apostrophe.

  Exemple : le caractère ASCII 7 (sonnerie) peut être obtenu soit par **#7**, soit par **^G** (G est la 7ième lettre de l'alphabet). Les caractères de contrôle ne sont pas les mêmes sous CP/M 2.2 et sous CP/M Plus. En fait, ils dépendent aussi du terminal installé. Sous CP/M Plus, l'Amstrad 6128 émule le terminal Zenith; les codes de contrôle sont ceux qui se trouvent à la page 7 du chapitre 4 du manuel du 6128. Sous CPM 2.2 les codes de contrôle sont les mêmes qu'en Basic.

LES CONSTANTES PREDEFINIES

En Turbo-Pascal, il existe quatre constantes prédéfinies :

Nom	Type	(Valeur)
• Pi	Real	3.145926536E+00
• False	Boolean	Valeur booléenne "faux"
• True	Boolean	Valeur booléenne "vrai"
• Maxint	Integer	Le plus grand entier possible (32767).

On peut donc utiliser ces constantes sans avoir à les redéfinir.

LES OPERATEURS ET EXPRESSIONS SUR LES TYPES SCALAIRES

Les expressions sont des phrases qui expriment quel traitement il faut effectuer sur leurs opérandes : variables, constantes, fonctions. Ces traitements sont spécifiés à l'aides des opérateurs. Les éléments qui encadrent un opérateur doivent être de même type ou de type compatible.

Les opérateurs arithmétiques

Ils ne portent que sur les **réels** et les **entiers**. Ce sont :

+	addition
-	soustraction
*	multiplication
/	division
div	division entière
mod	modulo (reste de la division entière)
not	négation logique bit à bit
or	ou logique bit à bit (sur deux octets)
xor	xor logique bit à bit (sur deux octets)
and	et logique bit à bit (sur deux octets)
shl	décalage des bits à gauche (shift left)
shr	décalage des bits à droite (shift right)

div, mod, not, or, xor, and, shl et shr n'admettent que les entiers, et le résultat est bien sûr un entier.

Pour les autres opérateurs arithmétiques, le résultat est un réel à moins que les deux opérandes ne soient entiers, auquel cas le résultat est ausi un entier.

Une expression arithmétique est la combinaison de deux ou plusieurs opérandes entiers, réels ou bytes, séparés par des opérateurs arithmétiques. La variable d'accueil du résultat doit avoir un type compatible avec ce résultat, sous peine d'une erreur à la compilation. L'ordre dans lequel les opérations sont effectuées est imposé par l'ordre de précédence des opérateurs donné un peu plus loin.

Les opérateurs logiques

Ils agissent sur les booléens. Ce sont not, and, or et xor:

- Not :

A	Not A
True	False
False	True

Table de vérité de Not

- And :

A	B	A And B
True	False	False
True	True	True
False	True	False
False	False	False

Table de vérité de And

- Or :

A	B	A or B
True	False	True
True	True	True
False	True	True
False	False	False

Table de vérité de Or

- Xor :

A	B	A Xor B
True	False	True
True	True	False
False	True	True
False	False	False

Table de vérité de Xor

Le résultat est toujours un **booléen**. Une expression booléenne est soit une combinaison d'opérandes booléens, séparés par des opérateurs booléens, soit une expression relationnelle (voir ci-dessous), soit une combinaison des deux.

Les opérateurs relationnels

Les opérateurs relationnels agissent sur tous les types ordonnés, c'est-à-dire les types scalaires prédéfinis et les types scalaires définis par l'utilisateur (intervalle et énumération). Les entiers, les bytes, les réels sont ordonnés naturellement; les booléens sont ordonnés **False<True**, et les **Char** (caractères) sont ordonnés suivant l'ordre ASCII. Les opérateurs relationnels sont :

=	égal
<>	différent
>	supérieur
>=	supérieur ou égal
<	inférieur
<=	inférieur ou égal

Le résultat d'une expression relationnelle est un **booléen** : **True** si la relation est vérifiée, **False** sinon : en Pascal, une expression relationnelle est en même temps une expression booléenne. Il est donc correct d'écrire :

```
var    test : boolean;
       i,j : integer;
begin
   ..........
   test := i<j;
end.
```

ORDRE DE PRECEDENCE DES OPERATEURS

Lorsqu'ils sont mélangés, les opérateurs sont pris dans un ordre bien défini par leur niveau de précédence ou priorité.

1) le changement de signe (- avec un seul opérande).

2) not.

3) *,/,div, mod, and, shl et shr.

4) +,-,or et xor.

5)=,<>,<,>,<=,>= et in (opérateur d'appartenance, voir le type ensemble au chapitre 12).

Les suites d'opérateurs de même niveau sont évaluées de gauche à droite. Les expressions entre parenthèses sont evaluées en priorité, quels que soient les opérateurs qui les précèdent ou les suivent. Si vous avez un doute sur l'ordre qui sera pris en compte par le compilateur, usez des parenthèses : c'est facile, ce n'est pas long, et cela peut éviter bien des erreurs !

Exemple : si on suppose faites les déclarations :

```
var  A,B,C,D,E : integer;
     test : boolean;
```

dites de quel type seront les expressions suivantes :

1) -A/B+(C-D*E) ;
2) A+D<C/8 ;
3) 10*E-D ;
4) B=C or A<E ;
5) A xor E ;
6) test and (-D*A=E)

Réponses :

1) Réelle (à cause de la division).
2) Booléenne.
3) Entière.
4) Booléenne.
5) Entière (Xor logique bit à bit).
6) Booléenne.

CHAPITRE 6

LES INSTRUCTIONS EXECUTABLES

Vous connaissez maintenant la structure de bloc ; vous savez qu'elle se divise en deux sections, une section déclarations et une section exécutable. Vous savez aussi que la structure de bloc est commune aux programmes, procédures et fonctions.

La section exécutable d'un bloc est une suite d'instructions, exécutées les unes après les autres. Elle doit commencer par **Begin** et se terminer par **End.** dans le cas du programme ou par **End;** dans le cas d'une procédure ou fonction. Les mots réservés **Begin** et **End** encadrent donc un groupe d'instructions (séparées par des points-virgules).

Certaines de ces instructions peuvent être réunies pour former des instructions composées (sous-groupes), qui commenceront elles aussi par **Begin** et se termineront par **End;**. Chaque sous-groupe peut alors être considéré comme une macro-instruction. Certains sous-groupes ne seront utiles qu'à la lisibilité du programme (c'est pourquoi on a l'habitude de les indenter vers la droite, ils sautent aux yeux), d'autres seront nécessaires pour éviter toute ambiguïté dans le déroulement du programme.

On peut distinguer trois sortes d'instructions :

- **Les instructions composées** : comme il est expliqué plus haut, on peut regrouper plusieurs instructions simples dans une instruction composée (ou sous-groupe d'instructions).

- **Les instructions de base** : ce sont les instructions prédéfinies qui forment la base du langage. Elles sont en nombre restreint. Elles comprennent les instructions simples et les instructions structurées (conditionnelles et boucles).

- **Les appels de procédures ou fonctions** : une fonction ou procédure, qu'elle soit prédéfinie ou déclarée par l'utilisateur, devient une instruction par elle-même ; pour l'utiliser, il suffit d'écrire son nom. Les procédures et fonctions sont étudiées aux chapitres 9 et 10.

LES INSTRUCTIONS COMPOSEES

Vous en savez déjà presque tout : ce sont des groupes d'instructions commençant par **Begin** et finissant par **End;**. On les appelle aussi sous-blocs. Partout où une instruction est possible, une instruction composée l'est aussi.

LES INSTRUCTIONS DE BASE SIMPLES

Elles sont peu nombreuses : instruction d'affectation, instruction nulle, **Goto**.

L'instruction d'affectation

Cette instruction sert à assigner une valeur à une variable. On utilise pour cela le signe := qu'il ne faut pas confondre avec le signe = qui dénote l'égalité réelle. Ce signe pourrait se lire "devient". En Basic, une expression de la forme A =A+1 est parfaitement légale. Pourtant, du point de vue mathématique ou simplement arithmétique, c'est une abérration! En Turbo-Pascal, il faut écrire A : =A+1, qu'on peut lire "A devient l'ancienne valeur de A, augmentée de 1." ou "J'affecte à la variable A son ancienne valeur augmentée de 1". Ce signe sera donc utilisé à chaque fois qu'une valeur doit être donnée à une variable, soit explicitement, soit par calcul.

Le signe = existe bien sûr aussi, mais ne pourra servir qu'en cas d'égalité véritable ; c'est pourquoi les constantes ou les types sont déclarés avec lui : la constante est réellement égale à la valeur déclarée, et le type est défini par la déclaration. On s'en sert aussi pour tester une valeur dans une expression booléenne : `nombre = quantité` ne donne pas la valeur `quantité` à `nombre`, mais est une expression qui retourne la valeur booléenne **True** (vrai), si les deux variables sont égales, ou **False** (faux) sinon.

La déclaration d'une variable ne lui **affecte pas** de valeur initiale. Il faut l'initialiser par une instruction d'affectation. Nous verrons qu'il y a moyen en Turbo-Pascal de contourner ce léger inconvénient, en utilisant les constantes typées, qui sont en fait plus proches des variables que des constantes (voir chapitre 7).

Voici quelques exemples d'affectations :

```
Program affectation ; {Ce programme ne fait rien d'autre que
                          d'initialiser des variables}
const beaucoup = 1000 ;
Var   condition,test : boolean;
      nombre, quantite : real;
      lettre : Char;
Begin
  condition := true;
  lettre := 'p';
  nombre := 56.84;
  quantité := beaucoup;
  test  :=  nombre=beaucoup  {test  prendra  la  valeur  false
                              puisque nombre vaut 56.84 et non
                              pas 1000};
End.
```

L'instruction nulle

Cette instruction ne fait absolument rien ; elle est utilisée quand la syntaxe demande une instruction, et qu'il ne faut rien faire. Elle est de la forme :

```
Begin
End ;
```

L'instruction Goto

L'instruction **Goto** sert à brancher l'exécution à une instruction référencée par un label. Il faut que ce label ait été déclaré dans le bloc où se produit l'appel. Cela signifie qu'on ne peut pas sauter dans une procédure ou fonction ou hors d'elle. On dit que la portée du label est le bloc où il est déclaré. La notion de portée se retrouve pour tous les identificateurs, et sera développée à propos du passage des paramètres, au chapitre 11.

L'instruction d'arrivée est précédée de son label et de deux points (**:**).

Exemple :

```
label NouveauTraitement;
...
Begin
  if A=B then goto NouveauTraitement;
  instruction1;
  instruction2;
  NouveauTraitement:  A :=0;
  instruction3;
End.
```

Si A=B, les `instruction1` et `instruction2` sont sautées, on ira directement à `NouveauTraitement`, c'est-à-dire mettre A à zéro, puis exécuter `instruction3`. Si A<>B, il n'y a pas de saut, `instruction1`, `instruction2`, `A:=0` et `instruction3` sont normalement exécutées en séquence. Cela introduit l'instruction **If...then...else**, commune à tous les langages, qui va être étudiée maintenant. Encore un mot cependant sur **Goto**. En Pascal, on évite le plus possible de l'utiliser. On préfèrera structurer le programme, et éventuellement le diviser en procédures plus nombreuses. Cela donne un programme beaucoup plus clair, séquentiel et à l'opposé des programmes "spaghetti" que l'on recontre parfois en Basic.

LES INSTRUCTIONS DE BASE STRUCTUREES

On regroupe sous ce titre les instructions conditionnelles et les instructions de boucle.

Les instructions conditionnelles

Il y a deux instructions conditionnelles en Turbo-Pascal. Ce sont **If...then...else** et **Case...of...else**.

L'instruction If...then...else...

C'est le test classique de condition (si...alors...sinon...). Sa syntaxe est :

```
if condition
  then instruction1
  else instruction2;
instruction3;
```

`Condition` doit être une expression booléenne. Si elle est évaluée **True** (vraie), on exécute `instruction1`, puis `instruction3`. Si elle est **False** (fausse), on exécute `instruction2`, puis `instruction3`. Remarquez l'absence

de point-virgule avant **else.** La clause **else** peut d'ailleurs être absente ; `instruction1` est alors terminée par un point-virgule.

Bien entendu, `instruction1` et `instruction2` peuvent être des instructions composées (**Begin**...**End**), mais dans tous les cas, il ne doit pas y avoir de point-virgule avant **else.**

```
Program Exemple_de_If;
Var     lettre : Char;
        code : Char;
Begin
        code := Chr(Trunc(Random*27)+65);
        WriteLn('J''ai choisi une lettre majuscule, trouvez-la
        en la tapant');
        ReadLn(lettre);
        If lettre = code then WriteLn('Bravo, vous avez gagné!)
                        else WriteLn('Perdu, c''etait',code);
End.
```

Chr, **Trunc**, et **Random** sont des fonctions que vous pourrez trouver au chapitre 9 ; sachez, dès à présent, que la combinaison utilisée ici retourne un caractère compris entre 'A' et 'Z'.

Les tests peuvent être imbriqués, de la forme :

```
if condition1 then
if condition2 then
  instruction1;
  instruction2
else
  intruction3
else
  instruction4;
```

Il vaut mieux introduire un bloc **Begin**...**End** pour éviter toute ambiguïté, surtout si un des **if** n'a pas de **else** (bien qu'en principe un **else** se rapporte au dernier **if** sans **else**). On obtient une structure beaucoup plus lisible :

```
if condition1 then
        Begin
            if condition2 then
                 instruction1;
                 instruction2
            else
                 instruction3;
        end
        else
            instruction4;
```

Remarquez à chaque fois l'absence de point-virgule avant **else**.

L'instruction Case...of...else

Cette instruction permet de choisir le traitement à effectuer d'après la valeur d'une variable non plus booléenne mais d'un type quelconque. Il peut donc y avoir plus de deux traitements possibles. La syntaxe est la suivante :

```
case expression of
  valeur1: instruction1;
  valeur2: instruction2;
  ..........................
  valeur n: instruction n;
else
  instruction n+1;
End;
```

Remarquez que, cette fois, il y a un point-virgule avant **else**. La clause **else** est d'ailleurs une extension Turbo par rapport au Pascal standard et est facultative. `valeur1, valeur2...` s'appellent des étiquettes de cas ; elles doivent être du même type qu'`expression`, qui doit elle-même être d'un type scalaire.L'instruction **case** évalue `expression`, puis si la valeur trouvée figure dans une des étiquettes de cas, l'`instruction` correspondante (qui peut être composée) est exécutée, puis on passe à l'instruction qui suit le **End** se rapportant au **Case.** Si la valeur trouvée ne figure pas dans une étiquette de cas, on exécute l'instruction qui suit **else**, si elle est présente, ou l'instruction suivante sinon.

```
Program Exemple_de_Case;
Var     age : Integer;
Begin
  WriteLn('Entrez votre age');
  ReadLn(age);
  Case age of
         0..15   : WriteLn('Vous etes bien jeune!');
        16..30   : WriteLn('Felicitations!');
        31..50   : WriteLn('Ca va encore!');
        51..99   : WriteLn('Ouh la la!');
        100..Maxint   : WriteLn('Menteur!');
  End;
End.
```

Les instructions de boucle

Les boucles, avec les conditions, sont un des éléments fondamentaux des langages informatiques. Elles donnent une grande puissance de calcul, en exécutant un grand nombre de fois un petit nombre d'instructions.

Il existe en Turbo-Pascal trois structures de boucles : **For...do**, **Repeat...until**, **While...do**.

La structure For...do

Cette structure de boucle sera utilisée lorsqu'on connaît à l'avance le nombre de boucles à effectuer. Elle est semblable à la structure FOR...NEXT du Basic.La syntaxe est :

```
For VariableDeControle := ValeurInitiale to ValeurFinale do
instruction;
```

si `ValeurInitiale` est plus petite que `ValeurFinale`, ou

```
For VariableDeControle := ValeurInitiale downto ValeurFinale
do instruction;
```

si `ValeurInitiale` est plus grande que `ValeurFinale`.

`VariableDeControle`, `ValeurInitiale` et `ValeurFinale` doivent être du même type (tous types scalaires sauf réel, voir chapitre 5).

`VariableDeControle` varie de `ValeurInitiale` à `ValeurFinale` par valeurs successives (croissantes avec **to**, décroissantes avec **downto**) du type concerné.

`ValeurInitiale` et `ValeurFinale` peuvent être des expressions dont les variables doivent alors être de même type.

`Instruction` peut bien sûr être une instruction composée.

```
Program alphabet;{affiche les caracteres de code ascii 65 à
                  122}
Var lettre:Char;
Begin
  For lettre := 'A' to 'z' do
       Write(lettre);
End.
```

A la différence du Basic, on ne peut pas, sous peine d'erreur dans les calculs, changer dans la boucle la valeur de la variable de contrôle. Si on doit sortir de la boucle avant que la valeur finale ne soit atteinte, il faut, soit employer une boucle **Repeat...until** ou **While...do** (voir ci-dessous), soit employer un **Goto** (très peu recommandé). Si `ValeurInitiale` > `ValeurFinale` avec la syntaxe **to,** ou si `ValeurInitiale` < `ValeurFinale` avec la syntaxe **downto**, la boucle n'est pas exécutée.

En Turbo-Pascal, la variable de contrôle est égale à la valeur finale en fin de boucle, à moins que celle-ci n'ait pas été exécutée, auquel cas la variable de contrôle n'a pas d'affectation (**attention** ceci est différent en Pascal standard).

Les boucles peuvent être imbriquées, comme le montre le programme suivant :

```
Program Table_De_Multiplication;
Var     i,j : Integer;
Begin
  For i :=1 to 9 do
        Begin
            WriteLn ;
            For j := 1 to 9 do WriteLn(i,' x ', j ,' = ', i*j);
        End;
End.
```

Ce programme affiche les tables de multiplication de 1 à 9. Il n'effectue pour l'instant aucune mise en page, c'est pourquoi il défile trop vite pour qu'on puisse lire les premières tables. Nous verrons plus loin comment remédier à cela à l'aide de la fonction **GotoXY**.

La structure While...do

Avec cette structure, on effectue la boucle tant que (While = tant que) la condition d'entrée est vraie. Cette condition est une expression booléenne. La syntaxe est la suivante :

```
While condition do instruction;
```

`Instruction` peut être une instruction composée. Si `condition` est **False** (fausse) à l'entrée, `instruction` n'est pas exécutée. Si `condition` est vraie et le reste indéfiniment, la boucle est infinie, et on n'en sort jamais... Cela signifie que `condition` doit être modifiée par `instruction`.

```
Program Exemple_de_While_qui_ne_s_arrete_jamais;
Var toujours : Boolean;
Begin
  Toujours := True;
  While toujours do
        WriteLn('Pour l''eternite');
End.
```

Cet exemple illustre bien la nécessité de modifier la variable de condition dans la boucle pour pouvoir en sortir.

La structure Repeat...until

Cette structure est semblable à la précédente, à la différence près que la condition est évaluée à la fin de la boucle. La condition d'entrée devient une condition de sortie et donc la boucle est exécutée au moins une fois, quel que soit l'état initial de la condition. Voici la syntaxe :

```
Repeat
  instruction;
  instruction2;
  ..............
Until condition;
```

`Instruction1, instruction2...` seront répétées jusqu'à ce que (Until = jusqu'à) `condition` devienne vraie. Là aussi, il faut que `condition` soit modifiée par une des `instructions` à l'intérieur de la boucle pour qu'on puisse en sortir.

```
Program Exemple_de_Repeat;
Var     Nombre_secret,nb,reponse : Integer;
Begin
  nb := 0;
  Nombre_secret := Trunc(Random(1000));
  WriteLn('J''ai choisi un nombre entre 0 et 1000, trouvez-le');
  Repeat
        WriteLn('Entrez votre essai');
        ReadLn(reponse) ;
        If reponse < Nombre_secret then WriteLn('Trop petit!');
        If reponse > Nombre_secret then WriteLn('Trop grand!');
        nb := nb+1;
  Until reponse = Nombre_secret;
  WriteLn('Vous avez gagné en ',nb,' coups');
End.
```

Vous trouverez facilement vous-mêmes les règles de ce petit jeu.

CHAPITRE 7

LES TYPES DECLARES PAR L'UTILISATEUR

En plus des types scalaires simples que nous avons déjà vus, Turbo-Pascal autorise des types scalaires déclarés.

LES TYPES DECLARES PAR ENUMERATION

On peut définir un type en énumérant toutes les valeurs que pourront prendre les variables associées. Dans la déclaration, ces valeurs sont des identificateurs séparés par des virgules et mis entre parenthèses :

```
type    micro = (Amstrad, Apple, MSX, IBM_PC, Commodore);

        mobilier = (chaise, table, fauteuil, lit, armoire);

        jours = (lundi, mardi, mercredi, jeudi, vendredi,
                 samedi, dimanche);
```

Une variable de type `mobilier` pourra prendre une des valeurs `chaise, table, fauteuil, lit ou armoire`. C'est d'ailleurs de cette façon qu'est défini le type boolean :

```
type Boolean = (False, True);
```

Les valeurs sont ordonnées par la déclaration, et on peut leur appliquer les opérateurs relationnels (**<**, **<=**, **>**, **>=**, **=**, **<>**). Dans les types ci-dessus, on aura :

```
Amstrad<Apple<MSX<IBM_PC<Commodore
```

Un même identificateur ne peut apparaître dans deux énumérations différentes. Après avoir déclaré le type micro ci-dessus, on ne pourrait donc pas définir un type :

```
type OrdinateurCouleur = (Amstrad, Oric, MSX, MO5);
```

car `Amstrad` et `MSX` seraient communs aux deux types.

FONCTIONS PREDEFINIES SUR LES TYPES SCALAIRES

Il y a trois fonctions prédéfinies relatives aux types scalaires :

- **Succ** qui retourne la valeur suivante de la liste : `Succ(jeudi)` retournera `vendredi`. **Succ** fonctionne aussi avec les types simples :
  - entier : `Succ(15)` retourne 16.
  - byte : `Succ(128)` retourne 129.
  - char : `Succ('D')` retourne 'E' (ordre ASCII).

- **Pred** qui retourne la valeur précédente de la liste. **Pred**(lit) retournera fauteuil. **Pred** fonctionne comme **Succ** avec les types simples.

- **Ord** retourne un **entier** donnant la position d'une valeur dans la liste : `Ord(Mercredi)` retournera 2 (la numérotation commence à Ø). De la même façon, **Ord** retourne le code ASCII d'un **Char** : `Ord('A')` retourne 65.

Le **Succ** de la dernière valeur et le **Pred** de la première valeur ne sont pas définis.

Il est très agréable de pouvoir ainsi définir des types particuliers à l'application que l'on veut faire. Cela permet une gestion des variables exactement appropriée à l'usage désiré. Cela est particulièrement vrai pour les variables tableaux ; nous n'étudierons celles-ci qu'au chapitre 12, mais vous pouvez déjà savoir qu'elles sont construites à partir des types scalaires (sauf les réels), tant pour les différentes valeurs qui remplissent le tableau, que, et c'est là une grande force du Pascal, pour les

indices qui repèrent ces valeurs dans le tableau. Ainsi nous verrons que si on fait les déclarations suivantes :

```
Type    Semaine=(lundi, mardi, mercredi, jeudi, vendredi,
                 samedi, dimanche);
        Lieux=(docteur, coiffeur, garage, bureau_de_vente,
               fiancee, parents);
```

On pourra définir un type `Rendez-vous` qui sera un tableau des `Lieux` où se rendre en fonction des jours :

```
Type    Rendez-vous =Array[semaine] of Lieux ;
```

Il ne restera plus qu'à déclarer une variable :

```
Var     Agenda : rendez-vous;
```

pour pouvoir travailler directement dans le tableau !

Les variables de type défini par énumération sont des variables scalaires à part entière, et peuvent parfaitement être utilisées comme compteur de boucles :

```
Program semaine;
type jour = (lundi, mardi, mercredi, jeudi, vendredi, samedi,
             dimanche);
Var     aujourd_hui : jour ;
        jour_de_travail,jour_chome : integer;
Begin
  jour_de_travail :=0
  jour_chome : =0
  For aujourd_hui := lundi to vendredi do
        Begin
            jour_de_travail := jour_de_travail+1;
        End ;

  For aujourd_hui := samedi to dimanche do
        Begin
            jour_chome := jour_chome +1;
        End;
write('Il y a ', jour_de_travail,' jours de travail');
write('et ',jour_chome,'jours de repos dans la semaine');
End.
```

Cependant, il y a un inconvénient de taille...Les variables de type déclaré par énumération ne sont pas acceptées par **Write** et **Read**. Cela signifie qu'on ne peut pas faire d'entrée/sortie directe avec elles. Il y a heureusement un moyen de contourner ce problème en se servant de chaînes et de constantes typées structurées (tableaux) ; nous étudierons cela au chapitre 12.

LE TYPE INTERVALLE

On peut aussi définir un type comme étant un intervalle ordonné de valeurs prises dans un type déjà (ou pré) défini. On se contente alors d'indiquer les limites inférieures et supérieures admises, séparées par les signes .. (deux points successifs). Le type prédéfini byte en est un bon exemple :

```
type byte = 0..255 ;
```

On ne peut pas définir de type intervalle sur les réels. Voici quelques exemples de types intervalles valides :

```
type    heure = 0..60;
        majuscule = 'A'..'Z';
```

et si on suppose définis les types `mobilier` et `jours` ci-dessus :

```
type    salon = chaise..fauteuil;
        travail = lundi..vendredi;
        week_end = samedi..dimanche;
```

Attention toutefois, la vérification d'appartenance à l'intervalle n'a pas lieu systématiquement pour toutes les variables. Cette vérification n'a lieu que si la directive de compilation **{$R+}** est donnée (voir les directives de compilation). Le programme teste alors si la valeur affectée aux variables de type intervalle est bien comprise dans les limites déclarées, et génère une erreur à l'exécution dans le cas contraire. Cette directive ne fonctionne pas si la variable est lue au clavier par un **Read** : prudence donc dans ce cas.

Le type intervalle est un type scalaire ; il peut donc servir à tout ce qui est autorisé pour ces derniers.

LES VARIABLES SCALAIRES

Comme nous l'avons vu au chapitre 4, toutes les variables utilisées dans un bloc doivent être déclarées, avec leur type. Toujours en supposant déclarés les types `jours, mobilier, et micro` ci-dessus, on pourra donc avoir une déclaration de variables :

```
Var     nombre, quantite :  integer;
        condition : Boolean;
        lettre : Char;
        journee : jours;
        meuble : mobilier;
```

Les variables de type intervalle peuvent être déclarées directement, sans que la déclaration de type ne soit obligatoire :

```
Var     age : 0..100;
        caractere :'A'..'z';
```

est équivalent à :

```
Type    vieillesse = 0..100;
        lettre ='A'..'z';
Var     age : vieillesse;
        caractere : lettre;
```

LES CONSTANTES TYPEES SIMPLES

Turbo-Pascal offre une possibilité supplémentaire (non présente en Pascal standard), celle de déclarer des constantes dites typées, structurées ou non. Les constantes typées structurées seront vues au chapitre 12, avec les données structurées. Voyons pour le moment les constantes typées simples.

Une constante typée, comme son nom l'indique, a non seulement une valeur, mais aussi un type. L'intérêt est double :

- On peut déclarer des constantes qui auront un type déclaré précédemment.
- On peut modifier la valeur de la constante typée dans le courant du bloc où elle est définie. Cela revient à dire qu'elle s'utilise comme une variable, avec l'avantage que cette variable est initialisée par la déclaration.

La syntaxe de déclaration est simple :

```
Const  indentificateur : type = valeur;
```

Soit par exemple :

```
Const  prix : Real = 250.69;
       verite : Boolean = true;
       controle : Char = '^G';
       titre : string[30] = 'Le Turbo-Pascal est facile'
```

L'emploi des constantes typées économise de la place en mémoire, car elles ne sont incluses qu'une seule fois dans le code, alors que les constantes sans type le sont à chaque fois qu'elles sont citées. Leur seul désavantage est qu'on peut faire varier des constantes typées qui sont supposées justement rester constantes, ce qui est un comble !

CHAPITRE 8

LE TYPE STRING (CHAINE)

Turbo-Pascal comporte le type prédéfini **string** pour manipuler les chaînes de caractères (ce qui n'est pas le cas en Pascal standard). Ce type n'est cependant pas un type scalaire ; il s'apparente aux types tableaux (**array**, voir chapitre 12), avec la différence que les tableaux ont une longueur fixée par leur déclaration et qui ne pourra pas changer pendant tout le programme, tandis que la longueur des chaînes peut varier dynamiquement, de 0 à la longueur déclarée, par affectation dans le courant du programme. Ceci est une aide précieuse, car il suffit d'indiquer, lors de la déclaration de type, la longueur maximale que la chaîne est susceptible d'atteindre.

La déclaration est de la forme :

```
type    nom = string[10];
        residence = string[50];
var     untel : nom;
        adresse : residence;
```

Les chaînes n'ont pas de longueur par défaut ; il est obligatoire de déclarer une longueur maximale pour chaque chaîne ou type de chaîne utilisé. Il est admis de déclarer directement les chaînes dans la déclaration de variables, sans passer par une déclaration de type; les déclarations ci-dessus sont donc équivalentes à :

```
var     untel : string[10];
        adresse : string[50];
```

Dans tous les cas, la longueur maximale d'une chaîne est de 255 caractères.

LES OPERATEURS ET EXPRESSIONS SUR LES CHAINES

Pour manipuler les chaînes, on emploie des expressions chaînes, composées de constantes chaînes, de variables chaînes et de fonctions chaînes, reliées par des opérateurs. Les fonctions peuvent être, soit prédéfinies, soit utilisateur.

- **L'opérateur d'affectation ":="** est utilisé pour donner une valeur à une variable chaîne. Si on affecte à une variable chaîne plus de caractères que sa déclaration ne le permet, il n'y a pas d'erreur, mais les caractères en trop à droite sont perdus.

  Exemple :
  ```
  untel := 'Bernard';
  adresse := '10, Rue de la Paix';
  ```

- **L'opérateur "+"** est utilisé pour réunir deux chaînes. Cette opération s'appelle **concaténation** (on peut aussi se servir de la fonction **Concat** pour obtenir le même résultat).

  Exemple :
  ```
  Phrase := 'Amstrad' +' est un ordinateur '+'fantastique';
  ```

- **Les opérateurs relationnels** peuvent s'appliquer aux chaînes. Le résultat de l'expression est un **booléen**. Les chaînes sont comparées caractère par caractère, de la gauche vers la droite, selon le code ASCII. Les chaînes sont alors classées d'après le premier caractère qui diffère.

  Exemples:
  `'A' < 'R'` est vrai;
  `'A' < 'a'` est vrai;
  `'Pascal' < 'Pascale'` est vrai;
  `'pascale' < 'Pascal'` est faux;
  `'Pascale ' = 'Pascale'` est faux (attention aux espaces!);
  `'pascal' = 'pascal'` est vrai.

On peut accéder aux caractères qui composent la chaîne par indexation de son nom exactement comme pour les éléments d'un tableau: si on a défini la chaîne `untel := 'Albert'`, `untel[1]` retournera 'A', `untel[2]` retournera `'l'` etc... (le premier caractère est numéroté 1); mais attention, si on indexe un caractère plus loin que la longueur actuelle de la chaîne, le caractère retourné a une valeur aléatoire. La directive de compilation `{$R+}` permet de s'assurer qu'on n'indexe pas une chaîne au delà de son dernier caractère.

Le premier octet d'une chaîne contient sa longueur actuelle; par conséquent, `ord(untel[0])` retounera ici `6` (résultat encore plus facilement trouvé par `length(untel)`, comme nous le verrons dans les fonctions prédéfinies).

On peut aussi forcer un caractère dans la chaîne : si on écrit `untel[2] := 'u';` la chaîne `untel` contiendra `'Aubert'`.

Vous ne pouvez cependant pas allonger une chaîne par cette méthode: `untel[7] := 'e'` n'ajoutera pas de `'e'` à la fin de `'Albert'`.

Le type **string** et le type scalaire **char** sont compatibles. Cela veut dire qu'on peut toujours affecter une valeur caractère à une variable chaîne, et qu'on peut affecter une valeur chaîne à une variable caractère (à condition que la longueur de la chaîne soit exactement un caractère).

CHAPITRE 9

LES PROCEDUDES ET FONCTIONS PREDEFINIES SUR LES CHAINES ET LES TYPES SIMPLES

FONCTIONS ET PROCEDURES SUR LES CHAINES

Dans tous les exemples ci-dessous, on supposera declarées les variables:

```
chaine1 := 'ABCDEFGH';
chaine2 := 'QWERTY';
```

Fonctions

Copy

Copy(mot, pos, nbr)
Retourne une **chaîne** contenant `nbr` caractères de `mot`, à partir de la position `pos`. `mot` est une expression chaîne, `pos` et `nbr` doivent être des expressions entières (`1<pos<255`). Si on demande plus de caractères que `mot` n'en contient, c'est-à-dire si `pos+nbr>length(mot)`, seuls les caractères à l'intérieur de `mot` sont retournés. Si `pos>length(mot)`, une chaîne vide est retournée.
Exemple :
`Copy(chaine1,2,3)` retourne `'BCD'`.

Concat

Concat (mot1, mot2, ... ,motn)
Retourne la **chaîne** qui est la concaténation de mot1, mot2... motn, dans l'ordre où ils se présentent. **Concat** est équivalent à l'opérateur "+". La chaîne résultat ne doit pas dépasser 255 caractères, sous peine d'une erreur à l'exécution.

Length

Length (mot) ;
Retourne un **entier** contenant la longueur de mot (mot est une expression chaîne).
Exemple :
length(chaine1) retourne 8.

Pos

Pos (mot1, mot2) ;
Retourne un **entier** contenant la position de la première occurrence de mot1 dans mot2. Si mot1 n'est pas trouvé, **Pos** retourne 0.
Exemple :
Pos('WE',chaine2) retourne 2.

Procédures

Delete

Delete (mot, pos, nbr) ;
Enlève nbr caractères à la variable chaîne mot à partir de la position pos. nbr et pos doivent être des expressions entières. Il faut que pos soit compris entre 0 et 255 sous peine d'une erreur à l'exécution. Si pos>length(mot), aucun caractère n'est enlevé.
Exemples :
Delete(chaine1,3,2) donne à chaine1 la valeur 'ABFGH'.
Delete(Chaine1,4,2) donne à chaine1 la valeur 'ABCFGH'.

Insert

Insert (mot1, mot2, pos) ;
Insère l'expression chaîne mot1 dans la variable chaine mot2, à la position pos, qui doit être une expression entière inférieure à 255. Si pos > length(mot2), mot1 est concaténé à mot2. Si le résultat dépasse la longueur maximale de mot2, les caractères en trop à droite sont perdus.
Exemple :
Insert('toto',chaine2,4) donne à chaine2 la valeur 'ABCDtotoEFGH'.

Str

Str (valeur, mot) ;
Convertit une valeur mumérique entière ou réelle en une chaîne. Les paramètres de formatage peuvent être utilisés ; ils sont décrits avec les procédures Write et WriteLn, au chapitre 13.

Val

Val(mot, var, erreur);
Convertit la chaîne mot, qui doit représenter un nombre (sous forme entière, décimale, hexadécimale ou scientifique) en une valeur numérique rangée dans la variable var qui doit être entière ou réelle. erreur est une variable entière qu'on doit déclarer avant d'utiliser **Val**(). Elle contiendra 0 en sortie si la conversion a eu lieu sans erreur, ou la position du premier caractère de mot qui a déclenché l'erreur si la conversion n'a pu se faire (si mot n'est pas un nombre ou si le type de var ne peut recevoir la valeur convertie).

LES FONCTIONS MATHEMATIQUES

L'argument des fonctions mathématiques (noté nombre dans les descriptions syntaxiques ci-dessous) peut toujours être une expression entière ou réelle.

Abs

Abs(nombre);
Retourne la valeur absolue de nombre. Le résultat est du même type que l'argument.

ArcTan

ArcTan(nombre);
Retourne l'angle (en radians) dont la tangente est nombre. Le résultat est toujours un **réel**.

Cos

Cos(nombre);
Retourne le cosinus de nombre (radians). Le résultat est toujours un **réel**.

Exp

Exp(nombre);
Retourne l'exponentielle de nombre. Le résultat est toujours un **réel**.

Frac

Frac(nombre);
Retourne la partie fractionnaire de nombre. Le résultat est toujours un **réel**.

Int

Int(nombre);
Retourne la partie entière de nombre. Le résultat est toujours un **réel**.

Ln

Ln(nombre);
Retourne le logarithme népérien de nombre. Le résultat est toujours un **réel**.

Odd

Odd(nombre);
Retourne une valeur booléenne: **True** (vraie) si nombre est impair, **False** (fausse) sinon.

Random **Random;**
Retourne un nombre **réel aléatoire** compris entre 0 et 1.

Random **Random** (nombre) ;
Retourne un **entier** compris entre 0 et nombre, qui doit être un entier.

Round **Round** (nombre) ;
Arrondit le réel nombre à l'entier le plus proche comme suit:
•Si nombre>0, Round(nombre)=Trunc(nombre+0.5)
•Si nombre<0, Round(nombre)=Trunc(nombre-0.5).
Le résultat est bien sûr un **entier**.

Sin **Sin** (nombre) ;
Retourne le sinus de nombre (radians). Le résultat est toujours un **réel**.

Sqr **Sqr**(nombre);
Retourne le carré de nombre. Le résultat est du même type que l'argument.

Sqrt **Sqrt** (nombre) ;
Retourne la racine carrée de nombre. Le résultat est un **réel**.

Trunc **Trunc** (nombre) ;
Enlève la partie décimale du réel nombre. Le résultat est bien sûr un **entier**.

FONCTIONS DIVERSES

Chr **Chr** (nombre) ;
Retourne le caractère de code ASCII nombre, qui doit être une expression entière. Les caractères sont retournés modulo 255 si nombre dépasse cette valeur. Le résultat est de type **char**.

Hi **Hi** (nombre) ;
Retourne un **entier** dont l'octet de poids fort est à 0 et dont l'octet de poids faible contient l'octet de poids fort de nombre.

Keypressed **Keypressed;**
Retourne la valeur booléenne **True** si une touche a été frappée au clavier, **False** sinon. **Keypressed** n'attend pas la frappe. Une boucle d'attente pourrait s'écrire :

```
repeat
until KeyPressed;
```

Le programme s'arrête alors jusqu'à ce qu'une touche soit enfoncée.

Lo — **Lo**(nombre);
Retourne un **entier** dont l'octet de poids fort est à 0 et dont l'octet de poids faible contient l'octet de poids faible de nombre.

Ord — **Ord**(valeur);
Retourne un **entier** donnant la position ordinale de valeur dans son type. valeur peut être de tout type scalaire sauf réel. **Ord** est inutile sur les entiers et les bytes; elle retourne le code ASCII d'un **char** ou la position dans la déclaration de type pour les types par énumération. Ord(première valeur) retoune 0.

Pred — **Pred**(valeur);
Retourne la valeur précédente de valeur, dans l'ordre défini par la déclaration de type. Cette fonction accepte des opérandes de tous les types scalaires. L'ordre est naturel pour les entiers et les bytes, et ASCII pour les **char**. Pred(premier) n'est pas défini.

Sizeof — **Sizeof**(nom);
Retourne le nombre d'octets occupés en mémoire par la variable ou le type nom. Le résultat est de type **entier**.

Succ — **Succ**(valeur);
Retourne la valeur suivante de valeur: fonction inverse de la précédente. Succ(dernier) n'est pas défini.

Swap — **Swap**(nombre);
Echange les octets de poids fort et faible de nombre, qui doit être un **entier**.

UpCase — **UpCase**(lettre);
Retourne la majuscule correspondant à lettre, qui doit être un **char** (ou une chaîne de longueur 1). Si le caractère n'a pas de majuscule, le caractère est retourné inchangé.

PROCEDURES PREDEFINIES

ClrEol — **CrEol;**
Efface toute la fin de la ligne depuis le curseur, sans bouger celui-ci (Clear End of Line).

ClrScr — **ClrScr;**
Efface l'écran, remet le curseur en haut à gauche.

CrtInit — **CrtInit;**
Ecriture de la chaîne d'initialisation définie lors de l'installation (inutile sur Amstrad).

CrtExit — **CrtExit;**
Ecriture de la chaîne de réinitialisation définie lors de l'installation (inutile sur Amstrad).

Delay — **Delay** (`temps`);
Crée une attente d'environ temps x1,5 millisecondes. `temps` doit être un **entier**.

DelLine — **DelLine;**
Efface la ligne où se trouve le curseur et remonte les suivantes vers le haut.

Exit — **Exit;**
Provoque la sortie du bloc en cours d'exécution. Peut être utilisée pour sortir d'une procédure, d'une fonction ou même du programme. Branche le programme à la première instruction suivant le bloc où se trouve **Exit**. Cette instruction ressemble à un **Goto** qui n'aurait pas besoin de label.

FillChar — **FillChar** (`variable, nombre, valeur`)
Remplit `nombre octets`, à partir du premier octet occupé par `variable`, avec la valeur `valeur`. `nombre` est une expression entière, `variable` peut être de tout type, et `valeur` doit être **byte** ou **Char**.

GotoXY — **GotoXY** (`horiz,vertic`);
Déplace le curseur jusqu'à la position de coordonnées spécifiées. Ces coordonnées sont comptées en caractères et en lignes depuis le coin en haut à gauche de l'écran (0,0).

Halt — **Halt;**
Provoque la fin du programme, et le retour sous Turbo-Pascal.

InsLine — **InsLine;**
Insère une ligne à la position du curseur, et déplace les suivantes vers le bas.

LowVideo — **LowVideo;**
Jusqu'à la rencontre de **NormVideo,** tous les caractères sont affichés en vidéo inverse.

Move — **Move** (`variable1,variable2,nombre`);
Recopie un bloc de `nombre` octets pris à partir du premier octet de `variable1` à l'emplacement commençant au premier octet de `variable2`. `variable1` et `variable2` peuvent être de tout type (y compris **array**). `nombre` doit être une expression entière. La procédure gère automatiquement les recouvrements possibles, mais attention à ne pas dépasser la longueur de `variable2`. A utiliser avec précaution.

NormVideo **NormVideo;**
Provoque le retour à l'affichage normal, s'il avait été affecté précédemment par **LowVideo**.

Randomize **Randomize;**
Initialise le générateur de nombre aléatoire.

CHAPITRE 10

LES PROCEDURES ET LES FONCTIONS DE L'UTILISATEUR

DEFINITIONS

Nous avons vu au chapitre 4 où et comment déclarer des procédures ou des fonctions. Nous allons à présent aborder le côté pratique de celles-ci et voir comment l'on s'en sert et à quel moment.

Les procédures et les fonctions se comportent exactement comme de petits programmes à l'intérieur du programme principal. En effet, on peut en Turbo-Pascal reprendre à l'intérieur d'une procédure ou d'une fonction toutes les déclarations permises dans le programme principal. On peut donc y déclarer des étiquettes, des constantes, des types, des variables et, bien sûr, des procédures et des fonctions. L'ensemble de ce qui est déclaré dans une procédure ou une fonction sera interne à celle-ci et invisible par les autres procédures déclarées au même niveau dans le programme principal.

Par exemple, si on effectue la déclaration suivante:

```
Program Exemple;
  Procedure A;
  Const Age=57;
  Type Animal=(Chat,Chien,Oiseau,Poisson);
  Begin
        ...
  End;
```

```
  Procedure B;
  Type Ordinateur=(Amstrad,Oric,MO5,Commodore,Atari);
  Begin
       ...
  End;
Begin
  ...
End.
```

Le type `Ordinateur` n'est pas utilisable dans la procédure A car invisible de celle-ci, de même que la constante `Age` ne sera pas utilisable dans la procédure B.

En revanche, toute déclaration faite dans une procédure ou une fonction pourra servir dans une autre procédure (ou fonction) interne à celle qui contient la déclaration. Ainsi si on prend l'exemple suivant :

```
Program Exemple;
  Procedure A;
  Const Age=57;
  Type Fruit=(cerise,banane,fraise,pomme);
       Procedure B;
       Begin
            ...
       End;
  Begin
       ...
  End;
Begin
  ...
End.
```

La déclaration de type et de constante qui est effectuée dans la procédure A sera utilisable dans la procédure B, car B est incluse (ou définie) dans A.

PROCEDURES

Une procédure se definit donc comme un programme, à ceci près :

- la procédure doit être déclarée par le mot réservé **Procedure** ;

- les procédures peuvent recevoir des paramètres depuis l'extérieur; ceux-ci doivent être déclarés avec leur type dans une parenthèse qui suit le nom de la procédure dans l'ordre où ils seront transmis.

Exemple :

```
Procedure Volume(Hauteur,Largeur,Profondeur : Real);
Procedure Bizarre(i : Integer;Test : Boolean;Nom : String[30]);
```

- Le passage de paramètres est différent pour le programme et les procédures. En particulier, les procédures peuvent recevoir des paramètres par valeur ou par variable, ce qui n'est pas le cas du programme. L'étude en détail du passage des paramètres fera l'objet du chapitre suivant;

- le **End** final d'une procédure est suivi d'un point-virgule (;) alors qu'il est suivi d'un point (.) dans le programme.

FONCTIONS

Pour une fonction, c'est un tout petit peu différent : en effet, une fonction possède un type qui est fixé dès sa déclaration. Ce type doit être scalaire (integer, real, boolean, char, énumération ou intervalle défini par l'utilisateur), chaîne ou pointeur.

De plus, le nom donné à la fonction devra obligatoirement être affecté d'une valeur compatible avec son type au moins une fois avant la fin du bloc exécutable de la fonction. Si l'on compare une fonction à un programme (comme on l'a fait pour une procédure), on peut dire les choses suivantes:

Une fonction se définit donc comme un programme, à ceci près :

- la fonction doit être déclarée par le mot réservé **Function** ;

- une fonction possède un type qui doit être fourni juste après la liste des paramètres. Voici quelques exemples de déclarations de fonctions :

```
Function Surface(Rayon : Real) : Real;
Function Factorielle(Nombre : Integer) : Real;
Function Existe : Boolean;
```

- la syntaxe est identique à celle des procédures à laquelle on adjoint le type du résultat retourné ;

- le passage de paramètres est différent pour le programme et les fonctions. En particulier, les fonctions peuvent recevoir des paramètres par valeur ou par variable (l'étude en détail du passage des paramètres fera l'objet du chapitre suivant) ;

- le nom de la fonction devra obligatoirement être affecté au moins une fois d'une valeur compatible avec son type avant la fin du bloc exécutable de celle-ci ;

- le **End** final d'une fonction est suivi d'un point-virgule (;) alors qu'il est suivi d'un point (.) dans le programme.

Note pour les habitués du Pascal standard : il peut vous paraître curieux que l'on puisse effectuer une déclaration d'étiquette à l'intérieur d'une procédure ou d'une fonction. Cela n'est en effet pas le cas en Pascal classique. En Turbo-Pascal, cette possibilité est offerte au programmeur et un label sera donc considéré par le compilateur comme local à la partie dans laquelle il se trouve déclaré. De ce fait, on ne pourra pas "sauter" du programme principal vers une procédure ou une fonction sur un label déclaré dans cette dernière. Le contraire ne sera pas non plus possible. De toute façon, le **Goto** étant fortement déconseillé et exceptionnel chez le programmeur expérimenté, ceci ne modifira en rien ses habitudes.

UTILITE ET VISIBILITE

Les procédures et les fonctions ont pour but de scinder le programme en parties indépendantes réalisant des tâches diverses. Elles permettent également à plusieurs programmeurs de travailler sur le même projet. Il suffit pour cela qu'ils se mettent simplement d'accord sur les paramètres à passer et à restituer entre les divers modules. En effet, ces diverses parties peuvent communiquer les unes avec les autres grâce à des paramètres qu'elles se transmettent lorsqu'elles s'appellent. La manière dont elles sont déclarées autorise le programmeur à ne laisser certaines procédures ou fonctions visibles que d'une partie des autres. Ceci constitue le principe de visibilité des procédures entre elles.

Prenons un exemple :

```
Program Essai;
  Procedure A;
        Procedure B;
        ...
        Procedure C;
        ...
  ...
  Procedure D;
        Procedure E;
              Procedure F;
              ...
              Procedure G;
              ...
        ...
        Procedure H;
        ...
  Procedure I;
  ...

Begin
  ...
End.
```

Dans cet exemple, les visibilités sont les suivantes :

A	est visible de	A,B,C,D,E,F,G,H,I,PP
B	est visible de	A,B,C
C	est visible de	A,C
D	est visible de	D,E,F,G,H,I,PP
E	est visible de	D,E,F,G,H
F	est visible de	E,F,G
G	est visible de	E,G
H	est visible de	D,H
I	est visible de	I,PP

L'abréviation PP représente **P**rogramme **P**rincipal. On peut remarquer que seules les procédures déclarées dans le programme principal sont visibles par celui-ci. Les procédures déclarées à l'intérieur de procédures déclarées dans le programme principal ne sont pas visibles par celui-ci. Ce principe s'applique également aux procédures elles-mêmes. On remarque également que toute procédure est visible par elle-même : c'est ce qui permet la récursivité.

LA RECURSIVITE

Qu'est ce que la récursivité ?

C'est peut être la question que vous vous posez si vous n'avez jamais utilisé de langage qui la supporte (le Basic n'autorise pas la récursivité). On peut dire qu'un objet est récursif, s'il fait directement ou indirectement référence à lui-même dans sa définition. Cette règle ne s'applique d'ailleurs pas qu'en programmation, et l'on peut définir des objets courants de manière récursive : voici par exemple une définition récursive du mot "ancêtre" : ancêtre : père ou mère ou ancêtre de père ou de mère.

Il est bien évident que cette définition facilite la tâche: elle substitue à la définition la manière grâce à laquelle on peut vérifier si une personne donnée est l'ancêtre d'une autre : on regarde d'abord s'il s'agit de son père ou de sa mère. Si la réponse est non, on regarde si ce n'est pas le père ou la mère de son père ou de sa mère (cela correspond aux quatre grands-parents), etc. jusqu'à ce que la réponse soit oui ou que la date jusqu'à laquelle on est remonté permette d'affirmer que la personne n'est pas notre ancêtre.

Cette définition est de toute façon meilleure que celle du *Petit Robert* qui nous dit: "Ancêtre: personne qui est à l'origine d'une famille dont on descend" (pour Descendant on trouve : "personne qui est issue d'un ancêtre") définition avec laquelle on ne fait que tourner en rond sans rien définir. C'est en fait un cas de récursivité croisée : nous l'étudierons plus loin.

Pour en revenir à notre récursivité "Turbo-Pascalienne", celle-ci consiste à déclarer des procédures ou des fonctions qui font, dans leur définition, référence à

elles-mêmes de manière directe ou indirecte. Prenons l'exemple de la fonction factorielle définie récursivement :

```
Function Fact(Nombre : Integer) : Real;
Begin
  If Nombre=0     Then Fact:=1
                  Else Fact:=Nombre*Fact(Nombre-1)
End;
```

Bien sûr, ceci constitue un exemple dans lequel la récursivité n'a rien d'obligatoire. En effet, cette fonction aurait pu être écrite:

```
Function Fact(Nombre : Integer) : Real;
Var      i : Integer;
         j : Real;
Begin
  j:=1;
  For i:=2 To Nombre Do j:=j*i;
  Fact:=j
End;
```

La définition récursive de la fonction factorielle fait un seul appel à elle-même. Elle sera donc dite directe (car elle fait appel à elle-même directement) et monadique (car elle ne fait qu'un seul appel). Dans la définition de cette fonction, on ne décrit pas l'ensemble des opérations à exécuter mais on dicte simplement à la machine : pour calculer Fact(N), il suffit de multiplier N par Fact(N-1), moyennant quoi la machine doit se débrouiller pour calculer Fact(N-1) et cela toujours en utilisant la même méthode.

Dans ce cas présent, la récursivité n'est pas obligatoire (elle reste seulement une manière élégante et naturelle de définir la fonction) mais il existe des cas où la récursivité fournit une solution rapide à un problème qu'il serait très compliqué de résoudre autrement. Prenons l'exemple (très connu !!!) des tours de Hanoi.

Pour ceux qui ne connaîtraient pas ce jeu, rappelons-en brièvement les règles :

- vous disposez de trois piquets sur lesquels des disques percés en leurs centres peuvent s'empiler. A l'origine, tous les disques sont empilés en ordre de diamètres décroissants sur le piquet numéro 1 et les piquets 2 et 3 sont libres. Le but du jeu est de transporter l'empilement des disques sur le piquet 3 avec les règles suivantes:

- vous n'avez le droit de déplacer qu'un seul disque à la fois ;

- un disque ne peut être empilé que sur un disque de diamètre supérieur.

La récursivité fournit ici une solution élégante. Voici la procédure Hanoi qui se charge d'afficher les divers mouvements des disques lors de la réalisation de ce jeu sur N disques. Nous numéroterons les piquets 1, 2 et 3.

```
Procedure Hanoi(N,Depart,Arrivee : Integer);
Var Intermediaire : Integer;
Begin
  If N=1      then  Write('Déplacer le disque supérieur du
                    piquet',Depart,'au piquet ',Arrivee)
              Else Begin
                    Intermediaire := 6-Depart-Arrivee;
                    Hanoi(N-1,Depart,Intermediaire);
                    Hanoi(1,Depart,Arrivee);
                    Hanoi(N-1,Intermediaire,Arrivee)
                End
End;
```

Le calcul 6-`Depart-Arrivee` permet de trouver le numéro du piquet intermédiaire (vous pouvez le vérifier).

Cette fonction est simple: Au lieu de chercher la solution permettant de résoudre le problème en décrivant toutes les étapes, on fournit à la machine le renseignement suivant :

Pour transporter N plateaux du piquet de départ au piquet d'arrivée, il suffit d'en transporter N-1 du piquet de départ vers le piquet intermédiaire, d'en transporter 1 (le plus gros) du piquet de départ au piquet d'arrivée et de transporter les N-1 plateaux restants du piquet intermédiaire au piquet d'arrivée. La machine, grâce à ces renseignements, résoud le problème sans rechigner.

Le problème que l'on se pose assez vite, lorsqu'on a compris le principe de la récursivité, est de savoir comment la machine fait pour ne pas se tromper dans les paramètres. En effet, lors du premier appel (depuis le programme principal), N vaut un certain nombre, `Depart` vaut 1 et `Arrivee` vaut 3. Lors du deuxième appel (exécuté quant à lui à l'intérieur de la procédure), N ne vaut plus que N-1, `Depart` vaut 1 et `Arrivee` vaut 6-1-3 soit 2. On s'aperçoit rapidement que deux ou trois des paramètres changent à chaque appel, mais conservent à l'intérieur de la procédure le même nom. Comment l'ordinateur fait-il pour s'y retrouver dans toutes ces variables qui, de plus, portent toutes le même nom?

Il faut savoir qu'à chaque appel, le programme crée de manière dynamique trois nouvelles variables (`N, Depart` et `Arrivee`) lors de l'appel et une autre (`Intermédiaire`) lors de la déclaration et qu'il ne s'occupe plus des précédentes. Il essaye alors de résoudre le problème à ce nouveau niveau. S'il y parvient, il élimine les quatre variables créées et retombe sur le niveau précédent. S'il n'y parvient pas, il s'enfonce encore d'un niveau de récursivité et recommence. Toutes ces variables ainsi que les adresses de retour (endroit où il doit continuer lorsqu'il remonte d'un niveau) sont gérées par un système de pile qui s'étend lorsqu'on s'enfonce et qui se rétrécit lorsqu'on remonte.

Pour mieux comprendre ce processus, nous pouvons détailler l'état des diverses variables après chaque appel de la procédure Hanoi pour N valant 3:

[Appel]	N	[Depart]	[Arrivee]	[Affichage]
1	3	1	3	
2	2	1	2	
3	1	1	3	Déplacer le disque de 1 vers 3
4	1	1	2	Déplacer le disque de 1 vers 2
5	1	3	2	Déplacer le disque de 3 vers 2
6	1	1	3	Déplacer le disque de 1 vers 3
7	2	2	3	
8	1	2	1	Déplacer le disque de 2 vers 1
9	1	2	3	Déplacer le disque de 2 vers 3
10	1	1	3	Déplacer le disque de 1 vers 3

Ce cas est relativement simple car la récursivité est directe. Il existe en effet des cas de récursivité dite "croisée" où une procédure A fait appel à une procédure B, cette dernière faisant elle-même appel à la procédure A. En voici un exemple :

```
Procedure A(Nombre : Integer);
Begin
  If Nombre=0     then WriteLn(Nombre)
                  Else B(Nombre-1)
End;
Procedure B(Nombre : Integer);
Begin
  If Nombre=0     then Write(Nombre)
                  Else A(Nombre-1)
End;
```

Le problème de la récursivité croisée (ou indirecte) est encore plus complexe que celui de la récursivité directe. C'est pourquoi, avant de faire un programme utilisant cette technique, vous devez vous assurer qu'il s'arrêtera bien.

Lors de l'utilisation de procédures récursives croisées, il y a nécessairement une des deux procédures qui mentionne une procédure pas encore déclarée. C'est pourquoi la déclaration de cette procédure doit se faire grâce au mot réservé **Forward** qui prévient le compilateur de l'utilisation prochaine d'un objet (procédure ou fonction) pas encore défini. La programmation des procédures ci-dessus se fera donc comme suit :

```
Program Recursivite_Croisee;
Var N : Integer;
  Procedure B (Nombre : Integer); Forward;
  Procedure A (Nombre : Integer);
  Begin
        ...
        B(Nombre-1);
        ...
  End;
  Procedure B;
  Begin
        ...
        A(Nombre-1);
        ...
  End;
Begin
  ...
End.
```

Il est à noter que la déclaration des paramètres de la procédure B se fait en même temps que le **Forward** et ne sera plus répétée ensuite. Cela permet au compilateur de vérifier si les paramètres passés lors de l'appel de la procédure B dans le corps de la procédure A sont corrects.

La récursivité est tentante : elle permet d'escamoter la définition d'un problème en lui substituant simplement une manière de le résoudre. Cependant, il ne faut pas tomber dans l'excès consistant à tout récursiver. Si une procédure ou une fonction est soluble de manière itérative, il faudra l'employer comme telle. Il faut, en effet, garder présent à l'esprit qu'une fonction ou procédure récursive est vorace en temps (gestion dynamique des variables et des adresses de retour) et en place (créations dynamiques de variables). C'est pourquoi, il faut s'assurer qu'un calcul écrit récursivement, se terminera bien au bout d'un temps fini et qu'il ne saturera pas la mémoire.

CHAPITRE 11

LE PASSAGE DES PARAMETRES

LE PASSAGE PAR VALEUR

Nous venons de voir, au chapitre précédent, comment déclarer, construire et exécuter des procédures et des fonctions. Ces petites parties indépendantes les unes des autres constituent, en quelque sorte, l'aboutissement d'un raisonnement cartésien du type : "Lorsqu'un problème est complexe, il suffit de le découper en petites parcelles, qui, elles, sont simples à résoudre." Des paramètres sont nécessaires pour exécuter ces procédures ou ces fonctions (mais cela n'a rien d'obligatoire); en effet, un programme comme celui qui suit fonctionne très bien avec une procédure n'ayant pas besoin de paramètre :

```
Program Exemple_Sans_Parametre;
Var  Resultat_General : Integer;
  Procedure ajouter;
  Begin
       Resultat_General:=Resultat_General+1;
       WriteLn('Resultat dans procedure : ',Resultat_General)
  End;
Begin
  Resultat_General:=27;
  ajouter;
  WriteLn('Resultat general : ',Resultat_General)
End.
```

Ce programme affiche les résultats suivants :

```
Resultat dans procedure : 28
Resultat general : 28
```

Cet exemple est très simple et n'est présenté qu'à titre pédagogique ; il donne tout de même l'illustration parfaite de ce qui n'est pas possible dans un programme de type Basic: si vous modifiez une variable dans un sous-programme, elle est aussi modifiée dans le programme principal. Dans ce cas, la variable `Resultat_general` est dite globale: elle est utilisable dans tout le programme.

Turbo-Pascal offre la possibilité de ne modifier un paramètre que localement (par opposition à globalement) si on le désire. Tapez le programme que voici :

```
Program Exemple_Avec_Parametre_Transmis_Par_Valeur;
Var  Globale : Integer;
  Procedure ajouter(Locale : Integer);
  Begin
       Locale:=Locale+1;
       WriteLn('Resultat dans procedure : ',Locale)
  End;
Begin
  Globale:=27;
  ajouter(Globale);
  WriteLn('Resultat dans programme : ',Globale)
End.
```

La différence est que maintenant un paramètre `Locale` apparaît dans la déclaration de la procédure.Vous obtiendrez à l'exécution les résultats suivants :

```
Resultat dans procedure : 28
Resultat dans programme : 27
```

La variable nommée `Globale` n'est plus modifiée; voici ce qu'il se passe :

- avant l'appel, la variable `Globale` vaut 27 et la variable `Locale` n'existe pas (puisqu'elle n'est pas mentionnée dans la partie déclaration du programme principal) ;

- à l'appel de la procédure, la variable `Locale` est créée dynamiquement et on lui affecte la valeur de la variable `Globale`, à savoir 27. Au début de la procédure, `Locale` vaut donc 27 ;

- pendant le déroulement de la procédure, la variable `Locale` est modifiée (l'affichage de `Locale` le prouve), mais ceci n'affecte en rien la variable `Globale` ;

- après la sortie de la procédure, lorsqu'on retourne au programme principal, la variable `Globale` n'a pas changé et vaut donc toujours 27 lorsqu'on la réécrit sur l'écran. La variable `Locale` n'existe plus; elle a été "oubliée" à la sortie de la procédure puisqu'elle était interne à celle-ci ;

Ce principe s'appelle le passage de paramètre par valeur. Il est essentiel pour la compréhension de la suite, il faut donc que vous l'ayez parfaitement assimilé avant de poursuivre. Si ce n'est pas le cas, il serait souhaitable que vous repreniez ce chapitre depuis le début.

Le passage de paramètre par valeur s'accompagne de la création dynamique de nouvelles variables. En ce sens, les noms que l'on donne aux variables, sont sans importance et le programme précédent pourrait également s'écrire :

```
Program Exemple_Avec_Parametre_Transmis_Par_Valeur;
Var  Machin : Integer;
  Procedure ajouter(Machin : Integer);
  Begin
       Machin:=Machin+1;
       WriteLn('Resultat dans procedure : ',Machin)
  End;
Begin
  Machin:=27;
  ajouter;
  WriteLn('Resultat dans programme : ',Machin)
End.
```

Les résultats suivants sont alors affichés :

```
Resultat dans procedure : 28
Resultat dans programme : 27
```

c'est-à-dire les mêmes que précédemment. On s'aperçoit donc qu'une variable prend le dernier nom déclaré. Cela correspond à la logique des choses, puisque sinon, les résultats fournis dépendraient des noms que l'on donne aux variables qui sont créées dynamiquement. Ce serait contraire à l'esprit modulaire du Pascal grâce auquel on peut travailler à plusieurs sur une même application sans s'occuper les uns des autres.

Bien sûr, le nombre de paramètres que l'on passe à une procédure n'est pas limité à un. On peut en transmettre autant que l'on veut, à condition de spécifier les types de chacun des paramètres transmis. Ainsi, une procédure nécessitant comme paramètre trois entiers et deux réels pourra être déclarée comme suit :

```
Procedure bidule(A,B,C : Integer;D,E : Real);
```

L'appel de la procédure devra substituer aux paramètres formels A,B,C,D et E une liste de paramètres effectifs compatibles avec eux. Vous devrez donc faire un appel de la procédure bidule du type :

```
Bidule(Titi,Toto,Tutu,Riri,Roro);
```

où `Titi, Toto` et `Tutu` devront être de type **integer** et `Riri` et `Roro` de type **real** (la compatibilité des types sera contrôlée au moment de la compilation). L'affectation des valeurs des paramètres effectifs aux paramètres formels est établie par la position des paramètres dans ces listes. Cependant, seuls certains types sont admis pour le passage des paramètres : il s'agit des types prédéfinis de Turbo-Pascal et des types définis par l'utilisateur; en particulier, les variables de type structuré ne peuvent être transmises que par le biais d'un nouveau type introduit par l'utilisateur. Ainsi, si vous avez une matrice colonne à transmettre vous ne devrez pas faire comme ceci :

```
Program Tres_Mauvais;
Var Tableau : Array[1..10] Of Integer;
  Procedure Incorrecte(Vecteur : Array[1..10] Of Integer);
  Begin
      ...
  End;
Begin
  ...
  Incorrecte(Tableau);
  ...
End.
```

Mais obligatoirement comme cela :

```
Program Bon;
Type Tab = Array[1..10] Of Integer;
Var Tableau : Tab;
  Procedure Correcte(Vecteur : Tab);
  Begin
      ...
  End;
Begin
  ...
  Correcte(Tableau);
  ...
End.
```

Cela peut vous paraître idiot car il s'agit exactement du même programme où l'on a déclaré un nouveau type et où l'on a remplacé `Array[1..10] Of Integer` par `Tab`. Cependant, si vous ne faites pas comme dans le deuxième exemple, le compilateur produira une erreur car il est incapable de comprendre les types structurés dans une déclaration de paramètres. Il faut obligatoirement que le type ait déjà été défini (soit au sein du Turbo-Pascal, soit par l'utilisateur). C'est d'ailleurs une des raisons d'être des types.

C'est de cette façon que vous pourrez transmettre n'importe quel paramètre à une procédure ou à une fonction. Lors de l'appel de la procédure `Correcte`, une nouvelle variable, `Vecteur` de type `Tab`, dans laquelle seront recopiées une à une toutes les cases de la variable `Tableau`, sera créée dynamiquement.

LE PASSAGE PAR VARIABLE

Par opposition au passage par valeur, il existe en Turbo-Pascal une autre manière de passer les paramètres : le passage par variable. Il a pour but de répercuter les modifications effectuées sur les paramètres formels aux paramètres effectifs transmis lors de l'appel. Les variables correspondantes du bloc appelant sont donc modifiées. Ainsi, la procédure ou la fonction peuvent avoir une influence sur des variables qui lui sont extérieures et qui lui sont transmises. Il est précisé par le mot réservé **Var** avant le nom du paramètre formel.

Reprenons notre petit programme en lui transmettant le paramètre par variable :

```
Program Exemple_Avec_Parametre_Transmis_Par_Variable;
Var  Globale : Integer;
  Procedure ajouter(Var : Locale:Integer);
  Begin
       Locale:=Locale+1;
       WriteLn('Resultat dans procedure : ',Locale)
  End;
Begin
  Globale:=27;
  ajouter(Globale);
  WriteLn('Resultat dans programme : ',Globale)
End.
```

Si vous exécutez ce programme, vous obtiendrez :

```
Resultat dans procedure : 28
Resultat dans programme : 28
```

Le paramètre `Globale` est donc modifié dès que l'on touche au paramètre `Locale`. Ceci est dû au passage par variable du paramètre `Globale`. Le processus en est le suivant :

- avant l'appel, la variable `Globale` vaut 27 et la variable `Locale` n'existe pas (puisqu'elle n'est pas mentionnée dans la partie déclaration du programme principal);

- à l'appel de la procédure, la variable `Locale` est créée mais ce n'est pas une "vraie" variable : il s'agit en fait d'une manipulation qui consiste à renvoyer tout ce qui concernera la variable `Locale` sur la variable `Globale`. En d'autres termes pour les connaisseurs, la variable `Locale` n'est qu'un pointeur pointant sur la variable `Globale`. En ce sens ces deux variables sont équivalentes et toute modification de la variable `Locale` sera en réalité une modification de la variable `Globale`. Tout se passe exactement comme si on avait simplement redonné un nouveau nom à la variable `Globale`. C'est pourquoi, à la sortie de la procédure, la variable `Globale` a été modifiée.

Si vous avez bien compris ce fonctionnement, vous êtes en droit de vous demander ce que le passage de paramètre par variable apporte en plus de l'utilisation d'une variable globale. Plusieurs réponses sont à apporter à cette question :

- le passage de paramètre par variable sert d'abord à conserver le principe de modularité : si j'écris une procédure et vous le programme principal, nous pouvons travailler indépendamment dès l'instant que nous nous sommes mis d'accord sur les paramètres à transmettre et à restituer, puisque le passage par variable permet de renommer celles-ci, et donc de ne pas s'occuper de ce qui se passe à l'extérieur de la partie que l'on écrit ;

- d'autre part, il peut arriver que l'on désire que seulement certaines procédures modifient les variables globales. Dans ce cas, le recours au passage par variable est indispensable. Regardons par exemple le programme que voici :

```
Program Exemple;
Var  Machin : Integer;
  Procedure A(Var : Nombre1 : Integer);
  Begin
       ...
  End;
  Procedure B(Nombre2 : Integer);
  Begin
       ...
  End;

Begin
  ...
  A(Machin);
  B(Machin);
  ...
End.
```

Dans cet exemple, la procédure A répercute les modifications subies par la variable `Nombre1` sur la variable `Machin`, mais à l'inverse, la procédure B n'altère pas la variable `Machin`. Ceci montre bien l'utilité du passage de paramètre par variable.

Bien sûr, de même qu'une procédure ou une fonction peut accepter un ou plusieurs paramètres, le mode de passage de ces paramètres est laissé au choix du programmeur. On peut ainsi mélanger les variables passées par valeur et celles passées par variable. En voici un exemple :

```
Program Exemple_de_passage_melange;
Var X,Y : Integer;
  Procédure Carre(N1 : Integer; Var N2 : Integer);
  Begin
       N1:=N1*N1;
       N2:=N2*N2;
       WriteLn(N1,N2)
  End;
```

```
Begin
  X:=3;
  Y:=5;
  WriteLn(X,Y);
  Carre(X,Y);
  WriteLn(X,Y)
End.
```

L'exécution de ce dernier programme donne les résultats suivants :

```
3     5
9     25
3     25
```

Seule le variable Y a été modifiée car elle seule a été passée par variable. La variable N1 n'a pas répercuté ses modifications sur X car X a été transmise par valeur.

EFFETS DE BORDS ET VISIBILITE

Comme nous venons de le voir, certains paramètres passés à des procédures ou des fonctions peuvent, ou non, influencer les variables extérieures. Ainsi, si l'on passe un paramètre par variable, nous avons vu que la variable passée était modifiée lorsqu'on sortait de la procédure ou de la fonction. C'est ce qu'on appelle les effets de bords. Ils peuvent être ou non souhaités par le programmeur. Pour bien comprendre les effets de bords il faut connaître les règles suivantes : une variable globale est visible dans tout le programme et peut à ce titre être modifiée n'importe où dans celui-ci. Cependant, s'il existe, dans certaines procédures ou certaines fonctions, une variable portant le même nom, c'est cette dernière qui sera modifiée au sein de la partie dans laquelle elle se trouve déclarée, et non la variable globale. Voyons l'exemple suivant :

```
Program Bizarre;
Var glob : Integer;
  Procedure A;
  Var glob : Integer;
  Begin
       glob:=glob+1
  End;
  Procedure B;
  Begin
       glob:=glob+1
  End;
Begin
  glob:=7;
  A;
  WriteLn(glob);
  B;
  WriteLn(glob)
End.
```

Ce programme retourne les résultats suivants :

```
7
8
```

La variable `glob` déclarée dans le programme principal est visible partout. Cependant, une variable de nom identique est déclarée au sein de la procédure A. C'est donc cette dernière qui sera modifiée (notez que l'on ne sait pas ce qu'elle vaut car on ne l'initialise pas au début de la procédure A). Dans la procédure B, aucune variable de même nom n'est déclarée, ce sera donc la variable `glob` déclarée dans le programme principal qui sera modifiée.

D'une manière générale, lorsqu'il y a plusieurs variables possédant le même nom, visibles au sein d'une partie P, c'est la variable qui a été déclarée ou référencée en dernier dans une partie (P ou incluant P) qui est prise en compte. Analysons le programme `Essai` pour illustrer ceci :

```
Program Essai;
Var glob : Integer;
  Procedure A;
  Var glob : Integer;
       Procedure B;
       Var glob : Integer;
       Begin
            ...
       End;
       Procedure C;
       Begin
            ...
       End;
  Begin
       ...
  End;
  Procedure D;
  Begin
  ...
  End;
Begin
  glob:=17;
  A;
  D;
End.
```

On s'aperçoit que :

- la procédure A ne modifiera pas la variable `glob` déclarée dans le programme principal car elle inclut la déclaration d'une variable portant le même nom. Si la variable `glob` est modifiée dans le corps de la procédure A, c'est celle qui est déclarée dans la procédure A qui le sera ;

- de même, la procédure B modifiera la variable `glob` qu'elle déclare mais ne pourra modifier ni la variable `glob` déclarée dans la procédure A, ni celle déclarée dans le programme principal ;

- si l'on modifie la variable `glob` dans la procédure C, c'est celle qui est déclarée dans la procédure A qui le sera. En effet, aucune variable n'est déclarée dans la procédure C; on fait donc référence à la variable déclarée dans la dernière partie incluant la procédure C, à savoir A ;

- si la procédure D utilise une variable `glob`, elle fera référence à celle déclarée dans le programme principal, puisque la procédure D n'inclut pas de déclaration de variable portant le nom `glob` ;

On peut compliquer un petit peu (!) en utilisant des procédures qui nécessitent des paramètres. Pour vous exercer à cette gymnastique sur les variables, essayez de voir les résultats que va fournir le programme suivant (si vous ne trouvez pas, tapez-le et exécutez-le).

```
Program Un_Peu_Plus_Difficile;
Var Compliquee : Integer;
  Procedure A(Var Truc : Integer);
  Var Compliquee : Integer;
      Procedure B(Machin : Integer);
      Var Compliquee : Integer;
      Begin
          Compliquee:=7;
          Machin:=Machin+1;
          WriteLn(Compliquee,Machin)
      End;
      Procedure C(Var Bidule : Integer);
      Begin
          Bidule:=Bidule+1;
          WriteLn(Bidule);
          B(Bidule);
          WriteLn(Bidule)
      End;
  Begin
      Compliquee:=17;
      B(Compliquee);
      WriteLn(Compliquee);
      C(Truc);
      WriteLn(Truc)
  End;
Begin
  Compliquee:=23;
  A(Compliquee);
  WriteLn(Compliquee)
End.
```

Faites attention, il y a des passages par valeur, des passages par variable et des déclarations multiples. Cet exemple reste cependant assez simple. On pourrait le compliquer à l'extrême en ajoutant d'autres niveaux de procédures et d'autres passages de paramètres. Ce n'est cependant pas nécessaire car si vous avez trouvé les résultats fournis par ce programme, vous êtes apte à comprendre des cas plus complexes encore.

QUE FAUT-IL UTILISER, ET QUAND?

Une question importante reste en suspens : quels types de variables et de passages de paramètres faut-il utiliser ? Comme vous vous en doutez, il n'existe pas de réponse générale à cette question. Nous allons voir comment éviter certains pièges :

Les variables globales

Les variables globales sont presque toujours à déconseiller. En effet, il est très rare d'avoir une variable qui serve dans tout un programme et que chaque partie ait le droit de modifier. De plus, si dans une procédure on utilise des variables globales, on ne voit pas directement quelles sont les variables utilisées; alors que si les variables sont transmises comme paramètres à la procédure, on cerne d'un seul coup toutes les variables nécessaires à l'exécution de cette procédure. En effet, dans un programme bien construit, on doit pouvoir comprendre une procédure en dehors de son contexte, ce qui est évidemment impossible si elle fait référence à des identificateurs extérieurs. De plus, il est préférable de ne pas avoir de variables globales à cause des effets de bords indésirables qui pourraient se produire.

Les variables locales

Ce sont elles qui sont conseillées, car elles évitent les effets de bords et n'existent plus lorsqu'on sort de la procédure où elles ont été déclarées. De plus, les variables locales permettent à plusieurs personnes de travailler sur un même programme sans se préoccuper de ce que font les autres. Même pour des variables "sans importance" (les variables servant aux boucles par exemple), il est préférable que celles-ci soient déclarées dans les diverses procédures pour conserver la modularité. Ainsi le programme `Bon` sera meilleur du point de vue de la modularité que le programme `Mauvais` :

```
Program Bon;
Var i : Integer;
  Procedure A;
  Var i : Integer;
  Begin
       For i:=1 To 10 Do …
  End;
```

```
  Procedure B;
  Var i : Integer;
       Procedure C;
       Var i : Integer;
       Begin
            i:=10;
            Repeat
                 …
                 i:=i-1;
            Until i:=0;
       End;
  Begin
       i:=1;
       While i<100 Do
            Begin
                 …
                 i:=i+1
            End
  End;
Begin
  For i:=1 To 100 Do …

  …
End.
```

```
Program Mauvais;
Var i : Integer;
  Procedure A;
  Begin
       For i:=1 To 10 Do …
  End;
  Procedure B;
       Procedure C;
       Begin
            i:=10;
            Repeat
                 …
                 i:=i-1;
            Until i:=0
       End;
  Begin
       i:=1;
       While i<100 Do
            Begin
                 …
                 i:=i+1
            End
  End;
```

```
Begin
  For i:=1 To 100 Do …
  …
End.
```

Même si l'on fait attention à bien contrôler la variable `i`, `Bon` sera toujours meilleur que `Mauvais`. Dans le programme `Bon`, la variable `i` est locale à toute procédure qui l'utilise . Dans le programme `Mauvais`, on est obligé de faire attention à ce que vaut `i` à l'entrée et à la sortie de chaque procédure. De plus, dans `Mauvais`, le travail doit être fait par une seule personne, au courant de ce qui se passe à l'extérieur alors que les diverses procédures du programme `Bon` peuvent être écrites par des programmeurs différents et indépendants.

Le passage de paramètre par valeur

Le passage de paramètre par valeur est à conseiller lorsque l'on désire éliminer tous les effets de bords. Chaque procédure ou fonction peut alors être écrite et testée seule sans s'occuper de ce qui se passe "au dehors". C'est ce mode de passage associé aux déclarations locales que nécessite la transparence des diverses unités d'un même programme. Cependant, il faut savoir que ce mode est vorace en temps : en effet, lors de l'appel d'une procédure avec passage de paramètres par valeur, nous avons vu qu'il y avait création dynamique de nouvelles variables et recopie des anciennes valeurs dans celles-ci. Cela s'accompagne d'une perte de temps, surtout si les variables prennent beaucoup de place en mémoire. Par exemple, si une procédure nécessite une variable du type tableau, il faudra faire attention à ne pas saturer la mémoire. Prenons un exemple :

- supposons que vous disposiez de 64 Ko,
- supposons que votre procédure ait comme paramètre un tableau 50 par 50 de réels,

Sachant qu'un réel est codé sur 6 octets (cf chapitres 15 et 16) 50x50x6 octets seront nécessaires à chaque appel pour créer ce tableau, à savoir 14,6 Ko de mémoire. Si en plus votre procédure est récursive, elle arrivera très rapidement à mettre "à genoux" votre Amstrad au niveau mémoire. De plus, le temps de recopie du tableau à chaque appel sera très important.

Il faut donc prendre garde à ne pas saturer votre système, aussi bien au niveau vitesse qu'au niveau temps d'exécution. Ceci pourra être évité par le passage par variable que nous allons voir tout de suite.

Le passage par variable

Le passage par variable est requis tout particulièrement lorsqu'une procédure ou une fonction doit transmettre des modifications de variable à l'extérieur. C'est ce mode qui sera choisi plutôt que celui de la variable globale car il correspond plus à l'esprit dans lequel a été conçu Turbo-Pascal. De plus, il pourra être utilisé dans

certains cas où le passage par valeur nécessiterait trop de temps d'exécution et/ou trop de place mémoire. Il faudra cependant vérifier si la procédure est bien transparente vis-à-vis du paramètre transmis par variable. Ce mode est rapide (puisqu'il n'y a pas recopie des valeurs dans les nouvelles variables créées dynamiquement) et économique en mémoire (puisqu'il n'y a pas création de nouvelles variables mais seulement de pointeurs).

CHAPITRE 12

LES DONNEES STRUCTUREES

Jusqu'à présent, nous nous sommes contentés de définir des données de types simples. Il existe en Turbo-Pascal, la possibilité de définir des types dits "structurés". Il s'agit des types tableau, ensemble et enregistrement. De même, nous verrons qu'il est également possible de définir des constantes structurées et de les initialiser.

LES TABLEAUX (ARRAY)

Un tableau est une manière assez naturelle de ranger des éléments de même type. Il permet de regrouper ceux-ci dans une stucture fixe et d'accéder à chaque élément par le biais d'un ou plusieurs indices.

Les tableaux à une dimension

Un tableau à une seule dimension est une structure dans laquelle peut être rangé un certain nombre (fixé dès la déclaration) d'éléments d'un certain type (lui aussi fixé lors de la déclaration). On peut alors y ranger ou y lire des éléments. Un type tableau se déclare par le mot réservé **Array**. Lors de la déclaration d'un tableau, on précise le nombre maximal de "cases" que l'on alloue au tableau et le type des éléments qui y figureront. Le nombre d'éléments est spécifié grâce au type d'index et le genre des éléments le sera à l'aide du type de base. On accède par la suite à des éléments qui sont du type de base grâce à un indice qui est du type d'index. Le type d'index est nécessairement de type scalaire (prédéfini ou déclaré par l'utilisateur) et le type de base est quelconque (prédéfini ou défini par l'utilisateur).Voici par exemple comment on déclarera un tableau de dix entiers :

```
Type  Tableau = Array[1..10] Of Integer;
Var   Tab : Tableau;
```

Ici le type d'index est `1..10` et le type de base est **Integer**. On nommera le troisième élément de ce tableau **`Tab[3]`**. On pourra affecter toute valeur de type compatible avec **Integer** à cette case, et on pourra récupérer son contenu par une affectation à une variable de type **Integer** ou de type incluant les entiers. Le type d'index n'est pas nécessairement numérique, il doit seulement être scalaire ; il serait également possible de construire ce tableau ainsi :

```
Type  Tableau = Array['A'..'J'] Of Integer;
Var   Tab : Tableau;
```

On accèderait alors au troisième élément du tableau Tab par **`Tab[C]`**.

Le type de base est également laissé au choix du programmeur; celui-ci peut par exemple déclarer un texte comme suit :

```
Type  Texte = Array[1..32000] Of Char;
Var   Txt : Texte;
```

Il accèdera alors au 15678^{e} élément de son texte par **`Txt[15678]`**.

Les types définis par l'utilisateur sont également admis pour le type de base ou pour le type d'index (pour ce deuxième cas, il faut qu'il soit scalaire); ils doivent avoir été déclarés plus haut, c'est une des raisons d'être de la déclaraton de type. On peut par exemple écrire :

```
Type  Fruit :
      (Banane,Fraise,Framboise,Abricot,Kiwi,Mangue,Ananas);
      Cocktail : Array[1..3] Of Fruit;
```

Une variable de type `Cocktail` sera alors un tableau de trois `Fruits`.

La définition du type **String** est en fait un **Array**. En effet, on a implicitement la déclaration suivante :

```
Type String = Array[0..255] Of Integer;
```

Cette définition permet d'accéder directement au n^{e} caractère d'une chaîne comme on le ferait avec un tableau.

Les tableaux à plusieurs dimensions

Nous avons vu dans la partie précédente que le type de base pouvait être quelconque. *A fortiori*, il peut être lui-même de type tableau. On pourra donc avoir la déclaration suivante :

```
Type Tableau = Array[1..10] Of Array[1..10] Of Integer;
```

Ceci définirait un tableau de dix tableaux de dix entiers, c'est-à-dire un tableau dix-dix d'entiers. Si l'on déclare une variable `Tab` de type tableau, le quatrième élément du septième tableau sera alors noté `Tab[7][4]`.

Pour simplifier la notation, le tableau ci-dessus pourra être déclaré :

```
Type Tableau = Array[1..10,1..10] Of Integer;
```

Et si l'on déclare une variable `Tab` de type `Tableau`, le quatrième élément du septième tableau sera alors noté `Tab[7,4]`.

On pourra définir ainsi des tableaux avec autant de dimensions que nécessaire et contenant des éléments d'un type quelconque. La seule restriction à apporter est que les types d'index doivent tous être scalaires. Voici pour vous familiariser quelques déclarations:

Jeu d'échecs :

```
Type  Piece = (Roi,Reine,Tour,Fou,Cavalier,Pion,Vide);
      Echiquier = Array['A'..'H',1..8] Of Piece;
Var   Jeu : Echiquier;
```

Ceci définit une variable jeu que l'on pourra remplir, tel un échiquier grâce à des pièces qui sont définies précédemment. On pourra au cours du programme, déplacer une tour de A1 à A4 en faisant tout simplement :

```
Jeu['A',4]:=Tour;
Jeu['A',1]:=Vide;
```

Rubik's cube :

```
Type  Position    = (Haut,Bas,Devant,Derriere,Droite,Gauche);
      Couleur     = (Rouge,Vert,Bleu,Jaune,Orange,Blanc);
      Face        = Array[1..3,1..3] Of Couleur;
      Cube        = Array[Haut..Gauche] Of Face;
Var   Rubik       : Cube;
```

On pourrait ainsi multiplier les exemples à l'infini. C'est à vous d'apprécier avec rigueur votre problème pour pouvoir choisir la structure de donnée la plus adéquate.

Note pour les habitués du Pascal standard : le mot **Packed** n'est pas employé en Turbo-Pascal. En effet, le compilateur optimise la place de manière systématique. **Packed** peut être présent dans un source, mais il sera sans effet sur la manière dont le compilateur organisera ses données.

LE TYPE ENSEMBLE (SET OF)

Déclarations

Les ensembles constituent un type de données structurées particulier. Celui-ci permet souvent de résoudre élégamment des problèmes alors qu'un programme utilisant des types plus "classiques" serait lourd à écrire et à manipuler. La déclaration d'un type ensemble se fait grâce au mot réservé **Set Of** suivi du type de base qui doit être scalaire non réel. Par exemple :

```
Type  Couleur  = (Bleu,Blanc,Rouge,Vert,Orange,Violet,Jaune);
      Drapeau  = Set Of Couleur;
Var   Pavillon : Drapeau;
```

Dans cet exemple, on déclare un type drapeau comme étant un ensemble de couleurs.

Les opérateurs et expressions sur les ensembles

Vous pouvez, tout comme en mathématiques, effectuer des opérations sur les ensembles que vous définissez en Turbo-Pascal : affectation, inclusion, intersection, union, appartenance, rien n'y manque. Nous allons voir en détail ces opérations.

- **L'affectation(:=)** : après avoir fait les déclarations ci-dessus, on pourra faire les affectations suivantes :

```
Pavillon:=[Bleu,Blanc,Rouge];
Pavillon:=[Vert];
Pavillon:=[Vert,Orange,Violet,Jaune,Blanc,Rouge];
Pavillon:=[];
```

On s'aperçoit que le nombre d'éléments d'une variable de type ensemble n'est pas fixé. On peut aussi bien lui affecter l'ensemble entier que l'ensemble vide ([] dans ce dernier cas). Les valeurs possibles d'une variable de type ensemble sont tous les ensembles que l'on peut former à partir du type de base, y compris l'ensemble vide. L'affectation se fait par l'ensemble des valeurs entouré par des crochets ([...]). L'ensemble vide est noté []. On peut également se servir de ".." dans une affectation. Pour l'exemple précédent, on aurait pu avoir l'affectation:

```
Pavillon:=[Bleu..Vert,Jaune];
```

Cette affectation est équivalente à :

```
Pavillon:=[Bleu,Blanc,Rouge,Vert,Jaune];
```

De plus, l'ordre dans lequel apparaissent les éléments dans une affectation est sans importance. Ainsi, les affectations suivantes `Pavillon:=[Bleu,Jaune]` et `Pavillon:=[Jaune,Bleu]` sont équivalentes. Cela signifie que le type ensemble n'est pas ordonné; les fonctions **Ord**, **Pred**, et **Succ** sont interdites.

Les notations des types intervalle et ensemble se ressemblent beaucoup, parenthèses pour intervalle, et crochets pour ensemble. Il y a là une source fréquente d'erreurs.

- **L'intersection (+)** : l'intersection de deux ensembles est la partie commune de ces deux ensembles. Ainsi :

```
[Bleu,Jaune,Rouge]+[Orange,Jaune,Blanc,Rouge] vaut [Jaune,Rouge]
[1..56] + [34..76] vaut [34..56]
['A'..'Y'] + ['J'..'P'] vaut ['J'..'P']
```

- **L'union (*) :** l'union de deux ensembles est la réunion des divers éléments de ces deux ensembles. Par exemple :

```
[Bleu,Jaune] * [Orange,Bleu,Rouge] vaut
                        [Bleu,Jaune,Orange,Rouge]
[17..34] * [1..18] vaut [1..34]
```

- **La différence(-)** : la différence d'un ensemble A et d'un ensemble B est l'ensemble des valeurs appartenant à A mais pas à B:

```
[Bleu,Jaune,Rouge] - [Jaune,Vert,Orange] vaut [Bleu,rouge]
['D'..'W'] - ['A'..'F'] vaut ['G'..'W']
```

- **L'égalité(=)** : l'égalité de deux ensembles n'est vraie que si tous les éléments du premier se trouvent dans le second et vice versa. L'ordre n'a pas d'importance. Par exemple :

`[Bleu,Rouge] = [Rouge,Bleu]` est une expression booléenne vraie.
`[1..7] = [2..8]` est une expression booléenne fausse.

- **L'inclusion(<= ou >=)** : l'inclusion de A dans B est vraie si tous les éléments de A se trouvent dans B :

`[5..9] <= [1..10]` est une expression booléenne vraie.
`['A'..'T'] => ['A'..'V']` est une expression booléenne fausse.

- **L'appartenance (In)** : un élément appartient à un ensemble si sa valeur fait partie de l'ensemble. Par exemple :

  Si `Reponse` vaut 'o' alors
  `Reponse` **`In`** `['O','N','o','n']` est une expression booléenne vraie.
  Si `Entier` vaut `17` alors
  `Entier` **`In`** `[45..67]` est une expression booléenne fausse.

Toutes ces opérations permettent de résoudre facilement certains problèmes. On peut par exemple faire la scrutation du clavier jusqu'à ce que l'utilisateur réponde par oui ou par non :

```
Repeat
  Read(Kbd,Reponse);
Until UpCase(Reponse) In ['O','N'];
```

Le clavier est scruté jusqu'à ce que l'utilisateur appuie sur la touche O ou la touche N, que le mode majuscule soit actif ou non.

LES ENREGISTREMENTS (RECORD)

Jusqu'à présent, nous avons regroupé des données en tableaux ou en ensembles. Ces structures ne permettent cependant pas beaucoup de fantaisie puisque les données d'un tableau ou d'un ensemble doivent être de même type. Il peut vous arriver d'avoir besoin d'une structure dans laquelle les divers éléments ne sont pas de même type. Si vous voulez établir une structure du type "carte d'identité", vous avez des parties alphanumériques (adresse), numériques (taille, poids) et alphabétiques (nom, prénom). Votre structure devra donc être apte à manipuler des chiffres, des lettres, etc. Turbo-Pascal permet de rassembler des éléments disparates dans une structure nommée **Record** (enregistrement). Il ne faut pas croire qu'un enregistrement sera obligatoirement écrit sur la disquette. Il le sera si vous le désirez, sinon il résidera en mémoire. On définit dans un enregistrement les diverses parties avec leur type respectif :

```
Type Alpha = Record
                 Champ1:Type1;
                 .........
                 Champ n:Type n;
             End;
```

Les diverses parties s'appellent des champs. Les types peuvent être soit prédéfinis, soit définis par l'utilisateur. Voici un exemple où l'on crée un enregistrement pour un nombre complexe, composé d'une partie réelle et d'une partie imaginaire :

```
Type Complexe =   Record
                      Reelle  : Real;
                      Imag    : Real;
                  End;
```

Dans cet enregistrement, les deux champs sont d'un type prédéfini. Cet enregistrement aurait aussi pu être déclaré comme suit :

```
Type Complexe =   Record
                      Reelle,Imag : Real;
                  End;
```

Voici une déclaration d'un enregitrement de type date où un des types utilisés est défini par l'utilisateur :

```
Type  Month = (Janvier,Fevrier,Mars,Avril,Mai,Juin,Juillet
              ,Aout,Septembre,Octobre,Novembre,Decembre);
      Date =  Record
                  Jour  :1..31;
                  Mois  :Month;
                  An    :Integer;
              End;
Var   Aujourd_hui: Date;
```

L'affectation des valeurs aux différents champs d'une variable de type enregistrement se fait par une opération du type :

```
Variable.Champs:=Valeur
```

Dans l'exemple ci-dessus, on pourra faire les affectations suivantes :

```
Aujourd_hui.Jour:=15;
Aujourd_hui.Mois:=Octobre;
Aujourd_hui.An:=1985;
```

Ce qui nous donne la date du 15 octobre 1985.

Les types des divers champs étant laissés au choix du programmeur, ceux-ci peuvent être également des types structurés. En particulier, ce peut être des enregistrements. En voici un exemple :

```
Type  Personne =  Record
                      Nom          : String[20];
                      Prenom    : String[30];
                      Sexe      : (Masculin,Feminin);
                  End;
      Caract =    Record
                      Taille    : Real;
                      Poids     : Real;
                  End;
```

```
        Adresse =   Record
                        Numero    : Integer;
                        Rue         : String[40];
                        Depart    : 1..95;
                        Ville     : String[30];
                    End;
        Fiche   =   Record
                        Citoyen   : Personne;
                        Carac     : Caract;
                        Domicile  : Adresse;
                    End;
Var     Rens : Fiche;
```

Dans ce cas, les affectations seront faites de la manière suivante :

```
Rens.Citoyen.Nom:='Dupont';
Rens.Citoyen.Prenom:='Jean';
Rens.Citoyen.Sexe:=Masculin;
Rens.Carac.Taille:=1.75;
Rens.Carac.Poids:=90;
Rens.Domicile.Numero:=17;
Rens.Domicile.Rue:='Rue des petits oiseaux';
Rens.Domicile.Depart:=75;
Rens.Domicile.Ville:='Paris';
```

Cela peut être fastidieux à écrire surtout si les noms des champs sont longs. Or vous devez toujours écrire `Rens.` pour accéder à un champ de la variable `Rens`. Turbo-Pascal dispose donc de l'instruction **With..do** pour vous éviter de répéter `Rens`. L'écriture des affectations ci-dessus deviendra alors :

```
With Rens do Begin
                Citoyen.Nom:='Dupont';
                Citoyen.Prenom:='Jean';
                ...
                Domicile.Depart:=75;
                Domicile.Ville:='Paris';
             End;
```

Mais ce qui est applicable à `Rens`. l'est aussi à `Citoyen., Carac.` et `Domicile,` et l'on peut finalement écrire la même affectation sous la forme :

```
With Rens,Citoyen,Caract,Domicile Do
     Begin
         Nom:='Dupont';
         Prenom:='Jean';
         ...
         Depart:=75;
         Ville:='Paris';
     End;
```

Cette notation est beaucoup plus concise lorsque l'on a affaire à des enregistrements à plusieurs niveaux. C'est donc celle-ci que l'on emploiera de préférence.

Jusqu'à présent, nous n'avons défini que des enregistrements de structure fixe. Or on peut, au sein d'un enregistrement, introduire une partie variable. Cette introduction se fait grâce à l'instruction **Case** qui permet de définir les champs d'une variable en fonction d'un selecteur de champ. Mais attention, les conditions suivantes sont à respecter :

- la partie variable d'un enregistrement devra être unique (elle pourra en revanche comporter un nombre quelconque de cas) et devra obligatoirement se trouver après la partie fixe de l'enregistrement ;

- le selecteur de champ devra être de type scalaire (non Real), spécifié dans l'instruction **Case** ;

- si le sélecteur de champ donne un nom de variable, il sera considéré comme un champ obéissant aux mêmes règles que les autres.

Voici un exemple de déclaration d'enregistrement avec une partie variable :

```
Type  Statut = (Celibataire,Marie,Divorce,Veuf);
      Date   = Record
                  Jour          :1..31;
                  Mois          :1..12;
                  An            :1900..2000;
               End;
      Perso  = Record
                  Nom           : String[30];
                  Prenom        : String[30];
               Case Sit_Fam     : Statut Of Celibataire: ();
                     Marie      : (Date_Mar : Date);
                     Divorce    : (Date_Mari,Date_Div:Date);
                     Veuf       : (Date_Maria,Date_Deces:
                                    Date);
               End;
```

Le **Case** étant nécessairement à la fin, il n'y a qu'un seul **End** qui ferme le **Case** et le **Record**. De plus, vous pouvez remarquer que dans le **Case**, tous les cas sont pris en compte, même s'il n'y a rien à faire. En effet, il faut impérativement que toutes les possibilités soient énumérées pour que le compilateur ne génère pas d'erreur. Dans notre exemple, c'est pour "célibataire" que l'on ne fait rien. On le signale alors par des parenthèses vides.

LES CONSTANTES STRUCTUREES

On dispose en Turbo-Pascal de la possibilité de déclarer des constantes de type structuré. La déclaration et l'initialisation s'effectuent exactement comme pour une constante typée simple. Par exemple, on peut déclarer et initialiser une constante qui soit le tableau de vérité du OU exclusif logique comme ceci :

```
Const      Ou_Exclusif    : Array[0..1,0..1] Of 0..1
                          = ((0,1),(1,0));
```

On dispose alors du tableau de vérité du Ou exclusif. Pour obtenir le résultat du Ou exclusif de 1 et de 0, il suffit de faire :

```
Resultat:=Ou_Exclusif[1,0]
```

Il faut bien sûr que la variable `Resultat` soit d'un type compatible avec celui des éléments se trouvant dans la constante structurée (ici, un type compatible avec 0..1).

On peut aussi utiliser des constantes structurées de type **Array**, mais aussi **Set** ou **Record**. En voici un exemple :

```
Type  Lettre = 'A'..'Z';
Const  Voyelles : Set Of Lettre
               =['A','E','I','O','U','Y']
```

définit une constante `Voyelles` comme un ensemble de lettres. On utilise également ici la possibilité de définir un type et de se servir de celui-ci pour la déclaration d'une constante, (cette possibilité n'existe pas en Pascal standard où la déclaration de constante précède obligatoirement le déclaration de type). Voyons un autre exemple avec une constante **Record** :

```
Type Coordonnees  =   Record
                        Rho,Teta,Phi:Real;
                      End;
Const      Origine :  Coordonnees
                   =  (Rho:0,Teta:0,Phi:0);
```

Cela permet au programmeur une grande liberté dans les constantes utilisables, ce qui n'est pas possible en Pascal standard.

Naturellement, les constantes peuvent également être définies à partir de nouveaux types. En voici un exemple :

```
Type  Animal  = (Homme,Chat,Chien);
      Donnees = (Hauteur,Poids);
Const Caracteristique : Array[Homme..Chien,Hauteur..Poids] Of
                          Integer
                        =((175,70),(20,5),(60,15));
```

Pour obtenir la hauteur moyenne d'un chien, il suffit alors d'accéder à l'élément `Caracteristique[Chien,Hauteur]` , par exemple en affectant cette valeur à une variable de type compatible avec Integer.

L'utilisation des constantes structurées permet de nombreuses possibilités. En effet, elles peuvent servir à une tâche telle que celle de transformer une valeur d'un type énumération en une chaîne de caractères. En effet, vous avez vu que le type énumération permettait des affectations, des comparaisons, etc, mais en aucun cas, il ne permet l'affichage, comme chaîne de caractères, de la valeur d'une variable. A ce problème, la déclaration de constantes structurées offre une solution des plus élégantes. Supposons que votre type énumération soit le suivant :

```
Type  Fleur = (Marguerite,Jonquille,Rose,Lilas,Paquerette);
```

Rien ne vous empêche alors d'effectuer la déclaration de constante suivante :

```
Const  Nom : Array[Marguerite..Paquerette] Of String[10]
           = ('Marguerite','Jonquille','Rose','Lilas',
              'Paquerette');
```

Quoi de plus facile maintenant que d'afficher la valeur d'une variable de type `Fleur`. Si votre déclaration est comme suit :

```
Var    Fl : Fleur;
```

Pour afficher la chaîne de caractères correspondant à votre variable, il suffira de faire :

```
WriteLn(Nom[Fl]);
```

ce qui affichera `Rose` si `Fl` vaut `Rose`, `Lilas` si `Fl` vaut `Lilas` etc.

Il est à noter que l'appellation de constante n'est pas très appropriée. En effet, si vous essayez de faire "varier" une constante typée ou structurée, le compilateur ne génèrera pas d'erreur. Vous pouvez très bien au cours de votre programme, changer la valeur d'une constante typée ou typée structurée. Plus que de constantes, il s'agit en fait de variables qui sont initialisées au début du programme.

Note pour les habitués du Pascal standard : la déclaration de constante de type structuré n'existe pas dans le Pascal standard. En Turbo-Pascal, cette possibilité est offerte au programmeur et comporte de nombreux avantages. En particulier, elle évite d'avoir recours à la clause **Value** pour initialiser des variables de type structuré que l'on emploiera comme des constantes dans la suite du programme. De plus, à cette possibilité est adjointe celle de définir un type avant une constante (l'ordre dans lequel on effectue les déclaration en Turbo-Pascal est sans importance). Cela permet une liberté totale pour les déclarations de constantes.

CHAPITRE 13

LES FICHIERS

Jusqu'à présent quand nous avons parlé de fichiers, il s'agissait d'informations stockées sur disquettes : programmes ou données. Dans ce sens, Turbo-Pascal ou CP/M sont des fichiers.

Mais Turbo-Pascal a lui aussi la notion de fichier et elle diffère un peu de cette définition.

NOTION DE FICHIER ET DIFFERENCE AVEC ENREGISTREMENT

Un fichier Turbo-Pascal est un canal grâce auquel on communique avec les organes d'entrée/sortie de l'Amstrad. Ce peut être une communication de type sortie (écran ou imprimante), de type entrée (clavier) ou de type entrée/sortie (fichier sur diquette). D'une manière générale, un fichier est un lien de communication entre l'ordinateur et un organe extérieur.

Il existe des différences fondamentales entre le fichier et l'enregistrement :

- le fichier est constitué d'un ensemble de données de même type, alors que l'enregistrement est une collection d'éléments, sans restriction de type;

- le fichier s'adresse nécessairement à un organe extérieur (le plus souvent l'unité de disquettes) alors que l'enregistrement ne manipule que des informations en mémoire;

- une variable de type fichier peut contenir autant de composantes qu'on le désire sans que ce nombre ne soit fixé dans la déclaration.

Il existe en Turbo-Pascal plusieurs sortes de fichiers. Pour commencer, nous allons aborder les fichiers sur disquette car ce sont eux qui correspondent à la notion habituelle de fichier.

FICHIERS SEQUENTIELS

Un fichier sur disquette est un ensemble de données de même type. On peut à loisir lire ces données, les modifier ou en ajouter. Voyons tout d'abord comment on déclare un fichier.

Le fichier se déclare par le mot réservé **File Of** suivi du type des composants. Par exemple, un type "fichier d'entiers" peut se déclarer :

```
Type Fich = File Of Integer;
```

Une variable du type `Fich` pourra alors servir à effectuer des opérations sur un fichier de ce type. La déclaration de cette variable sera :

```
Var Objet : Fich;
```

Pour pouvoir travailler sur un fichier, il faut passer obligatoirement par deux phases :

- assigner un nom de travail à un fichier externe ;
- ouvrir ce fichier soit pour une lecture, soit pour une réécriture.

Vous devez fournir ce qu'on appelle un nom de travail (nom logique) qui remplacera le nom effectif (nom physique) dans toute les opérations concernant les fichiers. C'est la procédure **Assign** qui permet de donner un nom de travail à un fichier externe. Elle nécessite deux paramètres qui sont :

- le nom de travail (ou nom logique);
- le nom physique qui sera celui sous lequel le fichier se trouvera sur le support magnétique.

Par exemple, pour déclarer un fichier `Objet` qui se trouvera sous le nom `ESSAI.INT` sur le disque A nous ferons :

```
Assign(Objet,'A:Essai.Int');
```

Par la suite, chaque fois que nous voudrons communiquer avec le fichier A:ESSAI.INT, nous l'appellerons Objet. Bien entendu, il faut qu'Objet corresponde au fichier dont on se sert. Ainsi, si le fichier A:ESSAI.REL est un fichier de réels et que vous désiriez travailler dessus, vous devrez le déclarer comme :

```
Type Fich =   File Of Real;
Var Objet :   Fich;
```

Une fois que vous avez assigné un nom au fichier extérieur, vous devez le préparer (on dit souvent l'ouvrir), soit pour la lecture (et éventuellement l'ajout d'éléments), soit pour la création (ou la réécriture).

L'ouverture se fait par l'instruction **Reset**. Le paramètre est le nom de travail. Par exemple, **Reset**(Objet) préparera le fichier Objet pour la lecture.

La création se fait par l'instruction **ReWrite**. Le paramètre est le nom de travail. Ainsi, **ReWrite**(Objet) préparera le fichier Objet pour une écriture ou une réécriture.

Voici un petit programme qui permet d'ouvrir un fichier nommé ESSAI.INT sur la disquette A et qui prépare ce fichier pour l'écriture.

```
Program Essai_d_ecriture;
Type  Fich    = File Of Integer;
Var   Objet   : Fich;
Begin
  Assign(Objet,'A:ESSAI.INT');
  ReWrite(Objet);
```

Bien sûr, ce n'est que le début du programme. Celui ci n'est pas encore compilable, puisqu'il ne possède même pas de **End** final.

Nous avons décidé d'écrire des entiers dans le fichier ESSAI.INT. Nous allons donc demander à l'utilisateur de les introduire au clavier, puis nous les écrirons sur la disquette. A la fin, nous fermerons le fichier avant de terminer le programme.

L'écriture d'une valeur dans un fichier se fait grâce à l'instruction **Write**. Attention car la syntaxe diffère du **Write** qui sert à écrire un message à l'écran. Le "**Write** fichier" nécessite deux paramètres : le nom du fichier de travail et la valeur qu'il faut écrire. Ainsi, l'écriture de la valeur d'une variable entière nommée Nombre dans le fichier "Objet" se fait par **Write**(Objet,Nombre).

Voici donc un programme qui enregistre dans le fichier déclaré tout à l'heure des nombres entiers. Après chaque nombre introduit, le programme demande si l'on veut continuer ou non. Si la réponse est n ou N, le processus s'arrête.

En fin de programme, nous fermerons le fichier par l'instruction **Close** qui nécessite comme paramètre le nom du fichier à fermer.

```
Program Essai_d_ecriture;
Type  Fich     = File Of Integer;
Var   Objet    : Fich;
      Nombre   : Integer;
      Reponse  : Char;
Begin
  Assign(Objet,'A:ESSAI.INT'); {Assigne un nom logique}
  ReWrite(Objet);  {Reecrit le fichier Objet}
  Repeat
       WriteLn('Entrez un entier');
       Read(Nombre);
       Write(Objet,Nombre);{Ecrit Nombre dans le fichier Objet}
       WriteLn('Encore ? (O/N)');
       ReadLn(Reponse);
  Until UpCase(Reponse)='N';
  Close(Objet) {Ferme le fichier Objet}
End.
```

Vous pouvez taper ce petit programme pour vérifier qu'il fait bien ce qu'on lui demande. Cependant, il ne faut pas vous inquiéter si le témoin de fonctionnement du lecteur de disquettes ne s'allume pas à chaque fois que vous fournissez un entier. En effet, pour éviter de perdre du temps inutilement, Turbo-Pascal ne décide d'écrire physiquement sur le disque que lorsqu'il dispose d'une certaine quantité de données. Il s'agit d'un **buffer**, nous en reparlerons.

Maintenant que vous avez créé un fichier d'entiers sur votre disquette, il pourrait être utile de pouvoir le relire. On utilisera pour l'avoir en lecture, l'instruction **Reset** dont nous avons déjà parlé.

De même que l'instruction **Write** permettait d'écrire (moyennant deux paramètres), l'instruction **Read** permet de lire. Les paramètres sont les mêmes que pour **Write**, à savoir le nom logique et la variable. On pourrait penser que ces éléments suffisent à construire un programme susceptible de relire les entiers que nous avons stockés sur la disquette avec le précédent programme. Il n'en est rien : en effet, à chaque lecture de nombre (par **Read**), le système "avance" un pointeur d'une position pour pouvoir lire l'élément suivant. Qu'advient-il si l'on décide d'avancer plus loin que la fin du fichier ?

Si vous essayez de le faire, vous aurez une erreur d'exécution. Pour y remédier, vous avez la possibilité de tester si l'on est arrivé à la fin du fichier grâce à la fonction **Eof** qui nécessite le nom logique du fichier comme paramètre. Si le pointeur est à la fin du fichier `Objet`, **Eof(**`Objet`**)** vaut **True**. Le programme de lecture des données, introduites précédemment avec affichage sur l'écran, sera donc :

```
Program Essai_de_lecture;
Type  Fich    = File Of Integer;
Var   Objet   : Fich;
      Nombre  : Integer;
Begin
  Assign(Objet,'A:ESSAI.INT'); {Assigne un nom logique}
  Reset(Objet);    {Ouvre le fichier pour lecture}
  While Not(Eof(Objet)) Do   Begin
                             Read(Objet,Nombre);
                             Write(Nombre,' ** ')
                             End;
  Close(Objet) {Ferme le fichier Objet}
End.
```

Ce programme lit les entiers présents dans le fichier `Objet` et les affiche à l'écran. Cependant, vous pouvez remarquer qu'il accède aux nombres dans l'ordre dans lequel ils ont été entrés. Ainsi, si nous voulons accéder au 3^{e} élément de la liste, nous sommes obligés de lire d'abord les deux qui le précèdent. C'est ce qu'on appelle un fichier séquentiel car on accède aux éléments en séquence. Aux fichiers séquentiels s'opposent les fichiers à accès direct qui sont également possibles en Turbo-Pascal. Nous les verrons juste après avoir résumé l'ensemble des fonctions que nous avons vues jusqu'à présent (ce sont les plus utiles) :

Fonction

Eof

Eof (`Nom`) ;
Teste si la fin du fichier `Nom` est atteinte. Si c'est le cas, Eof (`Nom`) vaut **True**, sinon elle vaut **False**.

Procédures

Assign

Assign(`Nom_Logique,Nom_Physique`) ;
Assigne `Nom logique` qui sera utilisée par la suite dans toutes les fonctions qui manipulent ce fichier. Cette instruction est obligatoire avant de travailler sur un fichier.

Close

Close (`Nom`) ;
Ferme le fichier `Nom` et interdit les opérations d'entrée/sortie dessus jusqu'à réouverture par **Reset** ou **ReWrite**.

Read

Read (`Nom,Variable`) ;
Lit la valeur se trouvant dans le fichier `Nom` et l'affecte à

`Variable`, puis avance d'une unité pour la prochaine lecture.

Reset

Reset (`Nom`) ;
Ouvre le fichier `Nom` pour lecture. Si l'on tente d'écrire, on écrase ce qui se trouve à notre position dans le fichier. Si l'on se trouve à la fin du fichier et que l'on écrit, on ajoute des éléments au fichier `Nom`.

ReWrite

ReWrite (`Nom`) ;
Crée le fichier `Nom` et le prépare en écriture. Si le fichier `Nom` existait déja, **ReWrite** l'efface d'abord et le prépare pour une nouvelle écriture.

Write

Write (`Nom,Variable`) ;
Ecrit la valeur de `Variable` dans le fichier `Nom` et avance d'une unité pour la prochaine écriture.

FICHIERS A ACCES DIRECT

Reprenons tout d'abord le programme de création d'un fichier d'entiers.

```
Program Essai_d_ecriture;
Type    Fich    = File Of Integer;
Var     Objet   : Fich;
        Nombre  : Integer;
        Reponse : Char;
Begin
   Assign(Objet,'A:ESSAI.INT');     {Assigne un nom logique}
   ReWrite(Objet); {Reecrit Objet}
   Repeat
        WriteLn('Entrez un entier');
        Read(Nombre);
        Write(Objet,Nombre);   {Ecrit Nombre dans Objet}
        WriteLn('Encore ? (O/N)');
        ReadLn(Reponse);
   Until UpCase(Reponse)='N';
   Close(Objet)     {Ferme Objet}
End.
```

Supposons maintenant que vous le lanciez et que vous rentriez les valeurs suivantes (en tapant la touche **<RETURN>** entre chacune d'elles) :

```
17,24,37,97,9876,32,3,789,123,543,0
```

Il existe une méthode permettant d'accéder au n^e nombre enregistré dans ce fichier. C'est ce qu'on appelle un accès direct. Celui-ci s'effectue grâce à la procédure **Seek**. Celle-ci nécessite deux paramètres : le nom logique du fichier et le numéro de l'élément auquel on veut accéder. Ce numéro vaut 0 pour le premier élément, ce qui fait que pour accéder au n^e nombre, il faudra spécifier n-1. Ainsi pour accéder au 9^e élément (qui est ici 123) vous devrez faire **Seek(**`Objet,8`**)**. Voici une lecture directe du fichier créé ci-dessus :

```
Program Essai_de_lecture_directe;
Type    Fich     = File Of Integer;
Var     Source   : Fich;
        Nombre   : Integer;
        Reponse  : Char;
Begin
   Assign(Source,'A:ESSAI.INT');   {Assigne un nom logique}
   Reset(Source);  {Ouvre Source pour lecture}
   Repeat
        Repeat
           WriteLn('Introduisez le numero d''ordre de l''entier
                   desiree);
           ReadLn(Nombre);
        Until Nombre<FileSize(Source);{Teste si Nombre pas
                                       trop grand}
        Seek(Source,Nombre);{Positionne le pointeur sur ordre
                             demande}
        Read(Source,Nombre); {Lit element demande}
        WriteLn('C''est ',Nombre);
        WriteLn('Encore ? (O/N)');
        ReadLn(Reponse)
   Until UpCase(Reponse)='N';
   Close(Source);  {Ferme Source}
End.
```

Ce programme permet donc l'accès direct à un élément quelconque du fichier. Cependant, il faut faire attention à ne pas se positionner après la fin physique du fichier. Ce contrôle doit être fait par le programmeur. C'est pourquoi, nous avons eu recours à la fonction **FileSize** qui permet de savoir combien d'éléments se trouvent dans un fichier : **FileSize(**`Source`**)** fournit le nombre d'entiers se trouvant dans le fichier `Source`. Nous verrons plus loin qu'il existe une autre fonction (**IOResult**) qui peut également résoudre ce genre de problème. La fonction **FileSize** fournit comme résultat le nombre d'éléments se trouvant dans le fichier, quels qu'ils soient (entier, réels, etc.). On peut donc s'en servir pour se positionner à la fin du fichier de manière à y rajouter des éléments : cela est réalisé par **Seek(**`Source`**, FileSize (**`Source`**))**. Voici un petit programme qui permet d'adjoindre des éléments supplémentaires au fichier `ESSAI.INT` que nous avons déjà créé.

```
Program ajout;
Type    Fich      = File Of Integer;
Var     Objet     : Fich;
        Nombre    : Integer;
        Reponse   : Char;
Begin
   Assign(Objet,'A:ESSAI.INT');
   ReSet(Objet);    {Ouvre Objet pour lecture/ecriture}
   Seek(Objet,FileSize(Objet));     {Positionne le pointeur a la
                                     fin de Objet}
   Repeat
        WriteLn('Entrez un entier');
        Read(Nombre);
        Write(Objet,Nombre);   {Ecrit Nombre}
        WriteLn('Encore ? (O/N)');
        ReadLn(Reponse);
   Until UpCase(Reponse)='N';
   Close(Objet)
End.
```

Ce programme vous permet d'ajouter des éléments au fichier déjà créé jusqu'à ce que vous décidiez d'arrêter.

Les quatre petits programmes que nous venons de voir sont les quatre parties principales d'une gestion de fichiers. Nous avons en effet :

- la création d'un fichier;
- la lecture globale d'un fichier;
- la lecture directe d'une fiche;
- l'ajout de fiche.

Certes, nous avons pris ces exemples avec des fichiers d'entiers. Mais la démarche aurait été identique si cela avait été des fichiers de réels, de chaînes de caractères ou même d'enregistrements. Il suffit de remplacer toutes les déclarations d'entiers servant dans le fichier par des déclarations appropriées. Nous donnerons d'ailleurs un programme complet de gestion de carnet d'adresses à la fin de cette partie.

Auparavant, voici le résumé des dernières fonctions utilisées ainsi que quelques autres fonctions opérant sur les fichiers. Celles-ci sont d'un usage moins courant que celles que nous venons de voir, mais elles peuvent peut-être vous servir dans des cas particuliers.

Fonctions

FilePos

FilePos (Nom_Logique)
Fournit le numéro d'ordre de l'élément sur lequel se trouve le pointeur de fichier. Ce numéro est un entier et vaut 0

pour le premier élément. Cette fonction ne peut pas s'appliquer à un fichier texte (voir partie sur les fichiers texte).

FileSize — **FileSize** (`Nom_Logique`) ;
Fournit la taille en nombre d'éléments du fichier `Nom_Logique`. Cette fonction ne peut pas s'appliquer à un fichier texte (voir partie sur les fichiers texte).

Procédures

Erase — **Erase** (`Nom_Logique`) ;
Efface le fichier ayant pour nom de travail `Nom_logique`. Celui-ci peut être indifféremment ouvert ou fermé avant effacement.

Flush — **Flush** (`Nom_Logique`)
Cette procédure écrit "physiquement" ce qui se trouve dans le buffer. Nous avons vu en effet que Turbo-Pascal, pour éviter d'utiliser l'unité de disquettes sans cesse, n'écrivait qu'au bout d'un certain temps lorsqu'on remplissait un fichier avec des éléments. Ainsi, un buffer est créé en mémoire, et Turbo-Pascal ne lance d'opération d'écriture physique que lorsque ce buffer est plein. **Flush** permet au programmeur de lancer cette opération avant qu'elle n'ait lieu "naturellement". Cela l'assure du fait que ses données ont été écrites physiquement sur le disque et que lors de la prochaine opération de lecture, les données seront lues physiquement. Cette procédure ne peut pas s'appliquer à un fichier texte (voir partie sur les fichiers texte).

Rename — **Rename**(`Nom_Logique,Nouveau_Nom`);
Renomme le fichier de nom de travail `Nom_Logique`. `Nouveau_Nom` est le nouveau nom physique du fichier. Le programmeur doit fermer un fichier avant de le renommer et doit s'assurer qu'il n'existe pas déjà un fichier ayant pour nom physique `Nouveau_Nom`.

Seek — **Seek**(`Nom_Logique,Nombre`);
Positionne le pointeur de fichier sur l'élément du numéro d'ordre `Nombre`. Ce dernier doit être égal à **FileSize**(`Nom_Logique`) au plus. Cette procédure ne peut pas s'appliquer à un fichier texte (voir partie sur les fichiers texte).

Voici, à présent, un programme reprenant les divers modules déjà créés, mais adaptés non plus aux entiers mais à un carnet d'adresses. Ces divers modules seront réunis en un seul pour éviter les manipulations de disquettes.

```
Program Gestion_Adresse;
Const   Defini : False;
Type    Personne   = Record
                     Nom ,Prenom : String[20];
                     Adresse     : String[60];
                     Telephone   : String[12];
                   End;
        Fichier    = File of Personne;
Var     Carnet : Fichier;

   Procedure Definir;
   Var  Nom : String[20];
   Begin
        ClrScr;
        WriteLn('Entrez le nom du fichier');
        Read(Nom);
        Assign(Carnet,Nom);
        {$I-}
        Reset(Carnet);
        {$I+}
        If IOResult<>0 Then ReWrite(Carnet);
        Defini:=True;
        Close(Carnet);
   End;

Procedure Ajouter;
Var  Ami : Personne;
     Rep : Char;
Begin
     Reset(Carnet);
     Seek(Carnet,FileSize(Carnet));
     Repeat
         ClrScr;
         Write('Nom : ');
         ReadLn(Ami.Nom);
         Write('Prenom : ');
         ReadLn(Ami.Prenom);
         Write('Adresse : ');
         ReadLn(Ami.Adresse);
         Write('Telephone : ');
         ReadLn(Ami.Telephone);
         WriteLn;
         Write('Encore ? (O/N) ');
         Repeat
            Read(Kbd,Rep);
         Until UpCase(Rep) In ['O','N'];
     Until UpCase(Rep)='N';
     Close(Carnet)
End;
```

```
Procedure Affichage(Ami : Personne);
Begin
      WriteLn('Nom : ',Ami.Nom);
      WriteLn('Prenom : ',Ami.Prenom);
      WriteLn('Adresse : ',Ami.Adresse);
      WriteLn('Telephone : ',Ami.Telephone);
      WriteLn;
      WriteLn('Appuyez sur une touche pour continuer');
      Repeat Until KeyPressed;
End;

Procedure LireTout;
Var   Mec : Personne;
Begin
      Reset(Carnet);
      While Not(Eof(Carnet)) Do
            Begin
                ClrScr;
                Read(Carnet,Mec);
                Affichage(Mec);
            End;
      Close(Carnet);
End;

Procedure LireUneFiche;
Var   Gars : Personne;
      Nom  : String[20];
Begin
      ClrScr;
      WriteLn('Introduisez le nom de la personne desiree');
      ReadLn(Nom);
      While Not(Eof(Carnet)) Do

            Begin
                ClrScr;
                Read(Carnet,Gars);
                If Gars.Nom=Nom Then Affichage(Gars)
            End;
      Close(Carnet)
End;

Procedure Erreur;
Begin
      GotoXY(1,15);
      WriteLn('Vous n''avez pas defini de fichier de
             travail');
      WriteLn;
      WriteLn('Appuyez sur une touche pour continuer');
      Repeat Until KeyPressed;
End;
```

```
Procedure Choisir;
Var Reponse : Char;
Begin
      Repeat
          ClrScr;
          WriteLn('  **  Choisissez une option  **');
          WriteLn;
          WriteLn('D --->    Definir   le   fichier   de
                             travail');
          WriteLn('A --->    Ajouter des fiches');
          WriteLn('L --->    Lire tout le fichier');
          WriteLn('R --->    Retrouver un nom');
          WriteLn('S --->    Sortir du programme');
          Repeat
             Read(Kbd,Reponse);
          Until UpCase(Reponse) In ['D','A','L','R','S'];
          Case UpCase(Reponse) Of
                 'D' : Definir;
                 'A' : If Defini Then Ajouter Else Erreur;
                 'L' : If Defini Then LireTout Else Erreur;
                 'R' : If Defini Then Retrouver Else
                       Erreur;
                 'S' :   ClrScr;
          End;
          Until UpCase(Reponse) = 'S';
    End;

 Begin
    ClrScr;
    Choisir;
 End.
```

Certes, cela n'est pas dBase III ! Mais c'est un bon début. On aborde la plupart des notions importantes de la gestion de fichiers. Certaines instructions peuvent vous paraître obscures dans ce listing. Ce sont **IOResult** et **Read(**`Kbd, reponse`**)**. Nous allons voir leur signification dans les parties nommées "Les organes logiques" et "Vérifications des entrées/sorties". Auparavant, nous allons étudier un type de fichier particulier nommé fichier texte.

LES FICHIERS TEXTE

Les fichiers texte constituent une catégorie particulière car certaines opérations de Turbo-Pascal ne s'appliquent qu'à eux. En effet, à la différence des autres fichiers, les fichiers texte contiennent des caractères du type "retour chariot" ou "Fin de ligne". Dans la suite, la séquence de fin de ligne sera notée **CR-LF**, qui correspond à **C**arriage **R**eturn et **L**ine **F**eed. La fin de texte sera notée **FT** (**F**in de **T**exte). Elle est codée **Ctrl-Z**.

Une variable de type fichier texte est déclarée par le mot réservé **Text**. Comme pour les autres fichiers, il faut assigner (grâce à **Assign**) le nom externe du fichier à un nom logique qui est une variable de type **Text**. De même, un fichier texte ne pourra être ouvert que par **Reset** ou **ReWrite**.

Le traitement des caractères s'effectue par les procédures **Read** et **Write**. D'autres procédures et fonctions permettent le traitement direct de lignes :

Fonctions

EoIn

EoIn (`Nom_Logique`) ;
Retourne **True** si l'on se trouve sur le caractère CR de fin de ligne et **False** dans le cas contraire.

SeekEof

SeekEof (`Nom_Logique`);
Fonction identique à **Eof** mais qui supprime les espaces et les caractères de tabulation avant d'effectuer le test. **SeekEof** est à **Eof** ce que **SeekEoln** est a **Eoln**.

SeekEoln

SeekEoln (`Nom_logique`) ;
Fonction identique à **Eoln** mais supprimant les espaces et les caractères de tabulation avant d'effectuer le test.

Procédures

ReadLn

ReadLn (`Nom_Logique`) ;
Cette procédure permet de sauter par dessus tous les caractères jusqu'à la prochaine ligne. Elle place le pointeur de fichier sur le début de cette ligne, c'est-à-dire juste après la première marque CR-LF rencontrée.

WriteLn

WriteLn (`Nom_Logique`) ;
Introduit dans le fichier texte assigné à `Nom_Logique` une séquence CR-LF pour marquer la fin de ligne.

LES FICHIERS SANS TYPE

Jusqu'à présent, lorsque nous avons déclaré des fichiers, nous avons toujours spécifié le type des éléments susceptibles de s'y trouver. Il existe en Turbo-Pascal la possibilité de ne pas spécifier de type. Ces fichiers peuvent contenir n'importe quel type de donnée. Le travail du programmeur est de gérer la manière dont il organise ses données dans un fichier sans type. La différence avec un fichier typé est que dans le fichier sans type, il n'y a pas de buffer de créé dans la mémoire. Les données sont directement transférées entre le disque et la variable par bloc de 128 octets.

Une économie de place en mémoire centrale est donc réalisée par l'absence de buffer. De plus, un fichier sans type est compatible avec tout autre type de fichier.

Un fichier sans type est déclaré par le mot réservé **File**. Par exemple :

```
Var Bazar : File;
```

Toutes les procédures et fonctions que nous avons vues pour les autres fichiers leur sont applicables, excepté **Read**, **Write** et **Flush**. Celle-ci n'a pas d'utilité dans le cas d'un fichier sans type puisque celui-ci se caractérise par l'absence de buffer et que **Flush** sert précisément à vider le buffer. Pour les deux autres, elles sont remplacées par les procédures **BlockRead** et **BlockWrite** que nous détaillons ici :

BlockRead

BlockRead(`Nom_Logique,Variable,Nbr,[Total]);`
Cette procédure lit `Nbr*128` octets dans le fichier `Nom_Logique` et les met dans `Variable`. Le paramètre `Total` est facultatif. Dans le cas où l'on le spécifie, il retourne le nombre d'enregistrements de 128 octets effectivement transférés. Le programmeur doit s'assurer que `Variable` est de taille suffisante pour accueillir la quantité d'informations réclamée.

BlockWrite

BlockWrite(`Nom_Logique,Variable,Nbr,[Total]);`
Cette procédure est à **BlockRead** ce que **Write** est à **Read**. Elle permet d'écrire dans le fichier associé à `Nom_Logique` les `128*Nbr` premiers octets de`Variable`. Le paramètre `Total` est facultatif et joue le même rôle que dans **BlockRead**.

Un fichier sans type doit, bien sûr, être associé à un fichier physique par **Assign**, être ouvert par **Reset** ou **ReWrite**. On peut en tester la taille par **FileSize** et en tester la fin par **Eof**. Excepté pour la lecture, l'écriture et le vidage du buffer, les choses se passent exactement de la même manière qu'avec un fichier classique.

Les fichiers sans type sont très utiles pour manipuler les fichiers dont on ne connait pas le type. Par exemple, pour dupliquer un fichier, on peut le lire et le réécrire à l'aide d'un fichier sans type. Les données pourront être ainsi dupliquées rapidement et sans qu'on se soucie de leur type.

LES ORGANES LOGIQUES

Comme nous en parlions au début de ce chapitre, un fichier est un lien de communication entre l'ordinateur et un organe extérieur. A ce titre, il existe d'autres fichiers que ceux que l'utilisateur peut créer sur disquette. Le clavier, l'imprimante ou

l'écran sont en effet considérés comme des fichiers par Turbo-Pascal. Voici le détail des organes logiques utilisables :

- **KBD** : c'est le clavier. Il n'est utilisable qu'en entrée. Il n'y a pas d'echo à l'écran quand on le lit.

- **LST** : c'est l'organe de listing, généralement l'imprimante. LST n'accepte que les sorties.

- **CON** : c'est la console. La sortie se fait sur l'écran et l'entrée sur le clavier. A la différence de TRM, cet organe utilise un buffer d'entrée. L'utilisateur aura donc des possibilités d'édition pendant l'entrée (**Read** ou **ReadLn**).

- **TRM** : c'est le terminal. Identique à CON mais sans buffer.

- **AUX** : c'est l'organe auxiliaire, c'est à dire RDR: et PUN:.

- **USR** : c'est l'organe utilisateur. Entrée et sortie se font par les routines de l'utilisateur.

Ces organes logiques sont considérés comme s'ils étaient des fichiers disque et fonctionnent comme eux à l'exception de :

- **ReWrite** et **Reset** sont équivalents.

- **Close** n'a pas d'effet.

- **Erase** provoque une erreur à l'exécution.

- **Eoln** et **Eof** opèrent sur le dernier caractère au lieu d'opérer sur le caractère suivant.

- **ReadLn** lit jusqu'au premier retour chariot, ce dernier inclus.

Les fichiers standard

Turbo-Pascal possède certains fichiers texte prédéclarés et affectés aux organes logiques dont voici la liste :

- **Input** : fichier d'entrée principal affecté soit à CON: soit à TRM:. C'est le fichier d'entrée par défaut. Cela explique pourquoi **Read** est accepté sans paramètre. On pourrait très bien écrire **Read(**`Input,Rep`**)** à la place de **Read(**`Rep`**)** où `Rep` peut être de tout type scalaire simple.

- **Output** : fichier de sortie principal affecté soit à CON: soit à TRM:. C'est le fichier de sortie par défaut. Cela explique pourquoi **Write** est accepté sans paramètre. On pourrait très bien écrire **Write(**`Output, Rep`**)** à la place de **Write(**`Rep`**)** où `Rep` peut être de tout type scalaire simple.

- **Con** : fichier affecté à CON:.

- **Trm** : fichier affecté à TRM:.

- **Kbd** : fichier affecté à KBD:. C'est avec Kbd que vous lirez le clavier lorsque vous ne voulez pas d'écho à l'écran.

- **Lst** : fichier affecté à LST:.

- **Aux** : fichier affecté à AUX:.

- **Usr** : fichier affecté à USR:.

L'emploi de **Assign**, **ReWrite**, **Reset** et **Close** est illégal sur ces fichiers.

LA VERIFICATION DES ENTREES/SORTIES

Les erreurs d'entrée/sortie sont très courantes dans les programmes. Qui n'a pas au moins une fois tapé un nom erroné pour accéder à un fichier ou tapé une lettre lorsque le programme réclame un chiffre ? Pour résoudre ce problème, Turbo-Pascal offre une fonction, **IOResult** qui, associée à la directive de com- apparaître au sein d'une application. Par exemple, vous devez saisir au clavier un paramètre de type entier :

```
WriteLn('Introduisez un entier');
ReadLn(Nombre);
```

où `Nombre` représente une variable de type entier. Mais s'il vous arrive d'entrer autre chose qu'un entier, le programme s'arrêtera et vous enverra un message d'erreur. Une des solutions (certainement la plus simple) est de contrôler la validité du paramètre introduit par **IOResult**. Cette fonction doit être employée conjointement à la directive de compilation **{$I-}** qui évite les arrêts sur erreurs d'entrée/sortie. Dans le cas ou une telle erreur surviendrait, toutes les opérations d'entrée/sortie seraient suspendues jusqu'à l'appel de la fonction **IOResult**. Cette dernière retourne le numéro de l'erreur qui est survenue; si tout s'est passé correctement, elle retourne zéro. Après consultation, **IOResult** est automatiquement remise à zéro et les

opérations d'entrée/sortie sont à nouveau autorisées. C'est au programmeur de corriger les erreurs avec ces indications. Votre lecture d'entier se fera donc comme suit :

```
Repeat
   WriteLn('Introduisez un entier');
   {$I-}
   ReadLn(Nombre);
   {$I+}
Until IOResult=0;
```

Cette méthode est générale et permet de contrôler toute la communication de la machine avec l'extérieur. Elle peut par exemple servir à vérifier si un fichier existe avant de l'ouvrir. Voici la méthode :

```
WriteLn('Introduisez le nom du fichier a tester');
ReadLn(Nom);
Assign(Fichier,Nom);
{$I-}
ReSet(Fichier);
{$I+}
If IOResult <> 0 Then WriteLn('Le fichier n''exite pas');
```

Elle peut également servir à tester si une opération d'écriture sur le disque s'est bien déroulée (l'écriture dans le cas où la disquette est absente ou lorsqu'elle est protégée en écriture est impossible) :

```
{$I-}
Write(Fichier,Donnee);
{$I+}
Erreur:=IOResult;
Case Erreur Of
        00 : WriteLn('Deroulement correct');
        01 : WriteLn('Le fichier n''existe pas');

        ...

        $F2 : WriteLn('Depassement de capacite');
End;
```

Bien sûr, nous ne détaillons pas ici tous les cas d'erreurs possibles, mais il vous suffit de les reprendre dans l'annexe 4 qui les indique tous.

Un programme bien construit ne doit pas se "planter" sur des erreurs d'entrée/sortie. Vous serez donc amené à vous servir très souvent de cette fonction et de la directive de compilation qui lui est associée.

READ ET WRITE

Nous avons déjà utilisé à de nombreuses reprises les procédures **Read** et **Write**. Cependant, elles ne se comportent pas exactement de la même manière avec toutes les sortes de fichiers. Ce paragraphe servira donc à préciser ces spécificités.

Read et ReadLn

Deux syntaxes possibles :

- **Read**(`Var1,Var2,....,Varn`) : dans ce cas, la lecture s'effectue depuis le fichier standard `Input` ;
- **Read**(`Nom_logique,Var1,Var2,....,Varn`) : dans ce cas, la lecture s'effectue depuis le fichier `Nom_Logique`.

Avec une variable `Char`, **Read** lit un caractère dans le fichier.

- Cas d'un fichier disque :
  - **Eoln** est vrai si le caractère suivant est un retour chariot ou un **Ctrl-Z** ;
  - **Eof** est vrai si le caractère suivant est un **Ctrl-Z** ou une fin physique de fichier.

- Cas d'un organe logique :
  - **Eoln** est vrai si le caractère lu est un retour chariot ou un **Ctrl-Z** ;
  - **Eof** est vrai si le caractère lu est un **Ctrl-Z**.

Avec une variable chaîne , **Read** lit autant de caractères que la taille de la variable l'autorise. Certaines commandes d'édition sont permises :

- **<DEL>** a sa fonction habituelle ;
- **Ctrl-X** recule au début de la ligne et efface tous les caractères entrés ;
- **<RETURN>** ou **Ctrl-M** termine l'entrée.

Write et WriteLn

La syntaxe est identique à celle de **Read** et de **ReadLn**. La sortie se fait soit sur le fichier standard `Output`, soit sur un fichier défini par l'utilisateur.

Pour la sortie sur l'écran, il existe certains paramètres de formatage dont voici le détail :

- **Write**(`Caract,n`), où `Caract` est de type **char** et `n` une expression entière, sort `Caract` justifié à droite sur n caractères ;

- **Write**(`Chaine,n`), où `Chaine` est de type **string** [] et `n` une expression entière, sort `Chaine` justifiée à droite et sur `n` caractères ;

- **Write**(`Bool`), où `Bool` est une expression booléenne, sort **True** si `Bool` est vraie et **False** sinon ;

- **Write** (`Reel:n`), où `Reel` est une expression réelle et `n` une expression entière, sort `Reel` sur `n` caractères, justifié à droite, avec un seul chiffre avant la virgule et la notation standard virgule flottante ;

- **Write**(`Reel:n:m`), où `Reel` est une expression réelle et `n` et `m` deux expressions entières, sort `Reel` justifié à droite sur `n` caractères avec la notation standard virgule flottante et `m` chiffres après la virgule.

CHAPITRE 14

LES POINTEURS

Toutes les variables que nous avons vues jusqu'à présent ont une caractéristique commune : elles sont dites "statiques". En effet, leur taille et leur durée de vie sont prédéterminées par le bloc dans lequel elles sont déclarées et référencées par un identificateur. A ces variables statiques s'opposent les variables dynamiques. Celles-ci possèdent les caractéristiques suivantes :

- elles ne sont pas explicitement déclarées comme les variables statiques et ne peuvent pas être référencées par un identificateur ;

- elles sont créées de manière dynamique au fur et à mesure des besoins dans le programme ;

- elles sont référencées grâce à une variable de type pointeur qui est en fait l'adresse de la variable dans la mémoire de l'ordinateur ;

- leur durée de vie est indépendante de celui du bloc dans lequel elles ont été créées.

LE TYPE POINTEUR

La notion de pointeur est assez délicate : il faut distinguer le type du pointeur, la variable de type pointeur, et la variable dynamique que celle-ci référence.

Un pointeur est un type qui a un type : c'est une indirection.

Le type d'un pointeur peut être quelconque, scalaire, tableau, enregistrement, etc.

Un type permet de déclarer des variables. Nous aurons donc des variables de type pointeur. Ces variables contiendront en fait l'adresse mémoire des variables dynamiques qu'elles référenceront, et qui seront du type du pointeur.

Une variable de type pointeur se déclare grâce au symbole " ^ "(lisez pointeur) suivi du type de la variable dynamique que l'on désire créer. Par exemple voici une déclaration de pointeur d'un nombre complexe :

```
Type    Complexe    = Record
                        Partie_Reelle,Partie_Imag : Real;
                      End;
        ComplexePrt = ^Complexe;
Var     Nbr1,Nbr2   : ComplexePrt;
```

Dans cet exemple, le type `Complexe` est un enregistrement contenant deux champs. Les variables `Nbr1` et `Nbr2` sont donc du type pointeur d'enregistrement `ComplexePtr`, et pourront faire référence à des variables dynamiques du type enregistrement.

Les pointeurs sont les seuls éléments du langage à pouvoir être déclarés avant leur type. Il aurait été légal d'écrire :

```
Type    ComplexePrt = ^Complexe;
        Complexe    = Record
                        Partie_Reelle,Partie_Imag : Real;
                      End;

Var     Nbr1,Nbr2   : ComplexePrt;
```

Cependant, la déclaration d'une variable de type pointeur ne crée pas vraiment de variable au moment de la compilation : les variables réellement utilisables seront créées dynamiquement dans le courant du programme. Avant de les utiliser, il faut leur réserver une place en mémoire (dans une structure particulière qui ressemble à une pile,et qu'on appelle le **Heap**; nous étudierons cette structure plus à fond au chapitre 16).

L'allocation de place pour une nouvelle variable dynamique se fera par la procédure **New(**`Ptr`**)**. Dans notre exemple, on écrira :

```
New(Nbr1);
New(Nbr2);
```

Les espaces correspondant à chacun des deux enregistrements seront alors réservés dans le **Heap**, et `Nbr1` et `Nbr2` contiendront leur adresse.

Pour référencer la nouvelle variable, donc celle pointée par la variable pointeur, on se sert également du signe " ^ ", mais cette fois placé après le nom du pointeur. La variable `Pointeur^` est donc du type associé au pointeur (enregistrements dans notre exemple).

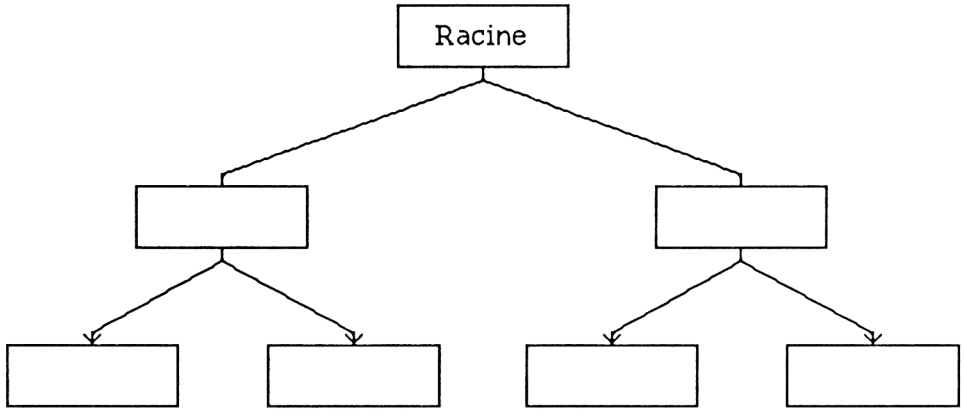
On pourra alors faire, par exemple, les affectations suivantes :

```
Nbr1^.Partie_Reelle := 17.3;
Nbr1^.Partie_Imag := 0.89E-7;
Nbr2^.Partie_Reelle := 0;
Nbr2^.Partie_Imag := -2*Pi;
```

Si on refait **New(**`Nbr1`**)**, on crée une nouvelle variable dynamique (toujours accessible par `Nbr1^`) , mais on perd l'accès à la précédente. C'est pourquoi cette manipulation est assez stupide.

Cependant, les pointeurs deviennent très intéressants dès que chaque donnée (ou ensemble de données) possède (via un pointeur) l'accès à la donnée suivante. On peut par cette méthode construire des listes chaînées, des arbres, des anneaux, etc. Voici par exemple comment on peut construire un arbre binaire :

Un arbre binaire est une structure dans laquelle chaque élément est issu d'un seul élément appelé père et donne naissance à deux (d'où l'adjectif binaire) éléments appelés fils. Le premier élément s'appelle la racine de l'arbre (il n'a pas de père). Les éléments terminaux de l'arbre s'appellent les feuilles (ils n'ont pas de fils). On peut représenter cette structure graphiquement de la manière suivante :



Chaque élément a deux fils et un père. La racine n'a pas de père et les feuilles n'ont pas de fils. La structure de données correspondante peut être définie comme suit :

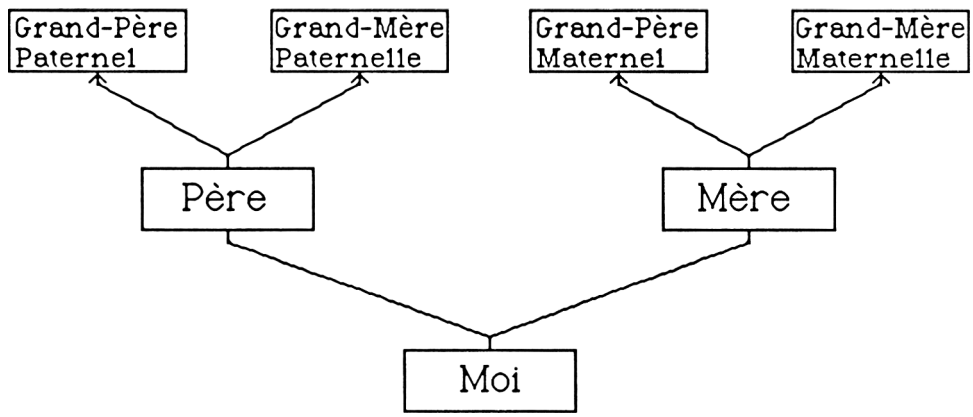
```
Type    Pointeur =  ^Element;
        Element  =  Record
                        Nom : String[20];
                        Gau : Pointeur;
                        Dte : Pointeur;
                    End;
Var     Arbre    :  Pointeur;
```

On s'aperçoit de nouveau ici que la déclaration du type pointeur utilise un type non encore défini qui est `Element`. On peut dire que la définition des deux types est récursive croisée : le type `Pointeur` utilise le type `Element` et le type `Element` contient des champs de type `Pointeur`.

L'arbre généalogique de quelqu'un (qui en est alors la racine) peut être représenté comme ceci :

```
Type    Parent      =   ^Personne;
        Personne    =   Record
                        Nom : String[20];
                        Pere: Parent;
                        Mere: Parent;
                    End;
Var     Arbre   :   Parent;
```

En effet, il correspond au dessin :



Bien sûr, la structure ne limite pas à trois générations puisque les grands-parents possèdent également deux pointeurs pointant respectivement vers leur père et leur mère. La structure ainsi définie peut se répéter indéfiniment (ou presque) et permet de remonter jusqu'au plus profond de l'arbre généalogique. Lorsqu'on ne sait plus qui sont les parents d'un ancêtre, on fait pointer les champs père et mère de celui-ci sur la constante prédéfinie **Nil**. Cette valeur particulière indique que l'on se trouve au niveau des feuilles de l'arbre qui n'ont pas de fils. Le pointeur doit alors pointer sur "rien". Le rien en question est représenté par la constante prédéfinie **Nil**.

L'arbre binaire est une des nombreuses possibilités des arbres. On peut également faire des arbres ternaires (ou plus) où chaque élément possède trois fils. La déclaration peut alors se faire comme suit :

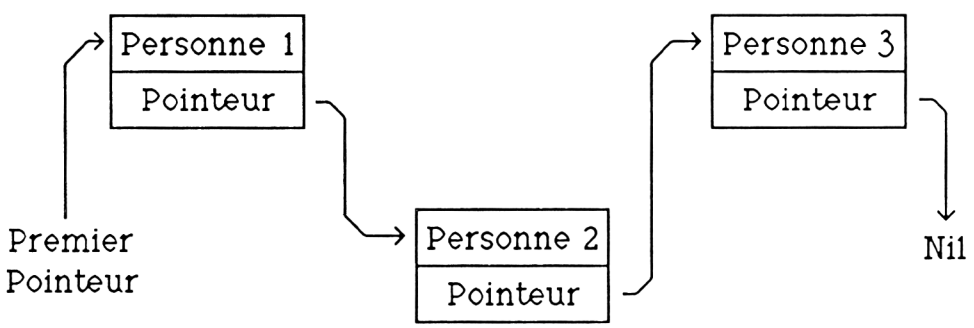
```
Type    Pointeur    =   ^Element;
        Element     =   Record
                            Nom :   String[20];
                            Gau :   Pointeur;
                            Mil :   Pointeur;
                            Dte :   Pointeur;
                        End;
Var     P1 : Pointeur;
```

Dans cette structure, chaque élément peut pointer sur trois fils différents.

Les arbres offrent une représentation de données élégante pour de nombreux problèmes. L'arbre généalogique en est une utilisation possible, mais cette structure de données est surtout utilisée conjointement à une recherche arborescente. Celle-ci se sert d'un arbre par nécessité ; prenons l'exemple d'un jeu tel que les échecs :

Supposons qu'au début de la partie, il y ait N coups possibles pour les blancs. Pour analyser chacun de ces coups un à un, on représente chaque coup par une branche partant de la racine (qui représente l'état initial du jeu). La racine possède donc N fils. A chacun de ces coups correspondent certaines réponses envisageables par l'adversaire. Soit P_1, P_2 ..., P_N les nombres de ces coups possibles. On se rend bien compte ici que cette structure va croître au fur et à mesure des observations et que la meilleure représentation du problème réside dans une structure d'arbre où pour chaque coup, les fils représentent les coups possibles pour l'adversaire au coup d'après. C'est ce qu'on appelle l'arbre d'un jeu. Les jeux d'échecs, de dames, de morpions, etc. peuvent tous être représentés par ce genre de structure. Celle-ci sert en effet par la suite à appliquer des algorithmes permettant de trouver le meilleur (ou plutôt le moins mauvais) coup à jouer à un moment donné de la partie (tout particulièrement si l'on utilise l'algorithme Alpha-Bêta).

Les pointeurs servent également à établir des stuctures de liste chaînée. Dans celles-ci, chaque élément pointe sur le suivant jusqu'au dernier qui pointe sur **Nil**. La représentation graphique peut être dessinée ainsi :



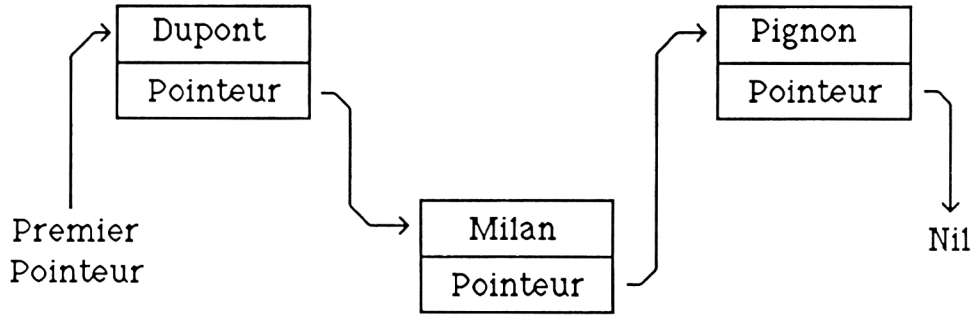
Cette structure assez simple peut être compliquée à loisir pour de nombreux problèmes du même type. Ici, nous avons un premier pointeur nous indiquant où se trouve la première fiche. Ce pointeur ne doit bien sûr pas être détruit si l'on veut conserver l'accès à la structure. Dans la première fiche se trouvent les renseignements qu'on a fournis, plus un pointeur permettant l'accès à la deuxième fiche. Ceci se répète jusqu'à la dernière fiche dont le pointeur pointe sur **Nil**. Au niveau programmation, la stucture ci-dessus peut se déclarer comme suit (nous supposerons que chaque fiche ne contient qu'un nom et un pointeur, mais la structure peut être étendue à volonté) :

```
Type    Pointeur    =   ^Fiche;
        Fiche       =   Record
                            Nom :    String[20];
                            Suivant :    Pointeur;
                        End;
Var     Premier     :   Pointeur;
```

Dans cette déclaration, le premier pointeur de fiche sera conservé dans la variable `Premier`.

Cette structure est particulièrement utile lorsqu'on veut ajouter ou supprimer des données. Il est en effet difficile de supprimer au beau milieu d'un tableau les données qu'on y a enregistrées. Cela devient très simple si on utilise des pointeurs : il suffit de trouver l'endroit où l'on veut supprimer la donnée et de modifier en conséquence les valeurs des pointeurs. Prenons un exemple :

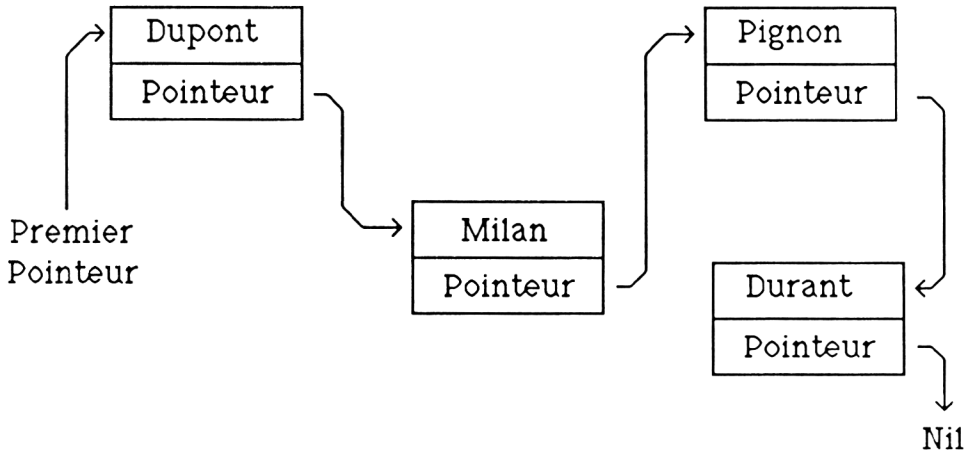
Vous désirez organiser votre carnet d'adresses à la manière d'une liste chaînée. Vous possédez à un instant donné le carnet suivant :



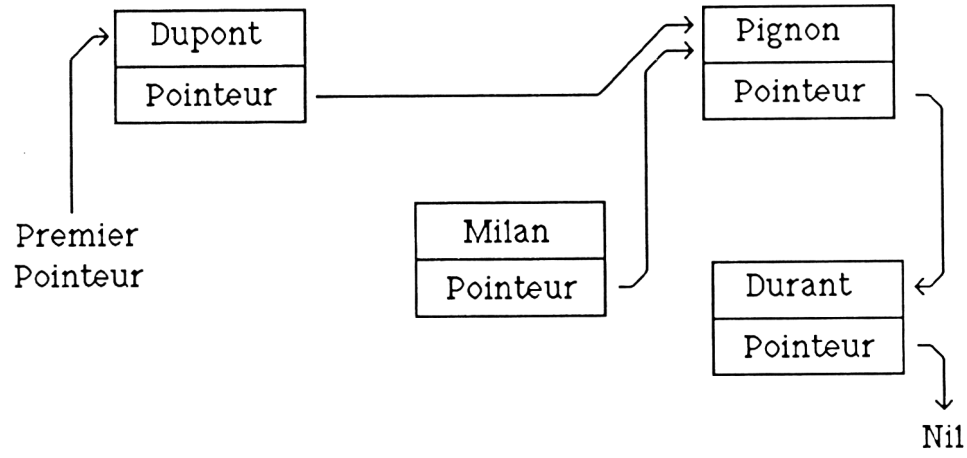
Vous voulez ajouter alors un certain monsieur Durant. Celui-ci doit se trouver après Pignon (en bout de liste). Il faut alors procéder comme suit :

- rechercher le dernier élément de la liste;
- faire pointer cet élément sur Durant;
- faire pointer Durant sur **Nil**.

Cela donne alors la structure suivante :



Cette manipulation est relativement simple. Mais la simplicité du procédé apparaît encore davantage lorsqu'il s'agit de supprimer un élément de la liste : il suffit de mettre à jour un pointeur. Pour supprimer la fiche de Milan, il suffit de récupérer son pointeur et de l'affecter au pointeur qui pointait jusqu'à présent sur lui. On obtient alors la structure :



Le pointeur de Milan pointe toujours sur Pignon, mais cela est sans importance puisqu'on ne peut plus accéder à Milan. La place inutilement occupée par Milan dans la mémoire peut, si on le désire, être récupérée par **Dispose** (voir la fin de ce chapitre), donc par l'emploi d'une seule fonction. Pour récupérer la place au sein d'un tableau, il faudrait déplacer tous les éléments suivant l'élément supprimé ce qui pourrait être très long dans le cas d'une liste importante.

Certes ces dessins ne créent pas de programme, mais ils expliquent de manière visuelle ce qui ce passe réellement dans l'ordinateur.

Voici à présent un programme qui permet de créer notre liste chaînée, de la lire entièrement, d'y inclure un élément en fin de liste et d'en supprimer un élément.

```
Program Liste_Chainee;
Type  Pointeur   =  ^Fiche;
          Fiche  =  Record
                        Nom : String[20];
                        Suivant :Pointeur;
                    End;
Var       Premier :Pointeur;

Procedure Ajouter;
Var P,Pt   : Pointeur;
Begin
      New(P); {Cree un nouvel enregistrement}
      If Premier=Nil    Then Premier=P
                        Else      Begin
                                  Pt:=Premier;
                                  While Pt^.Suivant<>Nil Do
                                  Pt:=Pt^.Suivant;
                                  Pt^.Suivant:=P
                                  End;
      Repeat
          ClrScr;
          WriteLn('Entrez le nom a ajouter');
          ReadLn(P^.Nom);
      Until P^.Nom<>'';
      P^.Suivant:=Nil {Fait pointer le dernier element sur NIL}
End;
Procedure Supprimer;    {Cette procedure recherche dans la
                         liste}
Var   Nom : String[20];{le nom a supprimer.Quand elle l_a
                        trouve}
      Pt  : Pointeur;   {elle retablit les pointeurs}
Begin
      ClrScr;
      WriteLn('Entrez le nom a supprimer');
      ReadLn(Nom);
      If Premier<>Nil
          Then If Premier^.Nom=Nom
                 Then Premier:=Premier^.Suivant
                 Else Begin
                        Pt:=Premier;
                        While Pt^.Suivant<>Nil Do
                            If Pt^.Suivant^.Nom=Nom
                              Then
                            Pt^.Suivant:=Pt^.Suivant^.Suivant
                              Else Pt:=Pt^.Suivant
                    End
End;
```

```
Procedure Visualiser;   {Cette procedure parcoure la
                        liste}
Var Pt     : Pointeur;  {et affiche les noms au fur et a
                        mesure}
Begin
        ClrScr;
        Pt:=Premier;
        While Pt<>Nil Do Begin
                              Write(Pt^.Nom,' ---> ');
                              Pt:=Pt^.Suivant
                         End;
        WriteLn('Fin de liste');
        WriteLn;
        WriteLn('Appuyez sur une touche pour continuer');
        Repeat Until KeyPressed
  End;
  Procedure Choisir;      {Cette procedure gere le menu
                          principal}
Var Reponse : Char;
  Begin
        Repeat
            ClrScr;
            WriteLn('Choisissez une option');
            WriteLn('**********************');
            WriteLn;
            WriteLn('A ---> Ajouter un element');
            WriteLn('S ---> Supprimer un element');
            WriteLn('V ---> Visualiser la liste');
            WriteLn('T ---> Terminer');
            Repeat
                Read(Kbd,Reponse);
            Until UpCase(Reponse) In ['A','S','V','T'];
            Case UpCase(Reponse) Of
                   'A'    : Ajouter;
                   'S'    : Supprimer;
                   'V'    : Visualiser;
                   'T'    : ClrScr
            End;
        Until UpCase(Reponse) = 'T'
  End;

  Begin{Debut du programme principal}
        Premier:=Nil;
        Choisir;
        ClrScr
  End.
```

Bien sûr ce programme ne se charge que d'enregistrer des noms, mais il suffit d'ajouter des champs au **Record** pour compléter ce squelette. Plus qu'un

programme complet de gestion de liste chaînée, il constitue une base dont vous pourrez vous servir pour élaborer un programme complet avec une meilleure présentation et de meilleures fonctions.

LES CONSTANTE, FONCTIONS ET PROCEDURES OPERANT SUR LES POINTEURS

Jusqu'à présent, nous n'avons parlé que des procédures et fonctions élémentaires travaillant sur les pointeurs. Nous allons dans cette dernière partie résumer les procédures et fonctions déja vues et en découvrir de nouvelles qui pourront vous être utiles.

Avant de les aborder, il faut rappeler que les variables dynamiques ne sont pas mises n'importe où dans la mémoire de l'ordinateur. En effet, elles sont rangées dans leur ordre d'allocation dans une structure appelée **Heap**. Celle-ci sera vue en détail au chapitre 16. Pour l'instant vous devez juste savoir que le **Heap** fonctionne comme une pile (mais dans l'autre sens) sur laquelle viennent se ranger les variables au fur et à mesure de leur allocation. On peut cependant accéder à n'importe quel élément à tout moment par le biais des pointeurs (alors que dans la pile, on ne peut accéder qu'au dernier élément empilé).

Constante

Nil

Nil;
La valeur **Nil** est compatible avec tous les types de pointeurs. Elle sert à faire pointer un pointeur sur rien (par exemple en bout d'une liste chaînée ou pour les feuilles d'un arbre). **Nil** peut être utilisée dans les tests sur les variables pointeurs.

Fonctions

MaxAvail

MaxAvail;
Retourne un **entier** contenant la taille (en octets) du plus grand espace (trou) disponible dans le **Heap**.

MemAvail

MemAvail;
Retourne un **entier** contenant l'espace disponible dans le **Heap** à un instant donné. Cette valeur permet de savoir si l'on peut ou non demander la création d'une nouvelle variable dynamique. Doit être employée par le programmeur pour que le **Heap** ne deborde pas sur la pile (plus de détails sur la structure Heap/Pile sont donnés dans le chapitre 16).

Ord

Ord(`Ptr`);
Retourne un **entier** contenant l'adresse sur laquelle la variable `Ptr` (qui doit être de type pointeur) pointe. Utilisation déconseillée au débutant.

Ptr

Ptr(`Ent`);
Convertit le paramètre `Ent` (qui doit être de type entier) en un pointeur compatible avec tous les types de pointeurs (opération inverse de **Ord**). Permet d'examiner tout endroit de la mémoire. Egalement déconseillée aux débutants.

Procédures

Dispose

Dispose(`Ptr`);
Permet de récupérer la place pointée par `Ptr` dans le **Heap**. Le fonctionnement est complètement différent de celui de **Release** qui ne doit jamais être utilisée en même temps. En effet, avec **Dispose**, seule la place correspondant au pointeur spécifié est libérée (alors qu'avec **Release**, c'est toute la place à partir et au dessus du pointeur spécifié qui est libérée). L'appel de **Dispose** provoque donc des "trous" dans le **Heap**, mais ces trous seront réutilisés par les prochains appels de **New** si l'espace disponible à l'intérieur de ceux-ci est suffisant.

FreeMem

FreeMem(`Ptr,Ent`);
Effectue l'opération inverse de **GetMem**: Elle permet de récupérer un bloc entier dans le **Heap**. `Ptr` pointe sur le début de la place à libérer et `Ent` est le nombre d'octets à récupérer. Celui-ci doit être exactement le même que celui qui a servi dans le **GetMem** sous peine de résultats indéterminés.

GetMem

GetMem(`Ptr,Ent`);
Cette procédure est un complément à la procédure **New**, elle permet au programmeur d'allouer une place dont il définit la taille (et non pas une place définie par la taille de l'argument comme avec **New**). `Ptr` est une variable de type pointeur qui pointe sur l'espace alloué et `Ent` est une expression entière qui détermine le variable de type pointeur qui pointe sur l'espace alloué et `Ent` est une expression entière qui détermine le nombre d'octets à allouer. Il faut bien sûr que le programmeur s'assure de la disponibilité dans le **Heap** de la place qu'il désire. Cela pourra être fait à l'aide des fonctions **MaxAvail** et **MemAvail**.

Mark

Mark(`Ptr`**);**
Affecte la valeur du pointeur de **Heap** à la variable `Ptr`. Cela permet de mémoriser l'état du **Heap** à un instant donné. Cet état pourra ensuite être restitué par la procédure **Release**.

New

New(`Ptr`**);**
Crée une variable associée au pointeur de variable `Ptr`. Cette variable est rangée dans le **Heap** qui progresse vers la pile de récursion dans la mémoire. Déplace donc vers le haut le pointeur de **Heap**. La variable créée sera référencée par `Ptr^`.

Release

Release(`Ptr`**);**
Affecte la valeur `Ptr` au pointeur de **Heap** (`Ptr` est un pointeur préalablement initialisé par **Mark**). Détruit toutes les variables dynamiques créées depuis l'exécution de **Mark**. Contrairement à **Dispose**, l'emploi de **Release** ne crée pas de trou dans le **Heap**, mais diminue seulement la taille de celui-ci.

CHAPITRE 15

L'INTERFACE TURBO-PASCAL ET EXTENSIONS

Turbo-Pascal est un outil de développement très ouvert. En effet, contrairement à de nombreux compilateurs, celui-ci permet une communication aisée avec l'extérieur. On peut dans un programme écrit en Turbo-Pascal utiliser des routines de CP/M, des sous-programmes externes et écrire directement des parties en langage machine. Nous allons détailler toutes ces possibilités dans le présent chapitre.

UTILISATION DE CP/M

Vous le savez surement, votre Amstrad requiert l'utilisation de CP/M pour pouvoir utiliser le Turbo-Pascal. Si vous possédez un 6128, vous avez d'ailleurs le choix entre CP/M 2.2 et CP/M Plus. Ces deux versions, bien qu'ayant le même cœur, diffèrent, en particulier, par l'endroit où elles se chargent dans votre Amstrad.

Si vous possédez le modèle 464 ou 664, vous emploierez CP/M 2.2 et vous disposerez d'une place de l'ordre de 8 Ko pour le source et l'objet. Ce n'est pas beaucoup, mais cela ne limite en rien la taille de vos programmes; il est en effet possible, par le biais de l'**Inclusion**, de faire référence dans le corps d'une partie, à une autre partie se trouvant sur disquette. Nous verrons ce principe ainsi que celui d'**Overlay** dans le paragraphe qui leur est consacré (Inclusion et Overlay) dans ce même chapitre.

Si vous possédez le modèle 6128, la différence se fait nettement sentir. CP/M 2.2 se charge de la même manière que sur un 464 ou un 664 et ne laisse que 8 Ko pour le source et l'objet. En revanche, si vous utilisez CP/M Plus, le système d'exploitation

est alors chargé dans la deuxième bank de 64 Ko et laisse ainsi 32 Ko libres pour le source et l'objet. Cela évite d'avoir trop souvent recours aux Inclusion et Overlay.

CP/M est un système d'exploitation dans lequel un certain nombre de routines standard sont définies. Si l'on passe d'un ordinateur à l'autre, ces routines auront les mêmes effets, dès l'instant que le système d'exploitation est toujours CP/M. Turbo-Pascal permet d'utiliser directement ces routines à l'intérieur même d'un programme. Les routines utilisables sont les routines standard du Bios et du Bdos de CP/M. Celles-ci ne sont pas données ici car elles pourraient faire l'objet d'un livre à elles seules. Pour vous les procurer, il vous suffit d'avoir une documentation sur le système d'exploitation CP/M. Nous ne donnerons donc ici que la manière d'accéder à ces routines.

Turbo-Pascal possède à cet effet deux procédures et quatre fonctions. Voici leur détail.

Fonctions et procédures s'adressant au Bdos

Bdos

Bdos(`C,DE`**);**
Cette procédure permet d'appeler les fonctions standard du Bdos de CP/M. Le paramètre `C` est un entier qui sera chargé dans le registre C du Z80 pour indiquer à CP/M quel est le numéro de la routine à appeler. Le paramètre `DE` est optionnel. Il indique ce qu'il faut charger dans le double registre DE avant l'appel de la fonction du CP/M. Ce paramètre est optionnel car certaines routines de CP/M ne nécessitent pas de paramètre. Après le chargement du ou des registres par cette procédure, un appel à l'adresse 5 est effectué pour appeler le Bdos.

Bdos

Bdos;
Cette fonction est appelée comme la procédure du même nom mais rend un résultat entier qui correspond à ce que le Bdos retourne dans le registre A du Z80 après avoir exécuté la fonction CP/M demandée par l'utilisateur.

BdosHL

BdosHL(`C,DE`**);**
Cette fonction est identique à Bdos mais elle retourne ce qui se trouve dans le registre double HL du Z80 après l'exécution de la fonction Bdos demandée par l'utilisateur.

Fonctions et procédures s'adressant au Bios

Bios

Bios(`Func,BC`**);**
Cette procédure est utilisée pour appeler les routines du Bios. Le paramètre `Func` est un entier qui indique le

numéro de la routine à appeler. `BC` est un paramètre facultatif qui permet de charger le double registre BC du Z80 avant l'appel. Pour le numéro de `Func`, on a WBOOT qui vaut 0, CONST qui vaut 1 etc.

Bios

Bios;
Cette fonction est appelée comme la procédure du même nom mais rend un résultat entier qui correspond à ce que le Bios retourne dans le registre A du Z80 après avoir exécuté la fonction CP/M demandée par l'utilisateur.

BiosHL

BiosHL(`C,DE`);
Cette fonction est identique à Bios mais elle retourne ce qui se trouve dans le registre double HL du Z80 après l'exécution de la fonction Bios demandée par l'utilisateur.

Fonctions sur la ligne de commande CP/M

Les fichiers .COM créés par le compilateur peuvent être exécutés directement sous CP/M. On peut en outre leur passer des paramètres depuis la ligne de commande CP/M. Vous pourrez voir une application de ces fonctions dans le programme CMDLIN.PAS qui se trouve sur la disquette originale.

Paramcount

Paramcount;
Retourne sous forme d'**entier** le nombre de paramètres passés au programme depuis la ligne de commande. Les espaces et les caractères de tabulation servent de séparateurs.

Paramstr

Paramstr(`n`);
Retourne sous forme de **chaîne** le n^e paramètre passé au programme depuis la ligne de commande.

Chain et Execute

Turbo-Pascal possède deux procédures permettant d'activer d'autres programmes depuis un programme Turbo.

Execute

Execute(`Nom_logique`);
Permet d'exécuter tout fichier .COM (compilé avec l'option **C**). Ce dernier est chargé et exécuté à l'adresse $100. `Nom_Logique` doit avoir été affectée à un fichier disque par la procédure **Assign**. Les fichiers .COM contiennent la librairie Pascal, c'est-à-dire toutes les routines de base qu'un programme Turbo peut utiliser. C'est pourquoi un programme .COM ne fait jamais moins de 8 Ko sur la disquette, même s'il est très court. C'est

aussi pourquoi le fichier Turbo-Pascal n'est pas nécessairement présent sur la disquette d'exécution d'un fichier .COM.

Chain **Chain(**`Nom_Logique`**);**
Permet d'exécuter un fichier .CHN (compilé avec l'option **H**). `Nom_Logique` doit avoir été affectée à un fichier disque par la procédure **Assign**. Elle est chargée et exécutée à l'adresse de départ du programme en cours, c'est-à-dire à l'adresse spécifiée à la compilation de celui-ci ; Elle en utilise en outre la librairie. Ce fichier, ne contenant que le code, est donc plus court.

Pour ces deux procédures, une erreur d'entrée/sortie survient si le fichier appelé n'existe pas sur la disquette. Pour tester sa présence utilisez **IOResult** dont nous avons parlé au chapitre 13.

Ces fonctions et procédures permettent d'avoir une interface directe entre Turbo-Pascal et le système d'exploitation. Cela évite d'avoir à recréer des routines existant déja au sein de CP/M. Ces fonctions seront d'une grande utilité pour le programmeur habitué à CP/M et désirant utiliser ses spécificités à l'intérieur d'un programme écrit avec Turbo-Pascal.

UTILISATION DE SOUS-PROGRAMMES EXTERNES

De même que Turbo-Pascal peut communiquer avec CP/M, il est également capable d'exécuter des sous-programmes dits "externes". Ces sous-programmes sont généralement écrits en langage machine, mais si un jour apparaissent sur l'Amstrad d'autres compilateurs que Turbo-Pascal, ce pourra être du code généré par un de ces compilateurs. Voyons comment l'on communique avec ces sous-programmes.

Lorsque l'on utilise des sous-programmes externes, il faut les déclarer par le mot réservé **External**. On est obligé de déclarer l'adresse où se trouve la partie externe en question. Le mot réservé **External** est donc immédiatement suivi par l'adresse mémoire de démarrage du sous-programme externe. Voici quelques exemples de déclarations :

```
Procedure Temporisation; External $5000;
Function Couleur:Integer; External $6000;
```

Dans le corps du programme, l'appel aux sous-programmes externes est identique à celui des fonctions ou procédures définies dans le programme.

Turbo-Pascal offre également la possibilité de passer des paramètres à ces sous-programmes externes. On peut prendre l'exemple d'une procédure externe écrivant

un caractère à une position donnée de l'écran. Si la routine correspondante était en $4000 (mais en fait, ce n'est pas le cas sur Amstrad), celle-ci pourra être déclarée comme suit :

```
Procedure Ecrire(Car:Char;Lig,Col:Integer);External $4000;
```

Lorsque vous appellerez cette procédure depuis le programme principal, quatre paramètres devront être transmis : le caractère à écrire, la ligne, la colonne et l'adresse de retour. Cette dernière permettra au programme de continuer son exécution juste après l'instruction d'appel de procédure externe. Ces paramètres sont placés sur la pile et occupent une certaine place en fonction de leur type. L'ordre dans lequel sont empilés les paramètres est tel que l'adresse de retour se trouve toujours au sommet de la pile. Le sous-programme externe devra se charger de dépiler cette adresse de retour, de dépiler les paramètres transmis et de remettre l'adresse de retour sur la pile car sinon, le RET final du sous-programme externe ne retournerait pas dans le programme ayant exécuté l'appel. La taille des paramètres transmis est soumise aux règles qui sont exposées dans la partie Paramètres de ce chapitre.

De même qu'on peut transmettre des paramètres à un sous-programme externe, celui-ci peut aussi en retourner. Dans le cas de retour de paramètres par résultat de fonction, le sous-programme externe doit respecter les règles suivantes :

- les valeurs de type scalaire (entiers, booléens, caractères) doivent être dans le registre HL. Si la valeur tient sur un octet, on la mettra dans L et on mettra H à zéro;

- les réels doivent être retournés dans BC, DE et HL avec B, C, D, E, H contenant la mantisse et L contenant l'exposant selon les mêmes règles que celles de la transmission des paramètres (cf partie Paramètres);

- les pointeurs seront retournés dans HL;

- les chaînes de caractères et les ensembles seront retournés dans la pile selon les mêmes conventions que pour la transmission (cf partie Paramètres).

PARAMETRES POUR FONCTIONS ET PROCEDURES EXTERNES

Les paramètres sont transférés aux procédures et aux fonctions grâce à la pile du Z80. Si vos appels se font uniquement à l'intérieur du programme, Turbo-Pascal gère cette pile et l'utilisation de celle-ci est totalement transparente au programmeur. En revanche, si vous utilisez des sous-programmes externes, vous devez alors gérer vous-même cette pile : vous devez récupérer vos paramètres, ne pas oublier de remettre l'adresse de retour au sommet de la pile, et éventuellement ranger d'autres paramètres dans cette pile. Les types des paramètres déterminent la longueur de ceux-ci. Le mode de transmission des paramètres (par variable ou par valeur)

détermine si l'on trouve dans la pile le paramètre effectif ou l'adresse de ce paramètre. Nous allons donc voir dans cette partie comment sont codés ces paramètres et la manière dont ils sont rangés sur la pile.

Paramètre transmis par variable

Dans ce cas, l'adresse absolue en mémoire est rangée sur la pile. Cette adresse occupe 16 bits et indique l'adresse en mémoire du premier octet de la donnée considérée. Vous pouvez, en récupérant cette adresse, accéder de manière indirecte au paramètre effectif.

Paramètre transmis par valeur

Dans ce cas, le paramètre se trouve de manière effective sur la pile. La taille occupée par chaque paramètre dépend de son type. Nous allons voir ici comment sont codés ces paramètres et la taille qu'ils occupent .

- Les entiers, les booléens, les caractères ainsi que les scalaires déclarés sont transférés sur la pile sous un format 16 bits. Si cette longueur convient très bien pour un entier, elle est trop importante pour les variables n'occupant qu'un seul octet. Dans ce cas, l'octet de poids fort est mis à zéro.

- Les réels sont codés sur six octets soit 48 bits. La mantisse occupe 5 octets (soit 40 bits) et l'exposant 8 bits. L'exposant réel est obtenu en soustrayant $80 de l'exposant codé. Ainsi un exposant valant $88 signifiera qu'il faut multiplier la mantisse par 2^8 car $88-$80=8. La mantisse est normalisée, c'est-à-dire qu'elle se présente toujours sous une forme du type 0.1... Le bit juste après la virgule est toujours à 1. Il n'est donc pas nécessaire de le coder et c'est pourquoi le bit lui correspondant dans les cinq octets de la mantisse sert à coder le signe de celle-ci (1 indique que le nombre est négatif et 0 qu'il est positif). Lorsqu'un réel est transmis sur la pile, il occupe 3 mots de 16 bits. Si on dépile par la suite d'instructions :

  ```
  POP HL
  POP DE
  POP BC
  ```

  on récupère l'exposant dans L, l'octet de poids fort dans B et les suivants dans C, D, E, H.

- Les chaînes de caractères sont codées avec un octet par caractère. Lorsqu'on a une chaîne en haut de la pile, le premier octet est la longueur de la chaîne. Les octets suivants sont les caractères de la chaîne avec le premier caractère à l'adresse la plus basse.

- Les pointeurs sont transmis sur la pile en occupant un mot de 16 bits. Si le pointeur vaut **Nil**, on trouve 0 comme valeur.

- Les tableaux et enregistrements, même transmis par valeur, ne sont pas recopiés dans la pile. C'est en fait l'adresse où est rangé en mémoire le tableau ou l'enregistrement qui se trouve dans la pile. C'est au programme de recopier, grâce à cette adresse, le contenu du tableau ou de l'enregistrement.

- Les ensembles sont transmis de manière assez particulière : un ensemble occupe toujours 32 octets. La routine suivante permet de dépiler l'ensemble et de le mettre dans buffer :

```
LD      DE,Buffer
LD      HL,0
ADD     HL,SP
LD      BC,32
LDIR
LD      SP,HL
```

  L'octet de plus faible poids de l'ensemble sera rangé à l'adresse la plus basse de buffer. Voir la partie Le format interne des données du chapitre 16 pour voir ce que signifient ces 32 octets.

COMMANDE INLINE

Le Turbo-Pascal offre au programmeur la possibilité d'inclure au sein d'un programme source écrit en Pascal une partie directement écrite en code machine. Pour cela, il faut employer l'instruction **InLine** qui indique au compilateur que les codes qui vont suivre sont des codes machine. Ces codes doivent être entourés par des parenthèses et séparés par des "slash" (/). Par exemple :

```
InLine ($06/$10/$00/$10/$FD/$C9);
```

génère les codes hexas spécifiés qui correspondent au programme Assembleur :

```
        LD      B,#$10
Bou:    NOP
        Djnz    Bou
        RET
```

qui ne fait qu'exécuter 16 boucles de NOP (c'est une boucle de temporisation).

Cela est très pratique car les routines que l'on désire construire en langage machine n'ont pas besoin d'être extérieures au programme (avec toutes les contraintes que cela implique). Chaque élément fourni génère un octet ou un mot. Par exemple, on aurait pu écrire le programme précédent comme :

```
InLine ($0610/$0010/$FDC9);
```

De même, on peut introduire du code par le biais d'identificateurs de variables, de

constantes entières, de fonctions, de procédures ou de références à un compteur d'emplacement qui sera alors indiqué par un astérisque (*). On peut également faire des opération sur les identificateurs (+ et -). Le code sera généré de la manière suivante :

- pour une constante, c'est la valeur de la constante qui est prise en compte;

- pour un identificateur de variable, c'est l'adresse en mémoire de la variable qui est prise;

- pour les procédures et les fonctions, c'est l'adresse de début en mémoire du code de la procédure ou de la fonction qui est prise;

- pour un compteur d'emplacement, c'est l'adresse du compteur qui est prise, c'est-à-dire l'adresse du prochain code généré en mémoire.

Vous pouvez appliquez sur tous ces éléments les opérateurs + et - pour les modifier.

Les codes fournis seront toujours sur 16 bits dans les cas des indentificateurs de variables, de procédures ou de fonctions et des compteurs d'emplacement. S'il s'agit d'une constante entière, ou si la valeur est comprise dans l'intervalle [0..255], le codage s'effectue alors sur 8 bits. Les caractères "<" et ">" peuvent être utilisés pour modifier la règle énoncée ci-dessus. Le caractère "<" limite la valeur codée à l'octet de poids faible, même s'il s'agit d'une donnée sur 16 bits. De même, le symbole ">" permet de coder une donnée sur 16 bits même si celle-ci est définie sur l'intervalle [0..255] (l'octet de poids fort est à 0). Voici quelques exemples pour illustrer toutes ces règles :

```
<$5467 sera codé      $67
>$12   sera codé      $12,$00
```

si `val` est une constante entière valant 17 on aura :

```
Val+7  sera codé      $18
```

Tous ces éléments vont vous permettre d'insérer de petites (ou de grosses) parties en langage machine directement dans le source de votre programme écrit en Turbo-Pascal. A cet effet, l'annexe 5 fournit tous les codes machine des instructions Assembleur du Z80.

INCLUSION ET OVERLAY

La place laissée en mémoire au programmeur pour le source n'est pas énorme. Si vous avez un 6128 et que vous utilisez CP/M Plus, vous disposez de 32 Ko, mais si vous avez un 464 ou un 664, il ne reste que 8 Ko. Cela est bien peu, surtout si vous

voulez construire un programme important. Certaines instructions du compilateur Turbo-Pascal permettent d'étendre à loisir la taille du programme source et du code objet. Grâce à elles, vous n'aurez aucun mal à compiler un source de 100 Ko ou à faire tourner un objet de 80 Ko.

L'Inclusion

Cette technique permet d'obtenir un source beaucoup plus grand que la place maximale disponible en mémoire pour le source. Elle consiste à fragmenter le programme en plusieurs parties qui seront enregistrées séparément sur la disquette dans différents fichiers. Pour inclure une partie de programme dans la partie résidant en mémoire, il suffit de donner le nom du fichier à l'endroit où l'on veut l'inclure dans le programme, de la manière suivante :

```
{$I Nom_de_Fichier }
```

C'est au moment de la compilation que le compilateur ira chercher sur le disque le fichier à inclure.

Vous devez faire attention à la syntaxe : en particulier, si vous ne spécifiez pas les trois lettres de l'extension de votre fichier à inclure, vous devez laisser un espace avant la parenthèse fermante (}). Dans le cas où l'extension n'est pas précisée, le compilateur prendra **.PAS** par défaut. Une directive d'inclusion devra être la seule sur une même ligne de programme.

Le nombre des fichiers que l'on peut inclure dans un programme n'est pas limité. Cependant un fichier que l'on inclut ne peut pas contenir lui-même une autre inclusion.

Cela permet de simuler un source de la longueur que l'on désire. De plus, cette technique permet d'écrire des librairies dans lesquelles sont stockées les procédures et les fonctions dont on a souvent besoin. Elle peut être associée à celle de la compilation sur disque qui évite de mobiliser de la place en mémoire pour le stockage du code objet. La compilation sur disque se fait grâce au choix de **O**ption puis de **C**om ou de c**H**n (cf chapitre 2).

L'Overlay

Cette technique sert à faire tourner en mémoire des codes objets beaucoup plus longs que la mémoire de l'ordinateur ne le permet. Elle consiste à n'appeler en mémoire à un instant donné que les parties dont on a besoin. Elle entraîne un ralentissement de l'exécution à cause des accès disque.

Dans le cas d'une structure d'Overlay, un certain nombre de procédures et de fonctions est déclaré comme absent au moment du chargement du programme.

C'est le programme lui-même qui chargera les parties nécessaires à l'exécution au moment de leur appel. Pour ce faire, il réservera dans la mémoire une place qui correspond à la place prise par la plus grande des parties déclarées en Overlay. Ainsi, le programme pourra rappeler dans cet espace n'importe quelle partie déclarée en Overlay. Cependant, une partie déclarée en Overlay ne pourra pas faire référence à une autre partie Overlay du même fichier par manque de place.

La création de sous-programmes Overlay est très simple : il suffit de rajouter le mot réservé **Overlay** devant sa déclaration. Par exemple comme ceci :

```
Overlay Procedure Premiere;
Begin
  ...
End;

Overlay Function Deuxieme;
Begin
  ...
End;
```

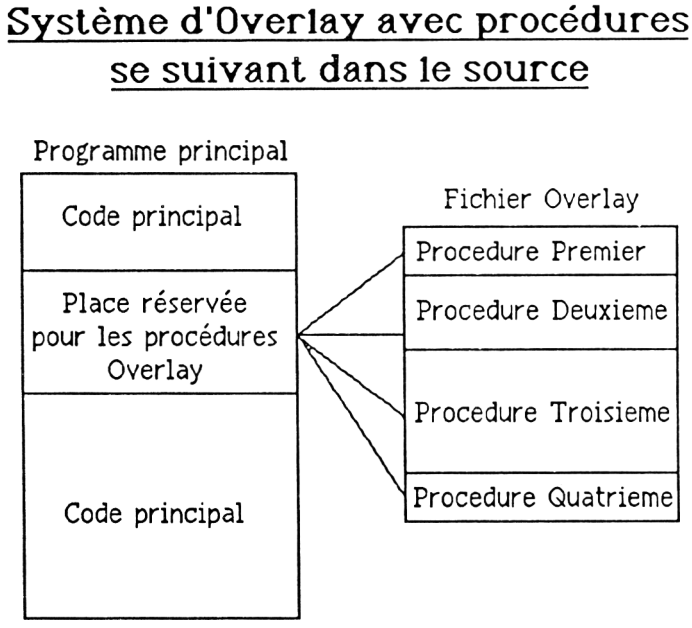
Les procédures et les fonctions ainsi déclarées ne seront présentes en mémoire que lorsqu'on les appelera. Lorsque le compilateur rencontrera ces Overlay, il créera des fichiers de même nom que le fichier principal mais avec les extensions allant de 000 à 999 et comptant les parties Overlay référencées dans le programme principal. Si les procédures et les fonctions déclarées Overlay se suivent, elles feront partie du même fichier Overlay. Si elles sont "interrompues" par des procédures "normales", elles feront partie de fichiers Overlay différents. Cela est important pour la place que le programme doit réserver pour les parties Overlay.

Si les fichiers Overlay apparaissent tous à la suite les uns des autres dans le programme, ils feront partie d'un seul fichier principal et le programme ne réservera que la place prise par la procédure la plus importante du fichier Overlay. Si les fichiers Overlay apparaissent de manière non continue dans le programme, plusieurs fichiers Overlay seront créés (avec les extensions 000, 001, etc.) et le programme réservera dans la mémoire une place correspondant à la taille du fichier le plus important du premier groupe, une place correspondant à la taille du fichier le plus important du deuxième groupe, etc. Globalement, la place à réserver par la deuxième solution est plus importante, mais elle permet d'avoir plusieurs sous-programmes Overlay en même temps dans la mémoire. Ces derniers font nécessairement partie de fichiers Overlay différents.

Prenons le programme suivant pour illustrer la manière dont les données sont organisées :

```
Program Exemple;

  Overlay Procedure Premier;
  Begin
      ...
  End;

  Overlay Procedure Seconde;
  Begin
      ...
  End;

  Overlay Procedure Troisieme;
  Begin
      ...
  End;

  Overlay Procedure Quatrieme;
  Begin
      ...
  End;

Begin
  ...
End.
```

Dans ce cas, les quatre procédures Overlay seront mises dans un seul fichier Overlay. Le programme réservera en mémoire centrale une place égale à la place prise par la plus importante des quatre procédures Overlay. On peut faire le schéma suivant de l'organisation mémoire :



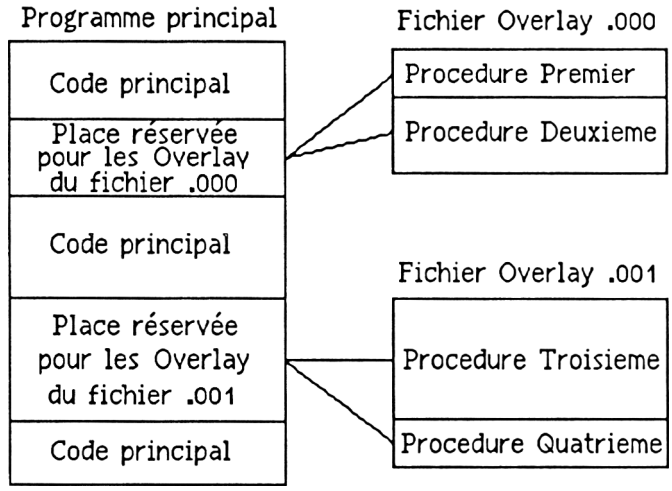
Dans cet exemple, il n'y a qu'un seul fichier Overlay contenant toutes les procédures Overlay et une seule place est réservée dans la mémoire pour loger l'une de ces procédures Overlay.

Prenons maintenant l'exemple du programme suivant :

```
Program Exemple2;

      Overlay Procedure Premier;
      Begin
          ...
      End;

      Overlay Procedure Seconde;
      Begin
          ...
      End;

      Procedure Intruse;
      Begin
          ...
      End;

      Overlay Procedure Troisieme;
      Begin
          ...
      End;

      Overlay Procedure Quatrieme;
      Begin
          ...
      End;

  Begin
      ...
  End;
```

Dans ce cas, l'organisation sera différente. En effet, les procédures Overlay ne se suivent pas dans le programme source. Le compilateur génèrera donc un fichier Overlay pour les procédures `Premiere` et `Seconde` (extension .000) et un autre avec les procédures `Troisieme` et `Quatrieme` (extension .001). L'organisation de la mémoire sera alors la suivante :

Système d'Overlay avec procédures ne se suivant pas dans le source



Ici, le programme réservera deux parties de sa mémoire. Elles auront chacune la taille de la procédure la plus longue qu'elles peuvent appeler : la première correspond à la taille maximale d'une procédure du premier fichier Overlay (extension .000) et la deuxième à la taille maximale d'une procédure du deuxième fichier Overlay (extension .001). Cela permet d'avoir en même temps deux procédures Overlay dans la mémoire de l'ordinateur : une faisant partie du premier fichier Overlay, l'autre du deuxième. Il ne faut cependant pas multiplier les fichiers Overlay, car plus il y en a et plus la place mobilisée en RAM est importante.

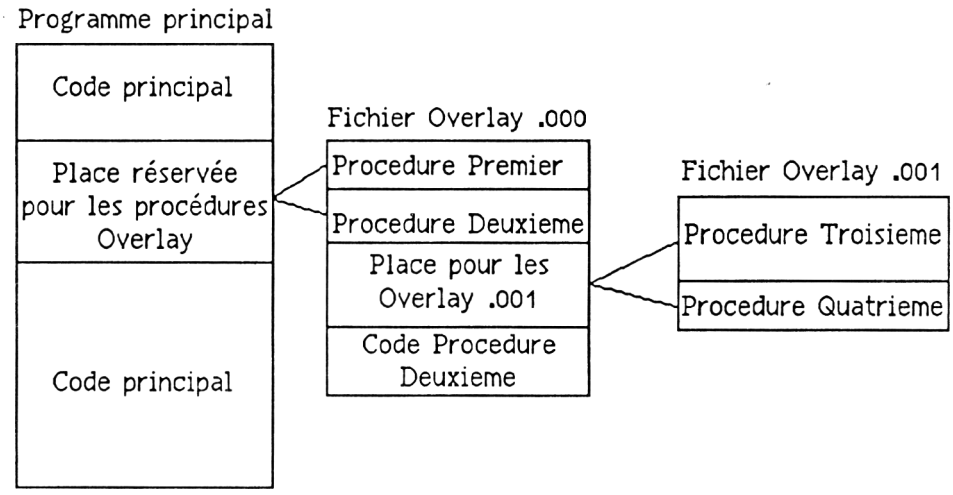
Nous n'avons pas encore vu les cas d'imbrication. En effet, et contrairement au inclusions vues précédemment, une procédure Overlay peut elle même contenir des procédures Overlay. Prenons l'exemple du programme suivant :

```
Program Exemple;

      Overlay Procedure Premier;
      Begin
         ...
      End;

      Overlay Procedure Seconde;

            Overlay Procedure Troisieme;
            Begin
               ...
            End;
```

```
            Overlay Procedure Quatrieme;
            Begin
                ...
            End;

    Begin
        ...
    End;


Begin
    ...
End.
```

Nous nous trouvons dans un cas où un fichier Overlay comporte une place pour inclure un autre fichier Overlay. A l'exécution, la structure des divers blocs en mémoire est alors la suivante :

Cas des procédures Overlay imbriquées



Ce principe peut être répété à n'importe quel niveau.

La gestion des fichiers Overlay est automatique. Une vérification est faite lors de l'appel d'une procédure Overlay pour vérifier si elle ne se trouve pas déjà en mémoire. Si c'est le cas, elle n'est pas rechargée depuis le disque, ce qui minimise le temps d'exécution, sinon, la partie demandée sera automatiquement lue dans le fichier Overlay approprié sur le disque par défaut.

Il se peut que vos fichiers Overlay ne résident pas tous sur le disque par défaut,

mais également sur un autre. Dans ce cas, la procédure **OvrDrive** permet d'aller chercher les fichiers Overlay sur un autre disque. Cette procédure nécessite un paramètre qui est le numéro du lecteur de disquettes. Ce paramètre vaut 0 pour le disque courant, 1 pour le disque A, 2 pour le B, etc. Cette procédure permet d'étendre encore plus la taille disponible pour les fichiers Overlay.

Certaines restrictions sont à apporter à l'utilisation des Overlay. En voici le détail :

- le fichier TURBO.OVR doit être présent sur la disquette d'exécution (disque par défaut);

- dans le cas de procédures Overlay faisant partie d'un même fichier, une procédure de ce bloc ne peut pas en appeler une autre du même bloc. Les sous-programmes Overlay peuvent donc utiliser le même emplacement de données ce qui minimise la place;

- les sous-programmes Overlay ne peuvent pas être déclarés en **Forward**. Ceci peut être contourné en utilisant une procédure Forward qui appelera les sous-programmes Overlay;

- les sous-programmes Overlay ne peuvent pas être récursifs. Cela peut également être contourné en utilisant une procédure récursive qui appelle à son tour un sous-programme Overlay.

L'utilisation de sous-programmes Overlay permet de résoudre aisément le problème de place mémoire dans votre Amstrad. Cependant, il ne faut pas tomber dans le "piège de l'Overlay" et l'utiliser même lorsque cela ne s'avère pas nécessaire. En effet, il faut savoir que le gain apporté à la taille du code est contrebalancé par la perte en rapidité d'exécution. Les accès sur disque sont en effet toujours plus longs que ceux en mémoire. Vous ne devez utiliser l'Overlay que si cette technique est nécessaire. Les parties déclarées en Overlay doivent donc être soigneusement étudiées afin de ne pas être appelées sans cesse par le programme principal.

CHAPITRE 16

AU CŒUR DU SYSTEME

HEAP, PILE ET PILE DE RECURSION

Trois structures de pile sont gérées en permanence par Turbo-Pascal. Vous connaissiez déjà la pile système, le **Heap**, auquel vient s'adjoindre la pile de récursion.

Lors de certaines opérations, il se peut que vous désiriez contrôler la taille et la progression de chacune de ces trois structures. Nous allons donc voir ici à quoi servent ces diverses piles, comment elles fonctionnent et dans quel sens elles évoluent.

Comme nous l'avons vu au chapitre 14 (Les pointeurs), les variables dynamiques sont rangées dans une structure nommée **Heap** au fur et à mesure de leur allocation. Ce **Heap** fonctionne de la même manière qu'une pile, mais progresse dans l'autre sens dans la mémoire de l'ordinateur. La taille du **Heap** évolue lorsqu'on utilise **New**, **Release**, **GetMem** et **FreeMem**. Lorsqu'on crée de nouvelles variables de manière dynamique, le **Heap** progresse dans le sens croissant des adresses mémoire. Au début de l'exécution d'un programme, une variable prédéfinie `HeapPtr` contient l'adresse du premier octet disponible pour le **Heap**. C'est en fait l'adresse du premier octet libre après le code exécutable.

La pile système sert à stocker des résultats intermédiaires pendant le calcul d'expressions. L'instruction **For** utilise également cette pile. Au début de l'exécution d'un programme, la variable prédéfinie `StackPtr` contient l'adresse du haut de la mémoire libre. La pile système croît en mémoire dans le sens des adresses hautes vers les adresses basses. Une place de 1Ko est réservée à cette pile. C'est largement suffisant dans le cas général. Cependant, cette taille est modifiable. Nous verrons comment dans quelques lignes.

La pile de récursion bien sûr n'est utilisée que dans le cas de procédures ou de fonctions récursives. Rappelons qu'il faut spécifier au compilateur la directive **{$A-}** pour avoir le droit d'utiliser des procédures ou des fonctions récursives. C'est à

l'entrée d'un sous-programme récursif qu'il y a copie dans la pile de récursion de l'espace de travail. Il y a restitution à la sortie. Au début d'un programme, la variable prédéfinie `RecurPtr` contient l'adresse de départ de la pile de récursion. Cette pile progresse, comme la pile système, dans le sens des adresses décroissantes et se trouve à l'origine 1 Ko plus haut que la pile système.

Les trois variables `HeapPtr`, `StackPtr` et `RecurPtr` sont de type entier. `HeapPtr` et `RecurPtr` peuvent être positionnées par le programmeur à une autre place que celle d'origine alors qu'on ne peut pas affecter de valeur à `StackPtr`. Si le programmeur déplace certains éléments en mémoire, il doit cependant s'assurer que la double inégalité suivante est respectée :

```
HeapPtr < RecurPtr < StackPtr
```

Si vous passez outre, les résultats pourront être aussi bizarres qu'étranges ! De plus, une fois qu'une des piles aura été utilisée, il ne faudra plus toucher à la variable lui correspondant.

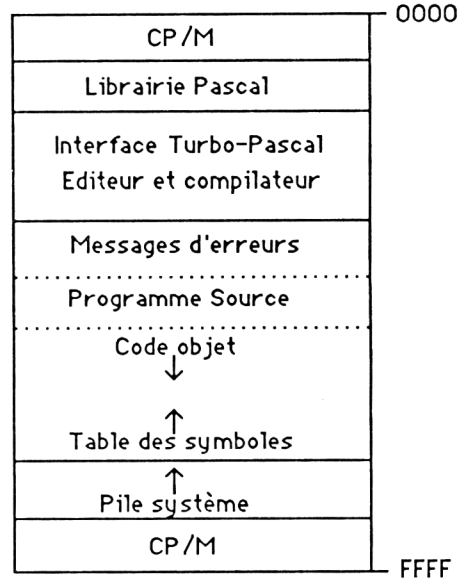
Pour ce qui est de l'exécution, il y a vérification que la pile système ne "grignote" pas la place de la pile de récursion. Si la pile système s'étend trop, il y a génération d'un message d'erreur. Si vous désirez étendre la taille disponible pour la pile système, il suffit de déplacer la variable `Recurptr` au début du programme. Par exemple l'instruction suivante fixe la taille de la pile système à 2 Ko au lieu de 1 Ko :

```
RecurPrt := StackPtr - 2*1024;
```

ETAT DE LA MEMOIRE A DIVERS STADES

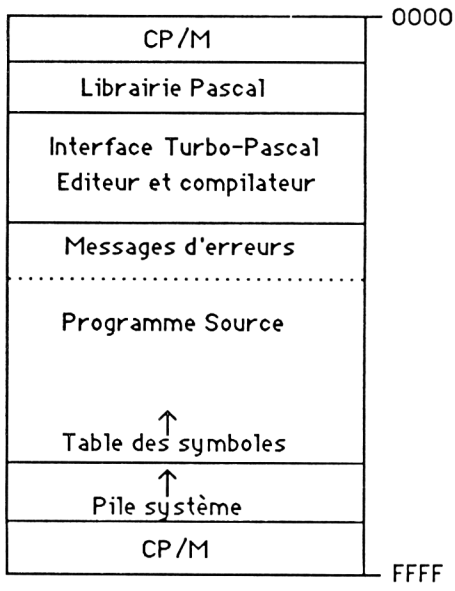
Compilation en mémoire

Le tableau suivant montre l'état de la mémoire lorsqu'on effectue une compilation en mémoire :



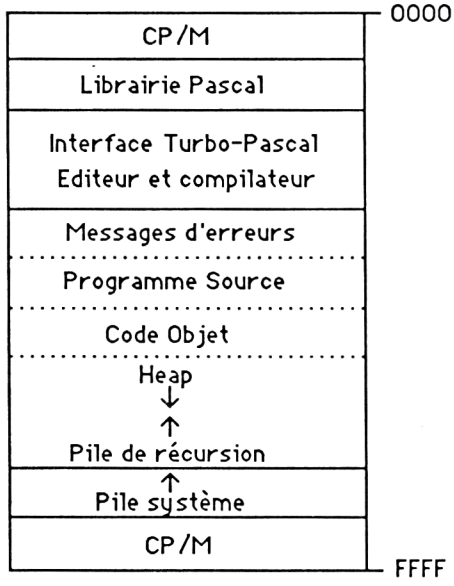
Compilation sur disque

Le code objet est directement stocké sur disque. Voici la carte de la mémoire dans cette phase :



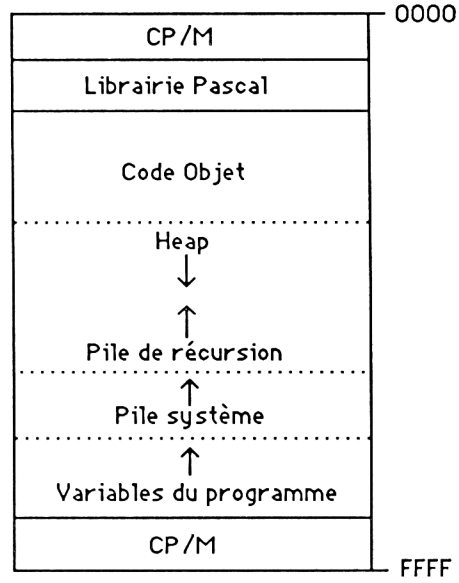
Exécution en mémoire

Lorsqu'un programme est exécuté directement depuis le compilateur (par la commande **Run**), la situation est différente : en effet, en plus des éléments que nous possédons déjà, il faut loger le **Heap** et la pile de récursion. C'est seulement après la phase de compilation que toutes les tailles sont connues et que les divers pointeurs de pile peuvent être positionnés. Voici l'état de la mémoire dans cette phase :



Exécution d'un programme

La carte mémoire que nous allons voir à présent est celle qui correspond à l'exécution d'un programme, soit depuis CP/M, soit par e**X**ecute depuis Turbo-Pascal. La mémoire a alors l'allure suivante :



LE FORMAT INTERNE DES DONNEES

Les variables et les types déclarés occupent en mémoire le nombre d'octets retourné par la fonction **SizeOf**. Il est cependant utile de connaître la manière dont les variables sont stockées :

- **Scalaires**

  1) Sont rangés sur un seul octet contenant la valeur ordinale de la variable :

  - les bytes et les intervalles d'entiers ayant moins de 256 éléments (en complément à 2) ;
  - les caractères ;
  - les scalaires déclarées ayant moins de 256 valeurs possibles.

  2) Sont rangés sur deux octets en complément à 2 (octet de poids faible en premier) :

  - les entiers ;
  - les intervalles d'entiers ayant plus de 256 éléments ;
  - les scalaires déclarées ayant plus de 256 valeurs possibles.

- **Réels**
  Ils sont stockés en virgule flottante sur 6 octets : mantisse sur 40 bits et exposant sur 8 bits. L'exposant est rangé dans le premier octet, sa valeur étant augmentée de $80 (cf chapitre 15, Passage de paramètres). La mantisse est normalisée et le bit de poids le plus fort est toujours considéré comme étant à 1. En fait il sert à indiquer le signe : 1 pour négatif, 0 pour positif.

- **Chaînes**
  Les chaînes occupent **Length(**`Chaine`**)+1** octets. `Chaine[0]`, c'est-à-dire le premier octet, contient la longueur de chaîne, et les octets suivants les caractères composant la chaîne, en allant vers les adresses hautes. Les caractères compris entre la longueur courante et la longueur maximale déclarée sont aléatoires (vieux octets).

- **Pointeurs**
  Ils sont codés sur 16 bits (représentant l'adresse mémoire), avec l'octet de poids faible en premier. **Nil** vaut 0.

- **Tableaux**
  Les éléments de valeurs d'index les plus basses sont rangés aux plus basses adresses mémoire. Si le tableau est multidimensionnel, l'index le plus à droite est rangé à la plus basse adresse mémoire.

- **Ensembles**
  Un ensemble ne peut contenir plus de 256 éléments. Chaque élément est codé sur un seul bit, ce qui fait qu'une variable de type ensemble n'occupe jamais plus de 32 octets. Des compressions sont effectuées par le compilateur en supprimant les octets nuls. Il est donc très difficile de travailler directement dans les codes des variables de type ensemble.

- **Enregistrements**
  Le premier champ est rangé à l'adresse mémoire la plus basse. S'il n'y a pas de partie variable, la longueur totale est donnée par la somme des longueurs de chaque champ. S'il y a une partie variable, elle a pour longueur celle de la plus grande variante qu'elle peut contenir. Chaque variante commence à la même adresse mémoire.

L'ACCES DIRECT A LA MEMOIRE

Variables absolues

Il est possible d'obliger une variable à résider à une adresse spécifique. Elle est dans ce cas appelée variable absolue. On la déclare à l'aide du mot réservé **Absolute** suivi d'une adresse (constante entière) :

```
Var Octet : Byte Absolute $FBCD;
```

Absolute peut servir à superposer une variable au dessus d'une autre.

Exemple :

```
Var   Chaine : String[40];
      Longueur : Byte Absolute Chaine;
```

Ici `Longueur` contiendra directement le longueur de `Chaine` (cf codage des chaînes).

Fonction Addr

Addr

Addr(`Nom`**);**
Retourne un **entier** contenant l'adresse mémoire du premier octet du type, de la variable, de la procédure ou de la fonction ayant `Nom` pour identificateur. Si `Nom` est un tableau, il peut être indexé; si c'est un enregistrement, on peut en choisir le champ.

Tableaux prédéfinis

Deux tableaux permettent en Turbo-Pascal d'accéder facilement à la mémoire ou à un port d'entrée/sortie.

Le tableau prédéfini **Mem** permet d'accéder directement à un octet donné de la mémoire de l'ordinateur. L'index de ce tableau est de type entier et détermine l'adresse à laquelle on veut accéder. On peut ainsi lire ce qu'il y a à une adresse donnée en regardant la valeur de **Mem[**`Adr`**]**, où `Adr` représente l'adresse que l'on désire observer, ou stocker des données en mémoire en affectant des valeurs à des cases du tableau **Mem**. Il est ainsi très facile de simuler les fonctions PEEK et POKE du Basic.

Le tableau **Port** permet d'accéder aux ports d'entrée/sortie de votre Amstrad. L'index que vous devez fournir est un octet car le Z80 ne peut avoir plus de 256 ports d'entrée/sortie. Un élément du tableau **Port** peut être affecté d'une valeur, dans ce cas, la donnée fournie est sortie sur le port correspondant. Si un élément du tableau **Port** est utilisé dans une expression, l'entrée se fera par la lecture du port correspondant. L'emploi de ce tableau est limité aux affectations et références et on ne peut pas passer ce tableau par variable dans une procédure ou une fonction. De plus, aucune opération se servant du tableau entier n'est autorisée. On ne peut se servir que des divers ports séparément.

LA GESTION DES INTERRUPTIONS

On peut écrire en Turbo-Pascal des routines d'interruption en utilisant l'instruction **InLine**. La gestion des interruptions est donc la même qu'en Assembleur. Il faut seulement respecter les règles suivantes :

- le code doit être généré avec l'option de compilation **{$A+}**;

- l'état des registres internes du Z80 doit être préservé;

- une routine d'interruption ne peut pas utiliser d'opération d'entrée/sortie par les procédures et les fonctions prédéfinies de Turbo-Pascal, car celles-ci ne sont pas réentrantes.

ANNEXE 1

CODES ASCII

Décimal	Hexa	Caractère		Décimal	Hexa	Caractère
000	00	NUL	(Ctrl @)	029	1D	GS
001	01	SOH	(Ctrl A)	030	1E	RS
002	02	STX	(Ctrl B)	031	1F	US
003	03	ETX	(Ctrl C)	032	20	SP
004	04	EOT	(Ctrl D)	033	21	!
005	05	ENQ	(Ctrl E)	034	22	"
006	06	ACK	(Ctrl F)	035	23	#
007	07	BEL	(Ctrl G)	036	24	$
008	08	BS	(Ctrl H)	037	25	%
009	09	HT	(Ctrl I)	038	26	&
010	0A	LF	(Ctrl J)	039	27	'
011	0B	VT	(Ctrl K)	040	28	(
012	0C	FF	(Ctrl L)	041	29	)
013	0D	CR	(Ctrl M)	042	2A	*
014	0E	SO	(Ctrl N)	043	2B	+
015	0F	SI	(Ctrl O)	044	2C	,
016	10	DLE	(Ctrl P)	045	2D	-
017	11	DC1	(Ctrl Q)	046	2E	.
018	12	DC2	(Ctrl R)	047	2F	/
019	13	DC3	(Ctrl S)	048	30	0
020	14	DC4	(Ctrl T)	049	31	1
021	15	NAK	(Ctrl U)	050	32	2
022	16	SYN	(Ctrl V)	051	33	3
023	17	ETB	(Ctrl W)	052	34	4
024	18	CAN	(Ctrl X)	053	35	5
025	19	EM	(Ctrl Y)	054	36	6
026	1A	SUB	(Ctrl Z)	055	37	7
027	1B	ESC		056	38	8
028	1C	FS		057	39	9

Décimal	Hexa	Caractère
058	3A	:
059	3B	;
060	3C	<
061	3D	=
062	3E	>
063	3F	?
064	40	@
065	41	A
066	42	B
067	43	C
068	44	D
069	45	E
070	46	F
071	47	G
072	48	H
073	49	I
074	4A	J
075	4B	K
076	4C	L
077	4D	M
078	4E	N
079	4F	O
080	50	P
081	51	Q
082	52	R
083	53	S
084	54	T
085	55	U
086	56	V
087	57	W
088	58	X
089	59	Y
090	5A	Z
091	5B	[
092	5C	\
093	5D	]
094	5E	†
095	5F	—
096	60	‘
097	61	a
098	62	b
099	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	
127	7F	

ANNEXE 2

PROCEDURES ET FONCTIONS DE TURBO-PASCAL

Les numéros renvoient aux pages où elles sont traitées en détail.

PAR CLASSES D'APPARTENANCE

Les fonctions mathématiques

Abs(nombre); 75
Retourne la valeur absolue de nombre. Le résultat est du même type que l'argument.

ArcTan(nombre); 75
Retourne l'angle (en radians) dont la tangente est nombre. Le résultat est toujours un **réel**.

Cos(nombre); 75
Retourne le cosinus de nombre (radians). Le résultat est toujours un **réel**.

Exp(nombre); 75
Retourne l'exponentielle de nombre. Le résultat est toujours un **réel**.

Frac(nombre); 75
Retourne la partie fractionnaire de nombre. Le résultat est toujours un **réel**.

Int(nombre); 75
Retourne la partie entière de nombre. Le résultat est toujours un **réel**.

Ln(nombre); 75
Retourne le logarithme népérien de nombre. Le résultat est toujours un **réel**.

Odd(nombre); 75
Retourne une valeur booléenne : **True** (vraie) si nombre est impair, **False** (fausse) sinon.

Random; 76
Retourne un nombre **réel aléatoire** compris entre 0 et 1.

Random(nombre); 76
Retourne un **entier** compris entre 0 et nombre, qui doit être un entier.

Round(nombre); 76
Arrondit le réel nombre à l'entier le plus proche.

Sin(nombre); 76
Retourne le sinus de nombre (radians). Le résultat est toujours un **réel**.

Sqr(nombre); 76
Retourne le carré de nombre. Le résultat est du même type que l'argument.

Sqrt(nombre); 76
Retourne la racine carrée de nombre. Le résultat est un **réel**.

Trunc(nombre); 76
Enlève la partie décimale du réel nombre. Le résultat est bien sûr un **entier**.

Fonctions et procédures sur les chaînes

Fonctions

Concat(mot1, mot2, ... ,motn); 74
Retourne la **chaîne** qui est la concaténation de mot1, mot2... motn, dans l'ordre où ils se présentent.

Copy(mot, pos, nbr); 73
Retourne une **chaîne** contenant nbr caractères de mot, à partir de la position pos.

Length(mot); 74
Retourne un **entier** contenant la longueur de mot (mot est une expression chaîne).

Pos(mot1,mot2); 74
Retourne un **entier** contenant la position de la première occurrence de mot1 dans mot2.

Procédures

Delete(mot, pos, nbr); 74
Enlève nbr caractères à la variable chaîne mot à partir de la position pos.

Insert(mot1, mot2, pos); 74
Insère l'expression chaîne mot1 dans la variable chaîne mot2, à la position pos.

Str(valeur, mot); 74
Convertit une valeur numérique entière ou réelle en une **chaîne**.

Val(mot, var, erreur); 75
Convertit la chaîne mot, qui doit représenter un nombre (sous forme entière, décimale, hexadécimale ou scientifique) en une valeur numérique rangée dans la variable var qui doit être entière ou réelle. erreur est une variable entière qu'on doit déclarer avant d'utiliser **Val()**.

Fonctions et procédures d'usage général

Fonctions

Chr(nombre); 76
Retourne le caractère de code ASCII nombre.

Hi(nombre); 76
Retourne un **entier** dont l'octet de poids fort est à 0 et dont l'octet de poids faible contient l'octet de poids fort de nombre.

Keypressed; 76
Retourne la valeur booléenne **True** si une touche a été frappée au clavier, **False** sinon.

Lo(nombre); 77
Retourne un **entier** dont l'octet de poids fort est à 0 et dont l'octet de poids faible contient l'octet de poids faible de nombre.

Ord(valeur); 64,77
Retourne un **entier** donnant la position ordinale de valeur dans son type. valeur peut être de tout type scalaire sauf réel.

Pred(valeur); 64,77
Retourne la valeur précédente de valeur, dans l'ordre défini par la déclaration de type.

Sizeof(`nom`**);** 77
Retourne le nombre d'octets occupés en mémoire par la variable ou le type `nom`. Le résultat est de type **entier**.

Succ(`valeur`**);** 64,77
Retourne la valeur suivante de `valeur` : fonction inverse de la précédente. `Succ(dernier)` n'est pas défini.

Swap(`nombre`**);** 77
Echange les octets de poids fort et faible de `nombre`, qui doit être un **entier**.

UpCase(`lettre`**);** 77
Retourne la majuscule correspondant à `lettre`.

Procédures

ClrEol; 77
Efface toute la fin de la ligne depuis le curseur, sans bouger celui-ci (Clear End of Line).

ClrScr; 77
Efface l'écran, remet le curseur en haut à gauche.

CrtExit; 78
Ecrit la chaîne de réinitialisation définie lors de l'installation.

CrtInit; 77
Ecrit la chaîne d'initialisation définie lors de l'installation.

Delay(`temps`**);** 78
Crée une attente d'environ `temps` x1,5 millisecondes. `temps` doit être un **entier**.

DelLine; 78
Efface la ligne où se trouve le curseur, et remonte les suivantes vers le haut.

Exit; 78
Provoque la sortie du bloc en cours d'exécution.

FillChar(`variable, nombre, valeur`**);** 78
Remplit `nombre` octets, à partir du premier octet occupé par `variable`, avec la valeur `valeur`.

GotoXY(`horiz,vertic`**);** 78
Déplace le curseur jusqu'à la position de coordonnées spécifiées.

Halt; 78
Provoque la fin du programme, et le retour sous Turbo-Pascal.

InsLine; 78
Insère une ligne à la position du curseur, et déplace les suivantes vers le bas.

LowVideo; 78
Jusqu'à la rencontre de **NormVideo**, tous les caractères sont affichés en vidéo inverse.

Move(`variable1,variable2,nombre`**);** 78
Recopie un bloc de `nombre` octets pris à partir du premier octet de `variable1` à l'emplacement commençant au premier octet de `variable2`.

NormVideo; 79
Provoque le retour à l'affichage normal, s'il avait été affecté précédemment par **LowVideo**.

Randomize; 79
Initialise le générateur de nombre aléatoire.

Fonctions et procédures opérant sur les fichiers

Fonctions

Eof(`Nom`**);** 121
Teste si la fin du fichier `Nom` est atteinte.

FilePos(`Nom_Logique`**);** 124
Fournit le numéro d'ordre de l'élément sur lequel se trouve le pointeur du fichier `Nom_Logique`.

FileSize(`Nom_Logique`**);** 125
Fournit la taille en nombre d'éléments du fichier `Nom_Logique`.

IOResult; 132
Si une erreur d'entrée/sortie s'est produite, **IOResult** retourne son numéro, sinon elle retourne 0.

Procédures

Assign(`Nom_Logique,Nom_Physique`**);** 121
Assigne `Nom_logique` au fichier `Nom_Physique`.

BlockRead(Nom_Logique,Variable,Nbr,[Total]); 130
Lit dans le fichier Nom_Logique pour mettre dans Variable une quantité de Nbr*128 octets. Le paramètre Total est facultatif. Dans le cas où on le spécifie, il indique le nombre d'enregistrements de 128 octets effectivement transférés.

BlockWrite(Nom_Logique,Variable,Nbr,[Total]); 130
Ecrit dans le fichier associé à Nom_Logique les 128*Nbr premiers octets de Variable. Le paramètre Total est facultatif et joue le même rôle que dans **BlockRead**.

Close(Nom); 121
Ferme le fichier Nom.

Erase(Nom_Logique); 125
Efface le fichier ayant pour nom de travail Nom_logique.

Flush(Nom_Logique); 125
Ecrit "physiquement" ce qui se trouve dans le buffer.

Read(Nom,Variable); 44,121, 134
Lit la valeur se trouvant dans le fichier Nom (le clavier par défaut) et l'affecte à Variable, puis avance d'une unité pour la prochaine lecture.

ReadLn(Nom_Logique); 44, 129, 134
Saute par dessus tous les caractères jusqu'à la prochaine ligne (juste après la première marque CR-LF rencontrée). Ira à la ligne après une entrée clavier.

Rename(Nom_Logique,Nouveau_Nom); 125
Renomme le fichier de nom de travail Nom_Logique.

Reset(Nom); 122
Ouvre le fichier Nom pour lecture.

ReWrite(Nom); 122
Crée le fichier Nom et le prépare en écriture.

Seek(Nom_Logique,Nombre); 125
Positionne le pointeur de fichier sur l'élément du numéro d'ordre Nombre.

Write(Nom,Variable); 44, 122, 135
Ecrit la valeur de Variable dans le fichier Nom (l'écran par défaut) et avance d'une unité pour la prochaine écriture.
WriteLn cf **Write**.

Fonctions et procédures opérant sur les fichiers texte

Fonctions

EoIn(`Nom_Logique`); 129
Retourne **True** si l'on se trouve sur le caractère CR de fin de ligne et **False** dans le cas contraire.

SeekEoIn(`Nom_logique`); 129
Identique à **EoIn** mais supprime les espaces et les caractères de tabulation avant d'effectuer le test.

SeekEof(`Nom_Logique`); 129
Identique à **Eof** mais supprime les espaces et les caractères de tabulation avant d'effectuer le test.

Procédures

ReadLn(`Nom_Logique`); 129
Cette procédure permet de sauter par dessus tous les caractères jusqu'à la prochaine ligne (juste après la première marque CR-LF rencontrée).

WriteLn(`Nom_Logique`); 129
Introduit dans le fichier texte assigné à `Nom_Logique` une séquence CR-LF pour marquer la fin de ligne.

Fonctions et procédures opérant sur les pointeurs

Fonctions

MaxAvail; 146
Retourne un **entier** contenant la taille (en octets) du plus grand espace (trou) disponible dans le **Heap**.

MemAvail; 146
Retourne un **entier** contenant l'espace disponible dans le **Heap** à un instant donné. Cette valeur permet de savoir si l'on peut ou non demander la création d'une nouvelle variable dynamique.

Ord(`Ptr`); 147
Retourne sous forme d'un **entier** l'adresse sur laquelle pointe la variable `Ptr` .

Ptr(`Ent`); 147
Convertit le paramètre `Ent` (qui doit être de type **entier**) en un pointeur compatible avec tous les types de pointeurs.

Procédures

Dispose(`Ptr`); 147
Permet de récupérer la place du pointeur `Ptr` dans le **Heap**.

FreeMem(`Ptr,Ent`); 147
Récupère un bloc entier dans le **Heap**. `Ptr` pointe sur le début de la place à libérer et `Ent` est le nombre d'octets à récupérer.

GetMem(`Ptr,Ent`); 147
Alloue dans le **Heap** une place définie. `Ptr` est une variable de type pointeur qui pointera sur l'espace alloué et `Ent` est une expression entière qui détermine le nombre d'octets à allouer.

Mark(`Ptr`); 148
Affecte la valeur du pointeur de **Heap** à la variable `Ptr`. Cela permet de mémoriser l'état du **Heap** à un instant donné. Cet état pourra ensuite être restitué par la procédure **Release**.

New(`Ptr`); 148
Crée une variable associée au pointeur de variable `Ptr`. La variable créée sera référencée par `Ptr^`.

Release(`Ptr`); 148
Affecte la valeur `Ptr` au pointeur de **Heap** (`Ptr` est un pointeur préalablement initialisé par **Mark**). Détruit toutes les variables dynamiques créées depuis l'exécution de **Mark**.

Fonctions et procédures accédant à CP/M

Fonctions

Bdos; 150
Retourne un résultat entier qui correspond à ce que le **Bdos** retourne dans le registre A du Z80 après avoir exécuté la fonction CP/M demandée par l'utilisateur.

BdosHL(`C,DE`); 150
Identique à **Bdos** mais elle retourne ce qui se trouve dans le registre double HL du Z80 après l'exécution de la fonction **Bdos** demandée par l'utilisateur.

Bios; 151
Retourne un résultat entier qui correspond à ce que le **Bios** retourne dans le registre A du Z80 après avoir exécuté la fonction CP/M demandée par l'utilisateur.

BiosHL(`C,DE`**);** 151
Identique à **Bios** mais elle retourne ce qui se trouve dans le registre double HL du Z80 après l'exécution de la fonction **Bios** demandée par l'utilisateur.

Paramcount; 151
Retourne le nombre de paramètres passés au programme depuis la ligne de commande CP/M.

Paramstr(`n`**);** 151
Retourne une **chaîne** contenant le n^{è} paramètre de la ligne passé au programme depuis la ligne de commande CP/M.

Procédures

Bdos(`C,DE`**);** 150
Charge les registres C et DE par les valeurs fournies et exécute un appel à l'adresse 5 pour appeler le **Bdos**.

Bios(`Func,BC`**);** 150
Appelle les routines du **Bios**. `Func` est un entier qui indique le numéro de la routine à appeler et `BC` est un paramètre facultatif à charger dans le double registre BC du Z80 avant l'appel. `Func` correspond aux numéros des routines de CP/M (WBOOT =0, CONST=1 etc...)

Chain(`Nom_Logique`**);** 152
Provoque l'exécution d'un fichier .CHN.

Execute(`Nom_Logique`**);** 151
Provoque l'exécution d'un fichier .COM.

Accès à la mémoire

Fonction

Addr(`Nom`**);** 170
Retourne un **entier** contenant l'adresse du premier octet de `Nom` spécifié.

Tableaux prédéfinis

Mem 170
Tableau prédéfini représentant la mémoire.

Port 170
Tableau prédéfini représentant les ports d'entrée/sortie.

PAR ORDRE ALPHABETIQUE

Procédures

Assign(`Nom_Logique,Nom_Physique`); 121
Assigne `Nom_logique` au fichier `Nom_Physique`.

Bdos(`C,DE`); 150
Charge les registres C et DE par les valeurs fournies et exécute un appel à l'adresse 5 pour appeler le **Bdos**.

Bios(`Func,BC`); 150
Appelle les routines du **Bios**. `Func` est un entier qui indique le numéro de la routine à appeler et `BC` est un paramètre facultatif à charger dans le double registre BC du Z80 avant l'appel. `Func` correspond aux numéros des routines de CP/M (WBOOT =0, CONST=1 etc...)

BlockRead(`Nom_Logique,Variable,Nbr,[Total]`); 130
Lit dans le fichier `Nom_Logique` pour mettre dans `Variable` une quantité de `Nbr*128` octets. Le paramètre `Total` est facultatif. Dans le cas où on le spécifie, il indique le nombre d'enregistrements de 128 octets effectivement transférés.

BlockWrite(`Nom_Logique,Variable,Nbr,[Total]`); 130
Ecrit dans le fichier associé à `Nom_Logique` les `128*Nbr` premiers octets de `Variable`. Le paramètre `Total` est facultatif et joue le même rôle que dans **BlockRead**.

Chain(`Nom_Logique`); 152
Provoque l'exécution d'un fichier .CHN.

Close(`Nom`); 121
Ferme le fichier `Nom`.

ClrEol; 77
Efface toute la fin de la ligne depuis le curseur, sans bouger celui-ci (Clear End of Line).

ClrScr; 77
Efface l'écran, remet le curseur en haut à gauche.

CrtExit; 78
Ecriture de la chaîne de réinitialisation définie lors del'installation.

CrtInit; 77
Ecrit la chaîne d'initialisation définie lors de l'installation.

Delay(`temps`**);** 78
Crée une attente d'environ `temps` x1,5 millisecondes. `temps` doit être un **entier**.

Delete(`mot, pos, nbr`**);** 74
Enlève `nbr` caractères à la variable chaîne `mot` à partir de la position `pos`.

DelLine; 78
Efface la ligne où se trouve le curseur, et remonte les suivantes vers le haut.

Dispose(`Ptr`**);** 147
Permet de récupérer la place du pointeur `Ptr` dans le **Heap**.

Erase(`Nom_Logique`**);** 125
Efface le fichier ayant pour nom de travail `Nom_logique`.

Execute(`Nom_Logique`**);** 151
Provoque l'exécution d'un fichier .COM.

Exit; 78
Provoque la sortie du bloc en cours d'exécution.

FillChar(`variable, nombre, valeur`**);** 78
Remplit `nombre` octets, à partir du premier octet occupé par `variable`, avec la valeur `valeur`.

Flush(`Nom_Logique`**);** 125
Cette procédure écrit "physiquement" ce qui se trouve dans le buffer.

FreeMem(`Ptr,Ent`**);** 147
Récupère un bloc entier dans le **Heap**. `Ptr` pointe sur le début de la place à libérer et `Ent` est le nombre d'octets à récupérer.

GetMem(`Ptr,Ent`**);** 147
Alloue dans le **Heap** une place définie. `Ptr` est une variable de type pointeur qui pointera sur l'espace alloué et `Ent` est une expression entière qui détermine nombre d'octets à allouer.

GotoXY(`horiz,vertic`**);** 78
Déplace le curseur jusqu'à la position de coordonnées spécifiées.

Halt; 78
Provoque la fin du programme, et le retour sous Turbo-Pascal.

Insert(`mot1, mot2, pos`); 74
Insère l'expression chaîne `mot1` dans la variable chaine `mot2`, à la position `pos`.

InsLine; 78
Insère une ligne à la position du curseur, et déplace les suivantes vers le bas.

LowVideo; 78
Jusqu'à la rencontre de **NormVideo**, tous les caractères sont affichés en vidéo inverse.

Mark(`Ptr`); 148
Affecte la valeur du pointeur de **Heap** à la variable `Ptr`. Cela permet de mémoriser l'état du **Heap** à un instant donné. Cet état pourra ensuite être restitué par la procédure **Release**.

Move(`variable1,variable2,nombre`); 78
Recopie un bloc de `nombre` octets pris à partir du premier octet de `variable1` à l'emplacement commençant au premier octet de `variable2`.

New(`Ptr`); 148
Crée une variable associée au pointeur de variable `Ptr`. La variable créée sera référencée par `Ptr^`.

NormVideo; 79
Provoque le retour à l'affichage normal, s'il avait été affecté précédemment par **LowVideo**.

Randomize; 79
Initialise le générateur de nombre aléatoire.

Read(`Nom,Variable`); 44, 121, 134
Lit la valeur se trouvant dans le fichier `Nom` (le clavier par défaut) et l'affecte à `Variable`, puis avance le pointeur de fichier d'une unité pour la prochaine lecture.

ReadLn(`Nom_Logique`); 44, 129, 134
Saute par dessus tous les caractères jusqu'à la prochaine ligne (juste après la première marque CR-LF rencontrée). Ira à la ligne après une entrée clavier.

Release(`Ptr`); 148
Affecte la valeur `Ptr` au pointeur de **Heap** (`Ptr` est un pointeur préalablement initialisé par **Mark**). Détruit toutes les variables dynamiques créées depuis l'exécution de **Mark**.

Rename(`Nom_Logique,Nouveau_Nom`); 125
Renomme le fichier de nom de travail `Nom_Logique`.

Reset(Nom); 122
Ouvre le fichier Nom pour lecture.

ReWrite(Nom); 122
Crée le fichier Nom et le prépare en écriture.

Seek(Nom_Logique,Nombre); 125
Positionne le pointeur de fichier sur l'élément du numéro d'ordre Nombre.

Str(valeur, mot); 74
Convertit une valeur numérique entière ou réelle en une **chaîne**.

Swap(nombre); 77
Echange les octets de poids fort et faible de nombre, qui doit être un **entier**.

Val(mot, var, erreur); 75
Convertit la chaîne mot, qui doit représenter un nombre (sous forme entière, décimale, hexadécimale ou scientifique) en une valeur numérique rangée dans la variable var qui doit être entière ou réelle. erreur est une variable entière qu'on doit déclarer avant d'utiliser **Val()**.

Write(Nom,Variable); 44, 122, 135
Ecrit la valeur de Variable dans le fichier Nom et avance d'une unité pour la prochaine écriture.

WriteLn(Nom_Logique); 44, 129, 134
Introduit dans le fichier texte assigné à Nom_Logique une séquence CR-LF pour marquer la fin de ligne.

Fonctions

Abs(nombre); 75
Retourne la valeur absolue de nombre. Le résultat est du même type que l'argument.

Addr(Nom); 170
Retourne un **entier** contenant l'adresse du premier octet de Nom spécifié.

ArcTan(nombre); 75
Retourne l'angle (en radians) dont la tangente est nombre. Le résultat est toujours un **réel**.

Bdos; 150
Retourne un résultat entier qui correspond à ce que le **Bdos** retourne dans le registre A du Z80 après avoir exécuté la fonction CP/M demandée par l'utilisateur.

BdosHL(C,DE); 150
Identique à **Bdos** mais elle retourne ce qui se trouve dans le registre double HL du Z80 après l'exécution de la fonction **Bdos** demandée par l'utilisateur.

Bios; 151
Retourne un résultat entier qui correspond à ce que le **Bios** retourne dans le registre A du Z80 après avoir exécuté la fonction CP/M demandée par l'utilisateur.

BiosHL(C,DE); 151
Identique à **Bios** mais elle retourne se qui ce trouve dans le registre double HL du Z80 après l'exécution de la fonction **Bios** demandée par l'utilisateur.

Chr(nombre); 76
Retourne le caractère de code ASCII nombre.

Concat(mot1, mot2, ... ,motn); 74
Retourne la **chaîne** qui est la concaténation de mot1, mot2... motn, dans l'ordre où ils se présentent.

Copy(mot, pos, nbr); 73
Retourne une **chaîne** contenant nbr caractères de mot, à partir de la position pos.

Cos(nombre); 75
Retourne le cosinus de nombre (radians). Le résultat est toujours un **réel**.

Eof(Nom); 121
Teste si la fin du fichier Nom est atteinte.

Eoln(Nom_Logique); 129
Retourne **True** si l'on se trouve sur le caractère CR de fin de ligne et **False** dans le cas contraire.

Exp(nombre); 75
Retourne l'exponentielle de nombre. Le résultat est toujours un **réel**.

FilePos(Nom_Logique); 124
Retourne le numéro d'ordre de l'élément sur lequel se trouve le pointeur du fichier Nom_Logique.

FileSize(Nom_Logique); 125
Retourne la taille en nombre d'éléments du fichier Nom_Logique.

Frac(nombre); 75
Retourne la partie fractionnaire de nombre. Le résultat est toujours un **réel**.

Hi(nombre); 76
Retourne un entier dont l'octet de poids fort est à 0 et dont l'octet de poids faible contient l'octet de poids fort de nombre.

Int(nombre); 75
Retourne la partie entière de nombre. Le résultat est toujours un **réel**.

IOResult; 132
Si une erreur d'entrée/sortie s'est produite, retourne le numéro de celle-ci, sinon retourne 0.

Keypressed; 76
Retourne la valeur booléenne **True** si une touche a été frappée au clavier, **False** sinon.

Length(mot); 74
Retourne un **entier** contenant la longueur de mot (mot est une expression chaîne).

Ln(nombre); 75
Retourne le logarithme népérien de nombre. Le résultat est toujours un **réel**.

Lo(nombre); 77
Retourne un **entier** dont l'octet de poids fort est à 0 et dont l'octet de poids faible contient l'octet de poids faible de nombre.

MaxAvail; 146
Retourne un **entier** contenant la taille (en octets) du plus grand espace disponible dans le **Heap**.

MemAvail; 146
Retourne un **entier** contenant l'espace disponible dans le **Heap** à un instant donné. Cette valeur permet de savoir si l'on peut ou non demander la création d'une nouvelle variable dynamique.

Odd(nombre); 75
Retourne une valeur booléenne : **True** (vraie) si nombre est impair, **False** (fausse) sinon.

Ord(Ptr); 147
Cette fonction retourne un **entier** contenant l'adresse sur laquelle la variable Ptr (qui doit être de type pointeur) pointe. Utilisation déconseillée au débutant.

Ord(valeur); 64, 77
Retourne un **entier** donnant la position ordinale de valeur dans son type. valeur peut être de tout type scalaire sauf réel.

Paramcount; 151
Retourne le nombre de paramètres passés au programme depuis la ligne de commande CP/M.

Paramstr(`n`**);** 151
Retourne une **chaîne** contenent le n^{e} paramètre de la ligne passé au programme depuis la ligne de commande CP/M.

Pos(`mot1,mot2`**);** 74
Retourne un **entier** contenant la position de la première occurrence de `mot1` dans `mot2`.

Pred(`valeur`**);** 64, 77
Retourne la valeur précédente de `valeur`, dans l'ordre défini par la déclaration de type.

Ptr(`Ent`**);** 147
Convertit le paramètre `Ent` (qui doit être de type entier) en un pointeur compatible avec tous les types de pointeurs (opération inverse de **Ord**). Permet d'examiner tout endroit de la mémoire. Egalement déconseillée aux débutants.

Random; 76
Retourne un nombre réel aléatoire compris entre 0 et 1.

Random(`nombre`**);** 76
Retourne un **entier** compris entre 0 et `nombre`, qui doit être un entier.

Round(`nombre`**);** 76
Arrondit le réel `nombre` à l'entier le plus proche.

SeekEof(`Nom_Logique`**);** 129
Identique à **Eof** mais qui supprime les espaces et les caractères de tabulation avant d'effectuer le test.

SeekEoln(`Nom_logique`**);** 129
Identique à **Eoln** mais supprime les espaces et les caractères de tabulation avant d'effectuer le test.

Sin(`nombre`**);** 76
Retourne le sinus de `nombre` (radians). Le résultat est toujours un **réel**.

Sizeof(`nom`**);** 77
Retourne le nombre d'octets occupés en mémoire par la variable ou le type `nom`. Le résultat est de type **entier**.

Sqr(`nombre`); 76
Retourne le carré de `nombre`. Le résultat est du même type que l'argument.

Sqrt(`nombre`); 76
Retourne la racine carrée de `nombre`. Le résultat est un **réel**.

Succ(`valeur`); 64, 77
Retourne la valeur suivante de `valeur` : fonction inverse de la précédente. `Succ(dernier)` n'est pas défini.

Trunc(`nombre`); 76
Enlève la partie décimale du réel `nombre`. Le résultat est bien sûr un entier.

UpCase(`lettre`); 77
Retourne la majuscule correspondant à `lettre`.

Swap(`nombre`); 77
Echange les octets de poids fort et faible de `nombre`, qui doit être un **entier**.

ANNEXE 3

MESSAGES D'ERREURS

MESSAGES D'ERREURS DU COMPILATEUR

N°	Signification
01	Symbole **;** attendu.
02	Symbole **:** attendu.
03	Symbole **,** attendu.
04	Symbole **(** attendu.
05	Symbole **)** attendu.
06	Symbole **=** attendu.
07	Symbole **:=** attendu.
08	Symbole **[** attendu.
09	Symbole **]** attendu.
10	Symbole **.** attendu.
11	Symbole **..** attendu.
12	**Begin** attendu.
13	**Do** attendu.
14	**End** attendu.
15	**Of** attendu.
17	**Then** attendu.
18	**To** ou **DownTo** attendu.
20	Expression booléenne attendue.
21	Variable file attendue.
22	Constante entière attendue.
23	Expression entière attendue.
24	Variable entière attendue.
25	Constante réelle ou entière attendue.

26 Expression réelle ou entière attendue.
27 Variable entière ou réelle attendue.
28 Variable pointeur attendue.
29 Variable enregistrement attendue.
30 Type simple attendu.
31 Expression simple attendue.
32 Constante chaîne attendue.
33 Expression chaîne attendue.
34 Variable chaîne attendue.
35 Fichier texte attendu.
36 Identificateur de type attendu.
37 Fichier sans type attendu.
40 Etiquette non déclarée.
41 Identificateur inconnu ou erreur de syntaxe.
42 Une définition de type pointeur fait référence à un identificateur inconnu.
43 Identificateur ou label déja utilisé.
44 Types non compatibles.
45 Constante hors de l'intervalle.
46 Constante et sélecteur de cas incompatibles.
47 Le type de l'opérande n'est pas en accord avec l'opérateur.
48 Type de résultat incorrect.Un résultat de fonction doit nécessairement être de type scalaire, pointeur ou chaîne.
49 Longueur de chaîne incorrecte. La longueur doit être comprise entre 0 et 255.
50 Longueur de chaîne incompatible avec le type.
51 Le type n'admet pas les intervalles. Les types valides sont : scalaires, chaînes et pointeurs.
52 La borne inférieure est plus grande que la borne supérieure.
53 Mot réservé. Un mot réservé ne peut pas être utilisé comme identificateur.
54 Affectation illégale.
55 Constante chaîne dépassant la ligne.
56 Erreur dans une constante entière.
57 Erreur dans une constante réelle.
58 Caractère illégal dans l'identificateur.
60 Constante interdite ici.
61 Fichier et pointeur interdits ici.
62 Variable structurée interdite ici.
63 Fichier texte interdit ici.
64 Fichier texte et/ou non typé interdit ici.
65 Fichier sans type interdit ici.
66 Entrées/sorties interdites ici. Le type des variables n'est pas correct.
67 Les fichiers doivent être transmis par variable.
68 Les composantes d'un fichier ne doivent pas être des fichiers. On ne peut pas en Turbo-Pascal faire des fichiers de fichiers.
69 Ordre de champs incorrect.
70 Type de base d'un ensemble incorrect. Le type de base d'un ensemble doit être scalaire avec moins de 256 valeurs ou un intervalle avec des bornes comprises entre 0 et 255.
71 **Goto** interdit. Un **Goto** ne peut servir pour sortir d'une boucle **For**.

72 **Label** absent dans le bloc courant. La déclaration des **Goto** n'est pas globale en Turbo-Pascal.
73 Procédure **Forward** indéfinie. Le corps de la procédure déclarée en **Forward** est absent.
74 Erreur dans un **Inline**.
75 Utilisation incorrecte d'**Absolute**. Seul un identificateur peut apparaître avant les deux points (:) dans une déclaration **Absolute**. **Absolute** ne peut s'appliquer à un enregistrement.
76 Sous-programme Overlay interdit en **Forward**.
77 Overlay interdit en mode direct. Les sous-programmes déclarés en Overlay ne peuvent être utilisés que lorsque la compilation est effectuée sur disque.
90 Fichier non trouvé.
91 Fin de source incorrecte.
92 Incapable de créer un fichier Overlay.
93 Directive de compilation invalide.
97 Trop de **With** imbriqués. Si vous voulez utiliser plus de **With** imbriquables, vous devez utiliser la directive de compilation **W**.
98 Mémoire insuffisante pour les variables. Vous essayez probablement d'allouer plus de 64 Ko de mémoire pour les variables.
99 Mémoire insuffisante pour la compilation. Il n'y a plus de place en mémoire centrale pour loger le code objet. Pour y remédier, vous pouver couper votre programme en morceaux et utiliser des inclusions ou effectuer la compilation sur le disque.

MESSAGES D'ERREURS A L'EXECUTION

N°	Signification
01	Dépassement de capacité en virgule flottante.
02	Division par zéro.
03	Erreur d'argument avec la fonction **Sqrt** (il doit être positif ou nul).
04	Erreur d'argument avec la fonction **Ln** (il doit être positif).
10	Erreur sur la longueur d'une chaîne. Une concaténation donne une chaîne de longueur supérieure à 255 caractères ou vous essayez de convertir en caractères une chaîne de longueur supérieure à 1.
11	Erreur sur index de chaîne. L'index n'est pas dans [0..255].
90	Index hors des limites.
91	Scalaires ou sous-ensemble hors des limites.
92	Hors des limites des entiers. Les limites des entiers sont -32768 et +32768.
F0	Fichier Overlay non trouvé. Sans doute avez-vous changé de disquette entre temps.
FF	La pile de récursion et le **Heap** se sont rencontrés. Sans doute avez-vous exécuté un programme "trop" récursif ou avez-vous alloué trop de variables dynamiques.

MESSAGES D'ERREURS D'ENTREE/SORTIE

N°	Signification
01	Fichier inexistant.
02	Fichier non ouvert pour l'entrée.
03	Fichier non ouvert pour la sortie.
04	Fichier non ouvert.
10	Erreur dans un format numérique.
20	Opération interdite sur un organe logique.
21	Opération interdite en mode direct.
22	**Assign** interdit sur les fichiers standard.
90	Erreur dans la longueur des enregistrements.
91	**Seek** au delà de la fin du fichier.
99	Fin de fichier imprévue.
F0	Erreur d'écriture sur disque.
F1	Répertoire saturé.
F2	Dépassement de capacité du fichier.
F3	Trop de fichiers ouverts simultanément.
FF	Fichier disparu.

ANNEXE 4

DIRECTIVES DE COMPILATION

Les directives de compilation servent à donner des indications au compilateur sur la manière dont il doit opérer. Pour rendre active une directive, on la fait suivre du signe "**+**"; pour la désactiver, on la fait suivre du signe "**-**". Elles ont chacune un mode par défaut, indiqué ci-après. A l'exception des directives $B et $C, on peut changer plusieurs fois les directives de mode dans le cours du source. Le code généré sera affecté par la dernière directive rencontrée.

Rappelons qu'une directive de compilation doit apparaître entre deux accolades, précédée du signe **$,** et qu'**elle doit être seule sur la ligne**. Plusieurs directives peuvent être mises dans la même accolade, séparées par des virgules. Par exemple :

```
{$I-}
{$A-}
{$U+,V-,X+}
```

Les espaces sont interdits avant le $, ainsi qu'avant et après les signes "**+**" et "**-**".

Voici leur détail :

A : code absolu/récursif.
Mode par défaut : {$A+}.
En mode actif, le code généré ne peut pas être récursif (aucun sous-programme ne peut s'appeler lui-même). En mode passif, les sous-programmes récursifs sont autorisés. Ceci s'accompagne d'un accroissement du code et d'une exécution plus lente.

B : sélection du mode pour les entrées/sorties.
Mode par défaut :{$B+}.
En mode actif, .CON est assigné aux fichiers Input et Output standard. En mode passif, c'est TRM qui est selectionné. **C'est une directive globale qui ne peut pas être redéfinie à l'intérieur du programme.**

C : caractères Ctrl-C et Ctrl-S.
Mode par défaut : {$C+}.
En mode passif, les Ctrl-C lors des opérations de lecture et les Ctrl-S lors de celles d'écriture seront sans effet. En mode actif,Ctrl-C stoppe le programme, et Ctrl-S active/désactive les sorties à l'écran. En mode passif, les caractères ne sont pas interprétés. **C'est une directive globale qui ne peut pas être redéfinie à l'intérieur du programme.**

I : a) gestion des erreurs d'entrée/sortie.
Mode par défaut :{$I+}.
En mode passif, une erreur d'entrée/sortie ne provoquera pas l'arrêt du programme, mais le chargement du code d'erreur dans IOResult et l'interdiction des entrées/sorties avant consultation d'IOResult. En mode actif, une erreur d'entrée/sortie provoquera l'arrêt du programme et l'affichage du code et/ou du message correspondant.

b) inclusion.
La directive $I suivie d'un nom de fichier indique au compilateur d'inclure ce fichier lors de la phase de compilation. Si le nom du fichier ne possède pas d'extension , .PAS sera prise par défaut. Si l'accolade fermante est collée à l'extension de moins de trois lettres, elle est incluse dans celle-ci. Il faut donc toujours laisser une espace entre la dernière lettre du nom du fichier et l'accolade fermante pour éviter toute erreur possible.

R : vérification des index.
Mode par défaut : {$R-}.
En mode actif, tous les contrôles d'index (pour les tableaux) et les contrôles d'affectations pour les ensembles sont effectués. Cela réclame plus de temps d'exécution et sera donc supprimé dès que le programme sera au point. En mode passif, aucun contrôle n'est fait.

U : interruption par l'utilisateur.
Mode par défaut :{$U-}.
En mode actif, l'utilisateur peut interrompre le programme à tout moment en tapant Ctrl-C. En mode passif, Ctrl-C n'a pas d'effet.

V : vérification du type pour les paramètres variables.
Mode par défaut :{$V+}.
En mode actif, les longueurs des paramètres formels et réels doivent être identiques. En mode passif, on peut passer des paramètres réels de longueur différente de celle des paramètres formels (en particulier pour les chaînes).

W : imbrication des **With**.
Mode par défaut : {$W2}.
W contrôle le nombre de **With** pouvant être imbriqués. La lettre W doit être suivie d'un nombre compris entre 1 et 9.

X : optimisation des tableaux.
Mode par défaut : {$X+}.
En mode actif, la vitesse d'exécution est privilégiée. En mode passif, c'est la longueur du code qui l'est.

ANNEXE 5

TABLEAU D'ASSEMBLAGE

Codes des indicateurs d'état :

★	Drapeau affecté par l'opération
0	Drapeau mis à 0 par l'opération
1	Drapeau mis à 1 par l'opération
?	Drapeau affecté de manière non significative par l'opération
rien	Drapeau non affecté par l'opération

Abréviations utilisées pour les opérandes

op	Valeur sur 8 bits
ad	Valeur sur 16 bits

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
ADC A, (HL)	8E	7	★	★	★	★	0	★	indirect
ADC A, (IX+op)	DD8Eop	19	★	★	★	★	0	★	indexé
ADC A, (IY+op)	FD8Eop	19	★	★	★	★	0	★	"
ADC A,A	8F	4	★	★	★	★	0	★	registre
ADC A,B	88	4	★	★	★	★	0	★	"
ADC A,C	89	4	★	★	★	★	0	★	"
ADC A,D	8A	4	★	★	★	★	0	★	"
ADC A,E	8B	4	★	★	★	★	0	★	"
ADC A,H	8C	4	★	★	★	★	0	★	"
ADC A,L	8D	4	★	★	★	★	0	★	"
ADC A,op	CEop	7	★	★	★	★	0	★	immédiat
ADC HL,BC	ED4A	15	★	★	?	★	0	★	registre

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
ADC HL,DE	ED5A	15	★	★	?	★	0	★	registre
ADC HL,HL	ED6A	15	★	★	?	★	0	★	"
ADC HL,SP	ED7A	15	★	★	?	★	0	★	"
ADD A, (HL)	86	7	★	★	★	★	0	★	indirect
ADD A, (IX+op)	DD86op	19	★	★	★	★	0	★	indexé
ADD A, (IY+op)	FD86op	19	★	★	★	★	0	★	"
ADD A,A	87	4	★	★	★	★	0	★	registre
ADD A,B	80	4	★	★	★	★	0	★	"
ADD A,C	81	4	★	★	★	★	0	★	"
ADD A,D	82	4	★	★	★	★	0	★	"
ADD A,E	83	4	★	★	★	★	0	★	"
ADD A,H	84	4	★	★	★	★	0	★	"
ADD A,L	85	4	★	★	★	★	0	★	"
ADD A,op	C6op	7	★	★	★	★	0	★	immédiat
ADD HL,BC	ED4A	11			?		0	★	registre
ADD HL,DE	ED5A	11			?		0	★	"
ADD HL,HL	ED6A	11			?		0	★	"
ADD HL,SP	ED7A	11			?		0	★	"
ADD IX,BC	DD09	15			?		0	★	"
ADD IX,DE	DD19	15			?		0	★	"
ADD IX,IX	DD29	15			?		0	★	"
ADD IX,SP	DD39	15			?		0	★	"
ADD IY,BC	FD09	15			?		0	★	"
ADD IY,DE	FD19	15			?		0	★	"
ADD IY,IY	FD29	15			?		0	★	"
ADD IY,SP	FD39	15			?		0	★	"
AND (HL)	A6	7	★	★	1	★	0	0	indirect
AND (IX+op)	DDA6op	19	★	★	1	★	0	0	indexé
AND (IY+op)	FDA6op	19	★	★	1	★	0	0	"
AND A	A7	4	★	★	1	★	0	0	registre
AND B	A0	4	★	★	1	★	0	0	"
AND C	A1	4	★	★	1	★	0	0	"
AND D	A2	4	★	★	1	★	0	0	"
AND E	A3	4	★	★	1	★	0	0	"
AND H	A4	4	★	★	1	★	0	0	"
AND L	A5	4	★	★	1	★	0	0	"
AND op	E6op	7	★	★	1	★	0	0	immédiat
BIT 0,(HL)	CB46	12	?	★	1	?	0		indirect
BIT 0, (IX+op)	DDCBop46	20	?	★	1	?	0		indexé
BIT 0, (IY+op)	FDCBop46	20	?	★	1	?	0		"
BIT 0,A	CB47	8	?	★	1	?	0		registre
BIT 0,B	CB40	8	?	★	1	?	0		"
BIT 0,C	CB41	8	?	★	1	?	0		"
BIT 0,D	CB42	8	?	★	1	?	0		"

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
BIT 0,E	CB43	8	?	★	1	?	0		registre
BIT 0,H	CB44	8	?	★	1	?	0		"
BIT 0,L	CB45	8	?	★	1	?	0		"
BIT 1, (HL)	CB4E	12	?	★	1	?	0		indirect
BIT 1, (IX+op)	DDCBop4E	20	?	★	1	?	0		indexé
BIT 1, (IY+op)	FDCBop4E	20	?	★	1	?	0		"
BIT 1,A	CB4F	8	?	★	1	?	0		registre
BIT 1,B	CB48	8	?	★	1	?	0		"
BIT 1,C	CB49	8	?	★	1	?	0		"
BIT 1,D	CB4A	8	?	★	1	?	0		"
BIT 1,E	CB4B	8	?	★	1	?	0		"
BIT 1,H	CB4C	8	?	★	1	?	0		"
BIT 1,L	CB4D	8	?	★	1	?	0		"
BIT 2, (HL)	CB56	12	?	★	1	?	0		indirect
BIT 2, (IX+op)	DDCBop56	20	?	★	1	?	0		indexé
BIT 2, (IY+op)	FDCBop56	20	?	★	1	?	0		"
BIT 2,A	CB57	8	?	★	1	?	0		registre
BIT 2,B	CB50	8	?	★	1	?	0		"
BIT 2,C	CB51	8	?	★	1	?	0		"
BIT 2,D	CB52	8	?	★	1	?	0		"
BIT 2,E	CB53	8	?	★	1	?	0		"
BIT 2,H	CB54	8	?	★	1	?	0		"
BIT 2,L	CB55	8	?	★	1	?	0		"
BIT 3, (HL)	CB5E	12	?	★	1	?	0		indirect
BIT 3, (IX+op)	DDCBop5E	20	?	★	1	?	0		indexé
BIT 3, (IY+op)	FDCBop5E	20	?	★	1	?	0		"
BIT 3,A	CB5F	8	?	★	1	?	0		registre
BIT 3,B	CB58	8	?	★	1	?	0		"
BIT 3,C	CB59	8	?	★	1	?	0		"
BIT 3,D	CB5A	8	?	★	1	?	0		"
BIT 3,E	CB5B	8	?	★	1	?	0		"
BIT 3,H	CB5C	8	?	★	1	?	0		"
BIT 3,L	CB5D	8	?	★	1	?	0		"
BIT 4, (HL)	CB66	12	?	★	1	?	0		indirect
BIT 4, (IX+op)	DDCBop66	20	?	★	1	?	0		indexé
BIT 4, (IY+op)	FDCBop66	20	?	★	1	?	0		"
BIT 4,A	CB67	8	?	★	1	?	0		registre
BIT 4,B	CB60	8	?	★	1	?	0		"
BIT 4,C	CB61	8	?	★	1	?	0		"
BIT 4,D	CB62	8	?	★	1	?	0		"
BIT 4,E	CB63	8	?	★	1	?	0		"
BIT 4,H	CB64	8	?	★	1	?	0		"
BIT 4,L	CB65	8	?	★	1	?	0		"
BIT 5, (HL)	CB6E	12	?	★	1	?	0		indirect

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
BIT 5, (IX+op)	DDCBop6E	20	?	★	1	?	0		indexé
BIT 5, (IY+op)	FDCBop6E	20	?	★	1	?	0		"
BIT 5,A	CB6F	8	?	★	1	?	0		registre
BIT 5,B	CB68	8	?	★	1	?	0		"
BIT 5,C	CB69	8	?	★	1	?	0		"
BIT 5,D	CB6A	8	?	★	1	?	0		"
BIT 5,E	CB6B	8	?	★	1	?	0		"
BIT 5,H	CB6C	8	?	★	1	?	0		"
BIT 5,L	CB6D	8	?	★	1	?	0		"
BIT 6, (HL)	CB76	12	?	★	1	?	0		indirect
BIT 6, (IX+op)	DDCBop76	20	?	★	1	?	0		indexé
BIT 6, (IY+op)	FDCBop76	20	?	★	1	?	0		"
BIT 6,A	CB77	8	?	★	1	?	0		registre
BIT 6,B	CB70	8	?	★	1	?	0		"
BIT 6,C	CB71	8	?	★	1	?	0		"
BIT 6,D	CB72	8	?	★	1	?	0		"
BIT 6,E	CB73	8	?	★	1	?	0		"
BIT 6,H	CB74	8	?	★	1	?	0		"
BIT 6,L	CB75	8	?	★	1	?	0		"
BIT 7, (HL)	CB7E	12	?	★	1	?	0		indirect
BIT 7, (IX+op)	DDCBop7E	20	?	★	1	?	0		indexé
BIT 7, (IY+op)	FDCBop7E	20	?	★	1	?	0		"
BIT 7,A	CB7F	8	?	★	1	?	0		registre
BIT 7,B	CB78	8	?	★	1	?	0		"
BIT 7,C	CB79	8	?	★	1	?	0		"
BIT 7,D	CB7A	8	?	★	1	?	0		"
BIT 7,E	CB7B	8	?	★	1	?	0		"
BIT 7,H	CB7C	8	?	★	1	?	0		"
BIT 7,L	CB7D	8	?	★	1	?	0		"
CALL C,ad	DCad	17/10							immédiat
CALL M,ad	FCad	17/10							"
CALL NC,ad	D4ad	17/10							"
CALL NZ,ad	C4ad	17/10							"
CALL P,ad	F4ad	17/10							"
CALL PE,ad	ECad	17/10							"
CALL PO,ad	E4ad	17/10							"
CALL Z,ad	CCad	17/10							"
CALL ad	CDad	17							"
CCF	3F	4			?		0	★	implicite
CP (HL)	BE	7	★	★	★	★	1	★	indirect
CP (IX+op)	DDBEop	19	★	★	★	★	1	★	indexé
CP (IY+op)	FDBEop	19	★	★	★	★	1	★	"
CP A	BF	4	★	★	★	★	1	★	registre
CP B	B8	4	★	★	★	★	1	★	"

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
CP C	B9	4	★	★	★	★	1	★	registre
CP D	BA	4	★	★	★	★	1	★	"
CP E	BB	4	★	★	★	★	1	★	"
CP H	BC	4	★	★	★	★	1	★	"
CP L	BD	4	★	★	★	★	1	★	"
CP op	FEop	7	★	★	★	★	1	★	immédiat
CPD	EDA9	16	★	★	★	★	1		implicite
CPDR	EDB9	16/21	★	★	★	★	1		"
CPI	EDA1	16	★	★	★	★	1		"
CPIR	EDB1	16/21	★	★	★	★	1		"
CPL	2F	4			1		1		"
DAA	27	4	★	★	★	★		★	"
DEC (HL)	35	11	★	★	★	★	1	★	indirect
DEC (IX+op)	DD35op	23	★	★	★	★	1	★	indexé
DEC (IY+op)	FD35op	23	★	★	★	★	1	★	"
DEC A	3D	4	★	★	★	★	1	★	registre
DEC B	05	4	★	★	★	★	1	★	"
DEC C	0D	4	★	★	★	★	1	★	"
DEC D	15	4	★	★	★	★	1	★	"
DEC E	1D	4	★	★	★	★	1	★	"
DEC H	25	4	★	★	★	★	1	★	"
DEC L	2D	4	★	★	★	★	1	★	"
DEC BC	OB	6							"
DEC DE	1B	6							"
DEC HL	2B	6							"
DEC SP	3B	6							"
DEC IX	DD2B	10							"
DEC IY	FD2B	10							"
DI	F3	4							implicite
DJNZ op	10op	8/13							relatif
EI	FB	4							implicite
EX (SP),HL	E3	19							indirect
EX (SP),IX	DDE3	23							"
EX (SP),IY	FDE3	23							"
EX AF,AF'	08	4							implicite
EX DE,HL	EB	4							"
EXX	D9	4							"
HALT	76	4							"
IM0	ED46	4							"
IM1	ED56	4							"
IM2	ED5E	4							"
IN A,(C)	ED78	12	★	★	★	★	0		indirect
IN B,(C)	ED40	12	★	★	★	★	0		"
IN C,(C)	ED48	12	★	★	★	★	0		"

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
IN D,(C)	ED50	12	★	★	★	★	0		indirect
IN E,(C;	ED58	12	★	★	★	★	0		"
IN H,(C)	ED60	12	★	★	★	★	0		"
IN L,(C)	ED68	12	★	★	★	★	0		"
IN A,(op)	DBop	11							"
INC (HL)	34	11	★	★	★	★	0		indirect
INC (IX+op)	DD34op	23	★	★	★	★	0		indexé
INC (IY+op)	FD34op	23	★	★	★	★	0		"
INC A	3C	4	★	★	★	★	0		registre
INC B	04	4	★	★	★	★	0		"
INC C	0C	4	★	★	★	★	0		"
INC D	14	4	★	★	★	★	0		"
INC E	1C	4	★	★	★	★	0		"
INC H	24	4	★	★	★	★	0		"
INC L	2C	4	★	★	★	★	0		"
INC BC	03	6							"
INC DE	13	6							"
INC HL	23	6							"
INC SP	33	6							"
INC IX	DD23	10							"
INC IY	FD23	10							"
IND	EDAA	16	?	★	?	?	1		implicite
INDR	EDBA	16/21	?	1	?	?	1		"
INI	EDA2	16	?	★	?	?	1		"
INIR	EDB2	16/21	?	1	?	?	1		"
JP ad	C3ad	10							immédiat
JP (HL)	E9	4							indirect
JP (IX)	DDE9	8							"
JP (IY)	FDE9	8							"
JP C,ad	DAad	10							immédiat
JP M,ad	FAad	10							"
JP NC,ad	D2ad	10							"
JP NZ,ad	C2ad	10							"
JP P,ad	F2ad	10							"
JP PE,ad	EAad	10							"
JP PO,ad	E2ad	10							"
JP Z,ad	CAad	12/7							"
JR C,op	38op	12/7							relatif
JR NC,op	30op	12/7							"
JR NZ,op	20op	12/7							"
JR Z,op	28op	12/7							"
JR op	18op	12							"
LD (BC),A	02	7							indirect
LD (DE),A	12	7							"

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
LD (HL),A	77	7							indirect
LD (HL),B	70	7							"
LD (HL),C	71	7							"
LD (HL),D	72	7							"
LD (HL),E	73	7							"
LD (HL),H	74	7							"
LD (HL),L	75	7							"
LD (HL),op	36op	10							imm./ind.
LD (IX+op),A	DD77op	19							indexé
LD (IX+op),B	DD70op	19							"
LD (IX+op),C	DD71op	19							"
LD (IX+op),D	DD72op	19							"
LD (IX+op),E	DD73op	19							"
LD (IX+op),H	DD74op	19							"
LD (IX+op),L	DD75op	19							"
LD (IX+op),op'	DD36opop'	19							ind./imm.
LD (IY+op),A	FD77op	19							indexé
LD (IY+op),B	FD70op	19							"
LD (IY+op),C	FD71op	19							"
LD (IY+op),D	FD72op	19							"
LD (IY+op),E	FD73op	19							"
LD (IY+op),H	FD74op	19							"
LD (IY+op),L	FD75op	19							"
LD (IY+op),op'	FD36opop'	19							ind./imm.
LD (ad),A	32ad	13							direct
LD (ad),BC	ED43ad	20							"
LD (ad),DE	ED53ad	20							"
LD (ad),HL	22ad	16							"
LD (ad),IX	DD22ad	20							"
LD (ad),IY	FD22ad	20							"
LD (ad),SP	ED73ad	20							"
LD A, (BC)	0A	7							indirect
LD A, (DE)	1A	7							"
LD A, (HL)	7E	7							"
LD A, (IX+op)	DD7Eop	19							indexé
LD A, (IY+op)	FD7Eop	19							"
LD A, (ad)	3Aad	13							direct
LD A,A	7F	4							registre
LD A,B	78	4							"
LD A,C	79	4							"
LD A,D	7A	4							"
LD A,E	7B	4							"
LD A,H	7C	4							"
LD A,L	7D	4							"

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
LD A,op	3Eop	7							immédiat
LD A,I	ED57	9	★	★	0	★	0		implicite
LD A,R	ED5F	9	★	★	0	★	0		"
LD B, (HL)	46	7							indirect
LD B, (IX+op)	DD46op	19							indexé
LD B, (IY+op)	FD46op	19							"
LD B,A	47	4							registre
LD B,B	40	4							"
LD B,C	41	4							"
LD B,D	42	4							"
LD B,E	43	4							"
LD B,H	44	4							"
LD B,L	45	4							"
LD B,op	06op	7							immédiat
LD C,(HL)	4E	7							indirect
LD C, (IX+op)	DD4Eop	19							indexé
LD C, (IY+op)	FD4Eop	19							"
LD C,A	4F	4							registre
LD C,B	48	4							"
LD C,C	49	4							"
LD C,D	4A	4							"
LD C,E	4B	4							"
LD C,H	4C	4							"
LD C,L	4D	4							"
LD C,op	0Eop	7							immédiat
LD D, (HL)	56	7							indirect
LD D, (IX+op)	DD56op	19							indexé
LD D, (IY+op)	FD56op	19							"
LD D,A	57	4							registre
LD D,B	50	4							"
LD D,C	51	4							"
LD D,D	52	4							"
LD D,E	53	4							"
LD D,H	54	4							"
LD D,L	55	4							"
LD D,op	16op	7							immédiat
LD E, (HL)	5E	7							indirect
LD E, (IX+op)	DD5Eop	19							indexé
LD E, (IY+op)	FD5Eop	19							"
LD E,A	5F	4							registre
LD E,B	58	4							"
LD E,C	59	4							"
LD E,D	5A	4							"
LD E,E	5B	4							"

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
LD E,H	5C	4							registre
LD E,L	5D	4							"
LD E,op	1Eop	7							immédiat
LD H, (HL)	66	7							indirect
LD H, (IX+op)	DD66op	19							indexé
LD H, (IY+op)	FD66op	19							"
LD H,A	67	4							registre
LD H,B	60	4							"
LD H,C	61	4							"
LD H,D	62	4							"
LD H,E	63	4							"
LD H,H	64	4							"
LD H,L	65	4							"
LD H,op	26op	7							immédiat
LD L, (HL)	6E	7							indirect
LD L, (IX+op)	DD6Eop	19							indexé
LD L, (IY+op)	FD6Eop	19							"
LD L,A	6F	4							registre
LD L,B	68	4							"
LD L,C	69	4							"
LD L,D	6A	4							"
LD L,E	6B	4							"
LD L,H	6C	4							"
LD L,L	6D	4							"
LD L,op	2Eop	7							immédiat
LD I,A	ED47	9							implicite
LD R,A	ED4F	9							"
LD BC,(ad)	ED4Bad	20							direct
LD BC,ad	01ad	10							immédiat
LD DE, (ad)	ED5Bad	20							direct
LD DE,ad	11ad	10							immédiat
LD HL, (ad)	2Aad	16							direct
LD HL,ad	21ad	10							immédiat
LD IX, (ad)	DD2Aad	20							direct
LD IX,ad	DD21ad	14							immédiat
LD IY,(ad)	FD2Aad	20							direct
LD IY,ad	FD21ad	14							immédiat
LD SP,(ad)	ED7Bad	20							direct
LD SP,HL	F9	6							implicite
LD SP,IX	DDF9	10							"
LD SP,IY	FDF9	10							"
LD SP,ad	31ad	10							immédiat
LDD	EDA8	16			0	*	0		indirect
LDDR	EDB8	16/21			0	0	0		"

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
LDI	EDA0	16			0	★	0		indirect
LDIR	EDB0	16/21			0	0	0		"
NEG	ED44	8	★	★	★	★	1	★	implicite
NOP	00	4							"
OR (HL)	B6	7	★	★	0	★	0	0	indirect
OR (IX+op)	DDB6op	19	★	★	0	★	0	0	indexé
OR (IY+op)	FDB6op	19	★	★	0	★	0	0	"
OR A	B7	4	★	★	0	★	0	0	registre
OR B	B0	4	★	★	0	★	0	0	"
OR C	B1	4	★	★	0	★	0	0	"
OR D	B2	4	★	★	0	★	0	0	"
OR E	B3	4	★	★	0	★	0	0	"
OR H	B4	4	★	★	0	★	0	0	"
OR L	B5	4	★	★	0	★	0	0	"
OR op	F6op	7	★	★	0	★	0	0	immédiat
OTDR	EDBB	16/21	?	1	?	?	1		indirect
OTIR	EDB3	16/21	?	1	?	?	1		"
OUT (C),A	ED79	12							"
OUT (C),B	ED41	12							"
OUT (C),C	ED49	12							"
OUT (C),D	ED51	12							"
OUT (C),E	ED59	12							"
OUT (C),H	ED61	12							"
OUT (C),L	ED69	12							"
OUT (op),A	D3op	11							"
OUTD	EDAB	16	?	★	?	?	1		"
OUTI	EDA3	16	?	★	?	?	1		"
POP AF	F1	10							"
POP BC	C1	10							"
POP DE	D1	10							"
POP HL	E1	10							"
POP IX	DDE1	14							"
POP IY	FDE1	14							"
PUSH AF	F5	11							"
PUSH BC	C5	11							"
PUSH DE	D5	11							"
PUSH HL	E5	11							"
PUSH IX	DDE5	15							"
PUSH IY	FDE5	15							"
RES 0,(HL)	CB86	15	?	★	1	?	0		indirect
RES 0,(IX+op)	DDCBop86	23	?	★	1	?	0		indexé
RES 0,(IY+op)	FDCBop86	23	?	★	1	?	0		"
RES 0,A	CB87	8	?	★	1	?	0		registre
RES 0,B	CB80	8	?	★	1	?	0		"

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
RES 0,C	CB81	8	?	★	1	?	0		registre
RES 0,D	CB82	8	?	★	1	?	0		"
RES 0,E	CB83	8	?	★	1	?	0		"
RES 0,H	CB84	8	?	★	1	?	0		"
RES 0,L	CB85	8	?	★	1	?	0		"
RES 1,(HL)	CB8E	15	?	★	1	?	0		indirect
RES 1,(IX+op)	DDCBop8E	23	?	★	1	?	0		indexé
RES 1,(IY+op)	FDCBop8E	23	?	★	1	?	0		"
RES 1,A	CB8F	8	?	★	1	?	0		registre
RES 1,B	CB88	8	?	★	1	?	0		"
RES 1,C	CB89	8	?	★	1	?	0		"
RES 1,D	CB8A	8	?	★	1	?	0		"
RES 1,E	CB8B	8	?	★	1	?	0		"
RES 1,H	CB8C	8	?	★	1	?	0		"
RES 1,L	CB8D	8	?	★	1	?	0		"
RES 2, (HL)	CB96	15	?	★	1	?	0		indirect
RES 2, (IX+op)	DDCBop96	23	?	★	1	?	0		indexé
RES 2, (IY+op)	FDCBop96	23	?	★	1	?	0		"
RES 2,A	CB97	8	?	★	1	?	0		registre
RES 2,B	CB90	8	?	★	1	?	0		"
RES 2,C	CB91	8	?	★	1	?	0		"
RES 2,D	CB92	8	?	★	1	?	0		"
RES 2,E	CB93	8	?	★	1	?	0		"
RES 2,H	CB94	8	?	★	1	?	0		"
RES 2,L	CB95	8	?	★	1	?	0		"
RES 3, (HL)	CB9E	15	?	★	1	?	0		indirect
RES 3, (IX+op)	DDCBop9E	23	?	★	1	?	0		indexé
RES 3, (IY+op)	FDCBop9E	23	?	★	1	?	0		"
RES 3,A	CB9F	8	?	★	1	?	0		registre
RES 3,B	CB98	8	?	★	1	?	0		"
RES 3,C	CB99	8	?	★	1	?	0		"
RES 3,D	CB9A	8	?	★	1	?	0		"
RES 3,E	CB9B	8	?	★	1	?	0		"
RES 3,H	CB9C	8	?	★	1	?	0		"
RES 3,L	CB9D	8	?	★	1	?	0		"
RES 4, (HL)	CBA6	15	?	★	1	?	0		indirect
RES 4, (IX+op)	DDCBopA6	23	?	★	1	?	0		indexé
RES 4, (IY+op)	FDCBopA6	23	?	★	1	?	0		"
RES 4,A	CBA7	8	?	★	1	?	0		registre
RES 4,B	CBA0	8	?	★	1	?	0		"
RES 4,C	CBA1	8	?	★	1	?	0		"
RES 4,D	CBA2	8	?	★	1	?	0		"
RES 4,E	CBA3	8	?	★	1	?	0		"
RES 4,H	CBA4	8	?	★	1	?	0		"

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
RES 4,L	CBA5	8	?	★	1	?	0		registre
RES 5, (HL)	CBAE	15	?	★	1	?	0		indirect
RES 5, (IX+op)	DDCBopAE	23	?	★	1	?	0		indexé
RES 5, (IY+op)	FDCBopAE	23	?	★	1	?	0		"
RES 5,A	CBAF	8	?	★	1	?	0		registre
RES 5,B	CBA8	8	?	★	1	?	0		"
RES 5,C	CBA9	8	?	★	1	?	0		"
RES 5,D	CBAA	8	?	★	1	?	0		"
RES 5,E	CBAB	8	?	★	1	?	0		"
RES 5,H	CBAC	8	?	★	1	?	0		"
RES 5,L	CBAD	8	?	★	1	?	0		"
RES 6, (HL)	CBB6	15	?	★	1	?	0		indirect
RES 6, (IX+op)	DDCBopB6	23	?	★	1	?	0		indexé
RES 6, (IY+op)	FDCBopB6	23	?	★	1	?	0		"
RES 6,A	CBB7	8	?	★	1	?	0		registre
RES 6,B	CBB0	8	?	★	1	?	0		"
RES 6,C	CBB1	8	?	★	1	?	0		"
RES 6,D	CBB2	8	?	★	1	?	0		"
RES 6,E	CBB3	8	?	★	1	?	0		"
RES 6,H	CBB4	8	?	★	1	?	0		"
RES 6,L	CBB5	8	?	★	1	?	0		"
RES 7, (HL)	CBBE	15	?	★	1	?	0		indirect
RES 7, (IX+op)	DDCBopBE	23	?	★	1	?	0		indexé
RES 7, (IY+op)	FDCBopBE	23	?	★	1	?	0		"
RES 7,A	CBBF	8	?	★	1	?	0		registre
RES 7,B	CBB8	8	?	★	1	?	0		"
RES 7,C	CBB9	8	?	★	1	?	0		"
RES 7,D	CBBA	8	?	★	1	?	0		"
RES 7,E	CBBB	8	?	★	1	?	0		"
RES 7,H	CBBC	8	?	★	1	?	0		"
RES 7,L	CBBD	8	?	★	1	?	0		"
RET	C9	10							indirect
RET C	D8	5/11							"
RET M	F8	5/11							"
RET NC	D0	5/11							"
RET NZ	C0	5/11							"
RET P	F0	5/11							"
RET PE	E8	5/11							"
RET PO	E0	5/11							"
RET Z	C8	5/11							"
RETI	ED4D	14							"
RETN	ED45	14							"
RL (HL)	CB16	15	★	★	0	★	0	★	indirect
RL (IX+op)	DDCBop16	23	★	★	0	★	0	★	indexé

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
RL (IY+op)	FDCBop16	23	★	★	0	★	0	★	indexé
RL A	CB17	8	★	★	0	★	0	★	registre
RL B	CB10	8	★	★	0	★	0	★	"
RL C	CB11	8	★	★	0	★	0	★	"
RL D	CB12	8	★	★	0	★	0	★	"
RL E	CB13	8	★	★	0	★	0	★	"
RL H	CB14	8	★	★	0	★	0	★	"
RL L	CB15	8	★	★	0	★	0	★	"
RLA	17	4			0		0	★	implicite
RLC (HL)	CB06	15	★	★	0	★	0	★	indirect
RLC (IX+op)	DDCBop06	23	★	★	0	★	0	★	indexé
RLC (IY+op)	FDCBop06	23	★	★	0	★	0	★	"
RLC A	CB07	8	★	★	0	★	0	★	registre
RLC B	CB00	8	★	★	0	★	0	★	"
RLC C	CB01	8	★	★	0	★	0	★	"
RLC D	CB02	8	★	★	0	★	0	★	"
RLC E	CB03	8	★	★	0	★	0	★	"
RLC H	CB04	8	★	★	0	★	0	★	"
RLC L	CB05	8	★	★	0	★	0	★	"
RLCA	07	4			0		0	★	implicite
RLD	ED6F	18	★	★	0	★	0		"
RR (HL)	CB1E	15	★	★	0	★	0	★	indirect
RR (IX+op)	DDCBop1E	23	★	★	0	★	0	★	indexé
RR (IY+op)	FDCBop1E	23	★	★	0	★	0	★	"
RR A	CB1F	8	★	★	0	★	0	★	registre
RR B	CB18	8	★	★	0	★	0	★	"
RR C	CB19	8	★	★	0	★	0	★	"
RR D	CB1A	8	★	★	0	★	0	★	"
RR E	CB1B	8	★	★	0	★	0	★	"
RR H	CB1C	8	★	★	0	★	0	★	"
RR L	CB1D	8	★	★	0	★	0	★	"
RRA	1F	4			0		0	★	implicite
RRC (HL)	CB0E	15	★	★	0	★	0	★	indirect
RRC (IX+op)	DDCBop0E	23	★	★	0	★	0	★	indexé
RRC (IY+op)	FDCBop0E	23	★	★	0	★	0	★	"
RRC A	CB0F	8	★	★	0	★	0	★	registre
RRC B	CB08	8	★	★	0	★	0	★	"
RRC C	CB09	8	★	★	0	★	0	★	"
RRC D	CB0A	8	★	★	0	★	0	★	"
RRC E	CB0B	8	★	★	0	★	0	★	"
RRC H	CB0C	8	★	★	0	★	0	★	"
RRC L	CB0D	8	★	★	0	★	0	★	"
RRCA	0F	4			0	★	0	★	implicite
RRD	ED67	18	★	★	0	★	0		indirect

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
RST 00h	C7	11							indirect
RST 08h	CF	11							"
RST 10h	D7	11							"
RST 18h	DF	11							"
RST 20h	E7	11							"
RST 28h	EF	11							"
RST 30h	F7	11							"
RTS 38h	FF	11							"
SBC A,op	DEop	7	*	*	*	*	1	*	immédiat
SBC A, (HL)	9E	7	*	*	*	*	1	*	indirect
SBC A, (IX+op)	DD9E	19	*	*	*	*	1	*	indexé
SBC A, (IY+op)	FD9E	19	*	*	*	*	1	*	"
SBC A,A	9F	4	*	*	*	*	1	*	registre
SBC A,B	98	4	*	*	*	*	1	*	"
SBC A,C	99	4	*	*	*	*	1	*	"
SBC A,D	9A	4	*	*	*	*	1	*	"
SBC A,E	9B	4	*	*	*	*	1	*	"
SBC A,H	9C	4	*	*	*	*	1	*	"
SBC A,L	9D	4	*	*	*	*	1	*	"
SBC HL,BC	ED42	15	*	*	?	*	1	*	"
SBC HL,DE	ED52	15	*	*	?	*	1	*	"
SBC HL,HL	ED62	15	*	*	?	*	1	*	"
SBC HL,SP	ED72	15	*	*	?	*	1	*	"
SCF	37	4			0		0	1	implicite
SET 0, (HL)	CBC6	15	?	*	1	?	0		indirect
SET 0, (IX+op)	DDCBopC6	23	?	*	1	?	0		indexé
SET 0, (IY+op)	FDCBopC6	23	?	*	1	?	0		"
SET 0,A	CBC7	8	?	*	1	?	0		registre
SET 0,B	CBC0	8	?	*	1	?	0		"
SET 0,C	CBC1	8	?	*	1	?	0		"
SET 0,D	CBC2	8	?	*	1	?	0		"
SET 0,E	CBC3	8	?	*	1	?	0		"
SET 0,H	CBC4	8	?	*	1	?	0		"
SET 0,L	CBC5	8	?	*	1	?	0		"
SET 1, (HL)	CBCE	15	?	*	1	?	0		indirect
SET 1, (IX+op)	DDCBopCE	23	?	*	1	?	0		indexé
SET 1, (IY+op)	FDCBopCE	23	?	*	1	?	0		"
SET 1,A	CBCF	8	?	*	1	?	0		registre
SET 1,B	CBC8	8	?	*	1	?	0		"
SET 1,C	CBC9	8	?	*	1	?	0		"
SET 1,D	CBCA	8	?	*	1	?	0		"
SET 1,E	CBCB	8	?	*	1	?	0		"
SET 1,H	CBCC	8	?	*	1	?	0		"
SET 1,L	CBCD	8	?	*	1	?	0		"

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
SET 2, (HL)	CBD6	15	?	★	1	?	0		indirect
SET 2, (IX+op)	DDCBopD6	23	?	★	1	?	0		indexé
SET 2, (IY+op)	FDCBopD6	23	?	★	1	?	0		"
SET 2,A	CBD7	8	?	★	1	?	0		registre
SET 2,B	CBD0	8	?	★	1	?	0		"
SET 2,C	CBD1	8	?	★	1	?	0		"
SET 2,D	CBD2	8	?	★	1	?	0		"
SET 2,E	CBD3	8	?	★	1	?	0		"
SET 2,H	CBD4	8	?	★	1	?	0		"
SET 2,L	CBD5	8	?	★	1	?	0		"
SET 3, (HL)	CBDE	15	?	★	1	?	0		indirect
SET 3, (IX+op)	DDCBopDE	23	?	★	1	?	0		indexé
SET 3, (IY+op)	FDCBopDE	23	?	★	1	?	0		"
SET 3,A	CBDF	8	?	★	1	?	0		registre
SET 3,B	CBD8	8	?	★	1	?	0		"
SET 3,C	CBD9	8	?	★	1	?	0		"
SET 3,D	CBDA	8	?	★	1	?	0		"
SET 3,E	CBDB	8	?	★	1	?	0		"
SET 3,H	CBDC	8	?	★	1	?	0		"
SET 3,L	CBDD	8	?	★	★	?	0		"
SET 4, (HL)	CBE6	15	?	★	1	?	0		indirect
SET 4, (IX+op)	DDCBopE6	23	?	★	1	?	0		indexé
SET 4, (IY+op)	FDCBopE6	23	?	★	1	?	0		"
SET 4,A	CBE7	8	?	★	1	?	0		registre
SET 4,B	CBE0	8	?	★	1	?	0		"
SET 4,C	CBE1	8	?	★	1	?	0		"
SET 4,D	CBE2	8	?	★	1	?	0		"
SET 4,E	CBE3	8	?	★	1	?	0		"
SET 4,H	CBE4	8	?	★	1	?	0		"
SET 4,L	CBE5	8	?	★	1	?	0		"
SET 5, (HL)	CBEE	15	?	★	1	?	0		indirect
SET 5, (IX+op)	DDCBopEE	23	?	★	1	?	0		indexé
SET 5, (IY+op)	FDCBopEE	23	?	★	1	?	0		"
SET 5,A	CBEF	8	?	★	1	?	0		registre
SET 5,B	CBE8	8	?	★	1	?	0		"
SET 5,C	CBE9	8	?	★	1	?	0		"
SET 5,D	CBEA	8	?	★	1	?	0		"
SET 5,E	CBEB	8	?	★	1	?	0		"
SET 5,H	CBEC	8	?	★	1	?	0		"
SET 5,L	CBED	8	?	★	1	?	0		"
SET 6, (HL)	CBF6	15	?	★	1	?	0		indirect
SET 6, (IX+op)	DDCBopF6	23	?	★	1	?	0		indexé
SET 6, (IY+op)	FDCBopF6	23	?	★	1	?	0		"
SET 6,A	CBF7	8	?	★	1	?	0		registre

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
SET 6,B	CBF0	8	?	★	1	?	0		registre
SET 6,C	CBF1	8	?	★	1	?	0		"
SET 6,D	CBF2	8	?	★	1	?	0		"
SET 6,E	CBF3	8	?	★	1	?	0		"
SET 6,H	CBF4	8	?	★	1	?	0		"
SET 6,L	CBF5	8	?	★	1	?	0		"
SET 7, (HL)	CBFE	15	?	★	1	?	0		indirect
SET 7, (IX+op)	DDCBopFE	23	?	★	1	?	0		indexé
SET 7, (IY+op)	FDCBopFE	23	?	★	1	?	0		"
SET 7,A	CBFF	8	?	★	1	?	0		registre
SET 7,B	CBF8	8	?	★	1	?	0		"
SET 7,C	CBF9	8	?	★	1	?	0		"
SET 7,D	CBFA	8	?	★	1	?	0		"
SET 7,E	CBFB	8	?	★	1	?	0		"
SET 7,H	CBFC	8	?	★	1	?	0		"
SET 7,L	CBFD	8	?	★	1	?	0		"
SLA (HL)	CB26	15	★	★	0	★	0	★	indirect
SLA (IX+op)	DDCBop26	23	★	★	0	★	0	★	indexé
SLA (IY+op)	FDCBop26	23	★	★	0	★	0	★	"
SLA A	CB27	8	★	★	0	★	0	★	registre
SLA B	CB20	8	★	★	0	★	0	★	"
SLA C	CB21	8	★	★	0	★	0	★	"
SLA D	CB22	8	★	★	0	★	0	★	"
SLA E	CB23	8	★	★	0	★	0	★	"
SLA H	CB24	8	★	★	0	★	0	★	"
SLA L	CB25	8	★	★	0	★	0	★	"
SRA (HL)	CB2E	15	★	★	0	★	0	★	indirect
SRA (IX+op)	DDCBop2E	23	★	★	0	★	0	★	indexé
SRA (IY+op)	FDCBop2E	23	★	★	0	★	0	★	"
SRA A	CB2F	8	★	★	0	★	0	★	registre
SRA B	CB28	8	★	★	0	★	0	★	"
SRA C	CB29	8	★	★	0	★	0	★	"
SRA D	CB2A	8	★	★	0	★	0	★	"
SRA E	CB2B	8	★	★	0	★	0	★	"
SRA H	CB2C	8	★	★	0	★	0	★	"
SRA L	CB2D	8	★	★	0	★	0	★	"
SRL (HL)	CB3E	15	★	★	0	★	0	★	indirect
SRL (IX+op)	DDCBop3E	23	★	★	0	★	0	★	indexé
SRL (IY+op)	FDCBop3E	23	★	★	0	★	0	★	"
SRL A	CB3F	8	★	★	0	★	0	★	registre
SRL B	CB38	8	★	★	0	★	0	★	"
SRL C	CB39	8	★	★	0	★	0	★	"
SRL D	CB3A	8	★	★	0	★	0	★	"
SRL E	CB3B	8	★	★	0	★	0	★	"

Mnémonique	Code objet (hexa)	Cycles	Registre d'état						Adressage
			S	Z	H	P/V	N	C	
SRL H	CB3C	8	★	★	0	★	0	★	registre
SRL L	CB3D	8	★	★	0	★	0	★	"
SUB (HL)	96	7	★	★	★	★	1	★	indirect
SUB (IX+op)	DD96op	19	★	★	★	★	1	★	indexé
SUB (IY+op)	FD96op	19	★	★	★	★	1	★	"
SUB A	97	4	★	★	★	★	1	★	registre
SUB B	90	4	★	★	★	★	1	★	"
SUB C	91	4	★	★	★	★	1	★	"
SUB D	92	4	★	★	★	★	1	★	"
SUB E	93	4	★	★	★	★	1	★	"
SUB H	94	4	★	★	★	★	1	★	"
SUB L	95	4	★	★	★	★	1	★	"
SUB op	D6op	7	★	★	★	★	1	★	immédiat
XOR (HL)	AE	7	★	★	0	★	0	0	indirect
XOR (IX+op)	DDAEop	19	★	★	0	★	0	0	indexé
XOR (IY+op)	FDAEop	19	★	★	0	★	0	0	"
XOR A	AF	4	★	★	0	★	0	0	registre
XOR B	A8	4	★	★	0	★	0	0	"
XOR C	A9	4	★	★	0	★	0	0	"
XOR D	AA	4	★	★	0	★	0	0	"
XOR E	AB	4	★	★	0	★	0	0	"
XOR H	AC	4	★	★	0	★	0	0	"
XOR L	AD	4	★	★	0	★	0	0	"
XOR op	EEop	7	★	★	0	★	0	0	"

ANNEXE 6

TABLEAUX DE DESASSEMBLAGE

INSTRUCTIONS SANS PREFIXE

- n : octet (8 bits, de 0 à 255) ;
- nn : double octet (16 bits, de 0 à 65535) ;
- d : déplacement pour l'adressage relatif (8 bits).

	Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Ø	NOP	LD BC,nn	LD (BC),A	INC BC	INC B	DEC B	LD B,n	RLCA	EX AF,AF'	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,n	RRCA
1	DJNZ d	LD DE,nn	LD (DE),A	INC DE	INC D	DEC D	LD D,n	RLA	JR d	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,n	RRA
2	JR NZ,d	LD HL,nn	LD (nn),HL	INC HL	INC H	DEC H	LD H,n	DAA	JR Z,d	ADD HL,HL	LD HL,(nn)	DEC HL	INC L	DEC L	LD L,n	CPL
3	JR NC,d	LD SP,nn	LD (nn),A	INC SP	INC (HL)	DEC (HL)	LD (HL),n	SCF C,d	JR C,d	ADD HL,SP	LD A,(nn)	DEC SP	INC A	DEC A	LD A,n	CCF
4	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
A	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
B	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
C	RET NZ	POP BC	JP NZ,nn	JP nn	CALL NZ,nn	PUSH BC	ADD A,n	RST Ø	RET Z	RET	JP Z,nn		CALL Z,nn	CALL nn	ADC A,n	RST 8
D	RET NC	POP DE	JP NC,nn	OUT (n),A	CALL NC,nn	PUSH DE	SUB n	RST 16	RET C	EXX	JP C,nn	IN A,(n)	CALL C,nn		SBC A,n	RST 24
E	RET PO	POP HL	JP PO,nn	EX (SP),HL	CALL PO,nn	PUSH HL	AND n	RST 32	RET DE	JP (HL)	JP PE,nn	EX DE,HL	CALL PE,nn		XOR n	RST 40
F	RET P	POP AF	JP P,nn	DI	CALL P,nn	PUSH AF	OR n	RST 48	RET M	LD SP,HL	JP M,nn	EI	CALL M,nn		CP n	RST 56

INSTRUCTIONS AVEC LE PREFIXE CB

Toutes les instructions de ce tableau doivent être précédées du préfixe CB.

	Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Ø	RLC B	RLC C	RLC D	RLC E	RLC H	RLC L	RLC (HL)	RLC A	RRC B	RRC C	RRC D	RRC E	RRC H	RRC L	RRC (HL)	RRC A
1	RL B	RL C	RL D	RL E	RL H	RL L	RL (HL)	RL A	RR B	RR C	RR D	RR E	RR H	RR L	RR (HL)	RR A
2	SLA B	SLA C	SLA D	SLA E	SLA H	SLA L	SLA (HL)	SLA A	SRA B	SRA C	SRA D	SRA E	SRA H	SRA L	SRA (HL)	SRA A
3									SRL B	SRL C	SRL D	SRL E	SRL H	SRL L	SRL (HL)	SRL A
4	BIT Ø,B	BIT Ø,C	BIT Ø,D	BIT Ø,E	BIT Ø,H	BIT Ø,L	BIT Ø,(HL)	BIT Ø,A	BIT 1,B	BIT 1,C	BIT 1,D	BIT 1,E	BIT 1,H	BIT 1,L	BIT 1,(HL)	BIT 1,A
5	BIT 2,B	BIT 2,C	BIT 2,D	BIT 2,E	BIT 2,H	BIT 2,L	BIT 2,(HL)	BIT 2,A	BIT 3,B	BIT 3,C	BIT 3,D	BIT 3,E	BIT 3,H	BIT 3,L	BIT 3,(HL)	BIT 3,A
6	BIT 4,B	BIT 4,C	BIT 4,D	BIT 4,E	BIT 4,H	BIT 4,L	BIT 4,(HL)	BIT 4,A	BIT 5,B	BIT 5,C	BIT 5,D	BIT 5,E	BIT 5,H	BIT 5,L	BIT 5,(HL)	BIT 5,A
7	BIT 6,B	BIT 6,C	BIT 6,D	BIT 6,E	BIT 6,H	BIT 6,L	BIT 6,(HL)	BIT 6,A	BIT 7,B	BIT 7,C	BIT 7,D	BIT 7,E	BIT 7,H	BIT 7,L	BIT 7,(HL)	BIT 7,A
8	RES Ø,B	RES Ø,C	RES Ø,D	RES Ø,E	RES Ø,H	RES Ø,L	RES Ø,(HL)	RES Ø,A	RES 1,B	RES 1,C	RES 1,D	RES 1,E	RES 1,H	RES 1,L	RES 1,(HL)	RES 1,A
9	RES 2,B	RES 2,C	RES 2,D	RES 2,E	RES 2,H	RES 2,L	RES 2,(HL)	RES 2,A	RES 3,B	RES 3,C	RES 3,D	RES 3,E	RES 3,H	RES 3,L	RES 3,(HL)	RES 3,A
A	RES 4,B	RES 4,C	RES 4,D	RES 4,E	RES 4,H	RES 4,L	RES 4,(HL)	RES 4,A	RES 5,B	RES 5,C	RES 5,D	RES 5,E	RES 5,H	RES 5,L	RES 5,(HL)	RES 5,A
B	RES 6,B	RES 6,C	RES 6,D	RES 6,E	RES 6,H	RES 6,L	RES 6,(HL)	RES 6,A	RES 7,B	RES 7,C	RES 7,D	RES 7,E	RES 7,H	RES 7,L	RES 7,(HL)	RES 7,A
C	SET Ø,B	SET Ø,C	SET Ø,D	SET Ø,E	SET Ø,H	SET Ø,L	SET Ø,(HL)	SET Ø,A	SET 1,B	SET 1,C	SET 1,D	SET 1,E	SET 1,H	SET 1,L	SET 1,(HL)	SET 1,A
D	SET 2,B	SET 2,C	SET 2,D	SET 2,E	SET 2,H	SET 2,L	SET 2,(HL)	SET 2,A	SET 3,B	SET 3,C	SET 3,D	SET 3,E	SET 3,H	SET 3,L	SET 3,(HL)	SET 3,A
E	SET 4,B	SET 4,C	SET 4,D	SET 4,E	SET 4,H	SET 4,L	SET 4,(HL)	SET 4,A	SET 5,B	SET 5,C	SET 5,D	SET 5,E	SET 5,H	SET 5,L	SET 5,(HL)	SET 5,A
F	SET 6,B	SET 6,C	SET 6,D	SET 6,E	SET 6,H	SET 6,L	SET 6,(HL)	SET 6,A	SET 7,B	SET 7,C	SET 7,D	SET 7,E	SET 7,H	SET 7,L	SET 7,(HL)	SET 7,A

INSTRUCTIONS AVEC LE PREFIXE ED

Toutes les instructions de ce tableau doivent être précédées du préfixe ED.

	Ø	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Ø																
1																
2																
3																
4	IN B,(C)	OUT (C),B	SBC HL,BC	ID (nn),BC	NEG	RETN	IM Ø	LD I,A	IN C,(C)	OUT (C),C	ADC HL,BC	LD BC,(nn)		RETI		LD R,A
5	IN D,(C)	OUT (C),D	SBC HL,DE	LD (nn),DE			IM 1	LD A,I	IN E,(C)	OUT (C),E	ADC HL,DE	LD DE,(nn)			IM 2	LD A,R
6	IN H,(C)	OUT (C),H	SBC HL,HL	LD (nn),HL				RRD	IN L,(C)	OUT (C),L	ADC HL,HL	LD HL,(nn)				RLD
7	IN F,(C)		SBC HL,SP	LD (nn),SP					IN A,(C)	OUT (C),A	ADC HL,SP	LD SP,(nn)				
8																
9																
A	LDI	CPI	INI	OUTI					LDD	CPD	IND	OUTD				
B	LDIR	CPIR	INIR	OTIR					LDDR	CPDR	INDR	OTDR				
C																
D																
E																
F																

INSTRUCTIONS INDEXEES

Toutes les instructions de ce tableau doivent être précédées du préfixe DD, pour le registre d'index IX, et de FD, pour le registre IY.

Code	Mnémonique	Code	Mnémonique
09	ADD IX,BC	CB d 0E	RRC (IX+d)
19	ADD IX,DE	CB d 16	RL (IX+d)
21	LD IX,nn	CB d 1E	RR (IX+d)
22	LD (nn),IX	CB d 26	SLA (IX+d)
23	INC IX	CB d 2E	SRA (IX+d)
29	ADD IX,IX	CB d 3E	SRL (IX+d)
2A	LD IX,(nn)	CB d 46	BIT 0,(IX+d)
2B	DEC IX	CB d 4E	BIT 1,(IX+d)
34	INC (IX+d)	CB d 56	BIT 2,(IX+d)
35	DEC (IX+d)	CB d 5E	BIT 3,(IX+d)
36	LD (IX+d),nn	CB d 66	BIT 4,(IX+d)
39	ADD IX,SP	CB d 6E	BIT 5,(IX+d)
46	LD B,(IX+d)	CB d 76	BIT 6,(IX+d)
4E	LD C,(IX+d)	CB d 7E	BIT 7,(IX+d)
56	LD D,(IX+d)	CB d 86	RES 0,(IX+d)
5E	LD E,(IX+d)	CB d 8E	RES 1,(IX+d)
66	LD H,(IX+d)	CB d 96	RES 2,(IX+d)
6E	LD L,(IX+d)	CB d 9E	RES 3,(IX+d)
70	LD (IX+d),B	CB d A6	RES 4,(IX+d)
71	LD (IX+d),C	CB d AE	RES 5,(IX+d)
72	LD (IX+d),D	CB d B6	RES 6,(IX+d)
73	LD (IX+d),E	CB d BE	RES 7,(IX+d)
74	LD (IX+d),H	CB d C6	SET 0,(IX+d)
75	LD (IX+d),L	CB d CE	SET 1,(IX+d)
77	LD (IX+d),A	CB d D6	SET 2,(IX+d)
7E	LD 1,(IX+d)	CB d DE	SET 3,(IX+d)
86	ADD A,(IX+d)	CB d E6	SET 4,(IX+d)
8E	ADC A,(IX+d)	CB d EE	SET 5,(IX+d)
96	SUB (IX+d)	CB d F6	SET 6,(IX+d)
9E	SBC A,(IX+d)	CB d FE	SET 7,(IX+d)
A6	AND (IX+d)	E1	POP IX
AE	XOR (IX+d)	E3	EX (SP),IX
B6	OR (IX+d)	E5	PUSH IX
BE	CP (IX+d)	E9	JP (IX)
CB d 06	RLC (IX+d)	F9	LD SP,IX

INDEX

Les procédures et les fonctions sont référencées à l'annexe 2.

CONSEILS DE LECTURE

Pour approfondir vos connaissances en Basic et en Pascal, mieux connaître le système des CPC 464, 664 et 6128, ainsi que du PCW 8256, P.S.I. vous propose une palette d'ouvrages utiles.

POUR MAITRISER LE BASIC AMSTRAD

- **Basic Amstrad 1 - Méthodes pratiques -**
  Jacques Boisgontier et Bruno César (Editions du P. S. I.)

  Pour ceux qui ont déjà pratiqué un Basic, voici un ouvrage de perfectionnement au Basic Amstrad. Un chapitre sur le CP/M 2.2 et Le CP/M Plus donne les principales commandes système.

- **Basic Amstrad 2 - Programmes et fichiers -**
  Jacques Boisgontier (Editions du P.S. I.)

  Pour pratiquer le Basic Amstrad, cet ouvrage donne de nombreux programmes de gestion, d'éducation et de jeu où le rôle des fichiers est expliqué et largement commenté.

- **Basic Plus - 80 routines sur Amstrad -**
  Michel Martin (Editions du P.S.I.)

  Pour pousser votre Amstrad au maximum de ses capacités : 80 routines de simulation d'instructions qui n'existent pas en Basic Amstrad.

POUR MIEUX CONNAITRE LE SYSTEME DES CPC ET DU PCW 8256

- **Clefs pour Amstrad - Systeme de base -**
  Daniel Martin (Editions du P.S.I.)

Memento présentant synthétiquement le jeu d'instructions du Z80, les points d'entrée des routines système, les connecteurs et brochages, etc. Le livre de chevet du programmeur sur Amstrad.

- **Clefs pour Amstrad 2 - Système disque -**
  Daniel Martin et philippe Jadoul (Editions du P.S.I.)

  Ce deuxième tome consacré au système disque présente les points d'entrées des routines disque, les blocs de contrôle, la programmation et les brochages des circuits spécialisés... La deuxième partie du livre est aussi destinée aux possesseurs d'Amstrad 8256.

- **CP/M Plus sur Amstrad .**
  Yvon Dargery (Editions du P.S.I.)

  Toutes les commandes CP/M et CP/M Plus pour maîtriser le système 6128 et 8256 : ouvrage de référence illustré par de nombreux programmes.

- **Le livre de l'Amstrad - Tome 1 -**
  Daniel Martin et Philippe Jadoul (BCM - diffusé par P.S.I.)

  Ce livre, destiné aux programmeurs des CPC 464, 664 et 6128, donne une étude complète de tous les circuits internes et analyse la structure interne du Basic. Vous y trouverez, en outre, une étude complète des RSX et des programmes de scrolling, de traçage de rectangle, de coloriage de surface et de manipulation vectorielle.

ENFIN, POUR UTILISER A FOND LE LANGAGE PASCAL ET SON FRERE LE TURBO-PASCAL

- **Programmer en Pascal**
  Daniel-Jean David et Jean-Luc Deschamps (Editions du P.S.I.)

  Voici une solide initiation à la programmation en Pascal, illustrée par de nombreux exercices résolus.

Votre avis nous intéresse

- Pour nous permettre de faire de meilleurs livres, adressez-nous vos critiques sur le présent livre.
- Si vous souhaitez des éclaircissements techniques, écrivez-nous, nous adresserons votre demande à l'auteur qui ne manquera pas de vous répondre directement.

- Ce livre vous donne-t-il toute satisfaction?

_____
_____
_____
_____
_____

- Y a-t-il un aspect du problème que vous auriez aimé voir abordé?

_____
_____
_____
_____
_____

Comment avez-vous eu connaissance de ce livre?

☐ publicité
☐ catalogue
☐ boutique micro
☐ autres
☐ cadeau
☐ librairie
☐ exposition

Avez-vous déjà acquis des livres PSI?

lesquels? _____
_____

qu'en pensez-vous? _____
_____
_____

Nom _____ Prénom _____ Age _____
Adresse _____
Profession _____
Centre d'intérêt _____

Achevé d'imprimer par l'imprimerie Tardy Quercy S.A. à Bourges
D. L. : Février 1986 - N° d'Imprimeur : 12784 - N° d'Édition : 86595-310-1

TURBO PASCAL SUR AMSTRAD

Turbo Pascal sur Amstrad s'adresse à vous, possesseurs des CPC 464, 664 et 6128, ou du PCW 8256, qui souhaitez apprendre et maîtriser ce langage très puissant.

Turbo Pascal est, en effet, d'une approche plus facile que le Pascal standard : tenant en un seul programme, il produit du véritable code machine, permet une remarquable gestion des erreurs, et autorise pratiquement le mode direct.

Mais *Turbo Pascal sur Amstrad* est aussi un livre de référence dans lequel vous trouverez toutes les commandes expliquées et illustrées. Vous y apprendrez même à faire de l'assembleur à l'intérieur des routines Pascal !

Cet ouvrage, très pédagogique, suit une difficulté croissante pour arriver à un haut niveau : fonctionnement de Heap et de Pile, maîtrise des pointeurs, etc.

9 782865 953103

ÉDITIONS DU P.S.I.
BP 86 - 77402 LAGNY-S/MARNE CEDEX - FRANCE

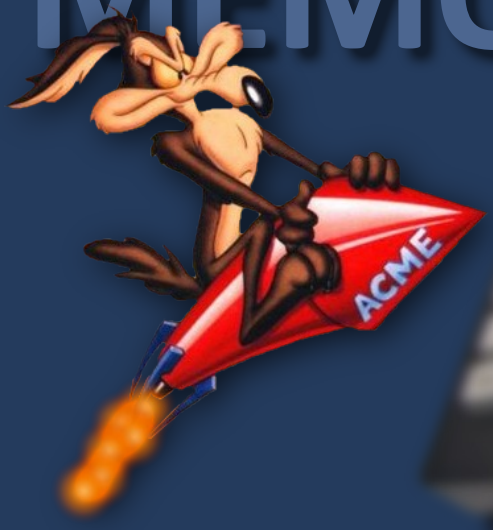
ISBN : 2 86595-310 6 135 F.F.

ILLUSTRATION : CLAUDE LOCHU

Document numérisé avec amour par :

AMSTRAD CPC

MÉMOIRE ÉCRITE



https://acpc.me/