Variables et structures de données Chapitre 5

Le Chapitre Un a traité le format de base des données en mémoire. Le Chapitre Trois a décrit comment un ordinateur organise physiquement les données. Ce chapitre complète la discussion en faisant le lien entre la représentation conceptuelle et physique des données. Ce qui implique l'étude de deux sujets fondamentaux : les variables et la structuration des données. Les notions traitées ici ne présupposent pas que vous ayez pris un cours formel préalable sur les structures de données, même si une telle expérience pouvait s'avérer utile.

5.0 Vue d'ensemble du chapitre

Ce chapitre décrit comment déclarer et accéder les variables scalaires, les entiers, les réels, les types de données, les pointeurs, les tableaux et les structures. Vous devez maîtriser ce sujet avant de passer au chapitre suivant. En particulier, la déclaration des tableaux et leur accès semble poser de nombreux problèmes pour les débutants. Le reste de ce livre dépend de votre compréhension de ces structures et de leur représentation dans la mémoire physique. N'essayez pas de sauter ce chapitre avec l'espoir de le reprendre au besoin. Vous devez apprendre cette matière tout de suite, sinon, ce qui viendra ensuite ne fera autre chose qu'augmenter votre confusion.

5.1 Quelques instructions supplémentaires : LEA, LES, ADD et MUL

Le but de ce chapitre n'est pas celui de présenter l'ensemble du jeu d'instructions des processeurs 80x86. Toutefois, il y a quatre instructions supplémentaires (qui s'ajoutent à mov tout en apportant un plus), qui s'avèrent très pratiques dans la matière couverte ici. Ces instructions sont *load effective address (lea)* (charger l'adresse réelle), *load ES and general purpose register (les)* (charger ES et un registre général, tel que ES:reg), *addition (add)* et multiplication *(mul)*. Ces instructions (combinées avec mov), donnent toute la puissance nécessaire pour avoir accès à tous les types de données présentés dans ce chapitre.

L'instruction lea prend la forme :

```
lea     reg16, mémoire
```

où *reg16* est un registre général de 16 bits et *mémoire* est l'adresse représentée par un octet mod/reg/rm (sauf que dans ce cas, il doit s'agir d'un emplacement de mémoire et non d'un registre).

Cette instruction charge le registre de 16 bits avec l'offset de l'emplacement spécifié par l'opérande de mémoire. Par exemple, lea ax, 1000h[bx][si] charge ax de l'adresse pointée par 1000h[bx][si]. Celle-ci, naturellement, correspond à la valeur de 1000h+bx+si. lea est également utile pour obtenir l'adresse d'une variable. Si vous avez une variable *I* quelque part dans la mémoire, lea bx, I chargera le registre bx avec l'adresse (offset) de I.

L'instruction les prend la forme :

```
les     reg16, mémoire
```

Elle charge le registre es et un des registres généraux de 16 bits avec l'adresse de mémoire spécifiée. Notez que toute adresse de mémoire que vous pouvez spécifier avec un octet mod/reg/rm est valide, mais comme dans le cas de l'instruction lea, il doit s'agir exclusivement d'un emplacement de mémoire et non d'un registre.

L'instruction *les* charge le registre général spécifié avec le mot à l'adresse fournie et il charge le registre es avec la valeur suivante dans la mémoire. Il y a aussi l'instruction *lds* (qui charge le registre ds au lieu de es). Ces deux instructions sont les seules qui chargent des données de 32 bits sur des systèmes antérieurs au 80386.

L'instruction *add*, tout comme sa contrepartie x86, additionne deux valeurs. Elle a diverses versions. En voici cinq qui nous concernent particulièrement :

```
add     reg, reg
add     reg, mem
add     mem, reg
```

```
add     reg, const
add     mem, const
```

Toutes ces versions additionnent la première et la seconde opérande en laissant le résultat dans la première. Par exemple, add bx, 5 calcule bx := bx + 5.

La dernière instruction, mul, a seulement une opérande et prend la forme :

```
mul     reg/mémoire
```

Il y a beaucoup de détails concernant mul que ce chapitre ignore. Dans le but de ce qui suit, présumez que le registre et l'emplacement de mémoire sont de 16 bits. Dans cette éventualité, cette instruction calcule : dx:ax :=ax*reg/mem[1]. Notez qu'il n'y a pas de mode immédiat pour cette instruction.

5.2 Déclaration des variables dans un programme en assembleur

Bien que vous ayez déjà déduit que les emplacements de la mémoire et les variables ont des relations, ce chapitre n'a pas encore souligné les forts parallélismes qu'il y a entre les deux. Eh bien, il est temps de rectifier à cela. Considérez le programme suivant en Pascal, aussi court qu'inutile :

```
program useless(input, output);
var i, j:integer;
begin
            i:=10;
            write('Entrez une valeur pour j: ');
            readln(j);
            i := i*j + j*j;
            writeln('Le résultat est ', i);
end.
```

Quand l'ordinateur exécute l'instruction i:=10;[2] il fait une copie de la valeur 10 et, d'une manière ou d'une autre, il se souvient de cette valeur pour une utilisation ultérieure. Pour ce faire, le compilateur réserve une adresse de mémoire pour l'usage exclusif de la variable *i*. En supposant qu'il assigne de façon arbitraire l'emplacement DS:10h pour le contenu de la variable *i*, l'opération en assembleur réalisant cette affectation est quelque chose comme mov ds:[10h], 10[3]. Si i est un mot de 16 bits, le compilateur affecterait probablement la variable j avec le mot commençant à l'adresse 12h ou 0Eh. En supposant que c'est 12h, la seconde opération d'affectation dans le programme ressemblerait à :

```
mov     ax, ds:[10h]    ;chargement de la valeur de i
mul     ds:[12h]        ;multiplication par j
mov     ds:[10h], ax    ;enregistrement dans i (en ignorant tout
                        ;débordement)
mov     ax, ds:[12h]    ;chargement de j
mul     ds:[12h]        ;calcul de j*j
add     ds:[10h], ax    ;Effectuer i*j + j*j et stocker résultat dans i
```

Bien qu'un tel code souffre de l'omission de certains détails importants pour son fonctionnement, il donne tout quand même une idée précise sur ce qui se passe en mémoire lorsqu'on affecte des valeurs à des variables.

Maintenant imaginez un programme de 5000 lignes avec des variables comme ds:[10h], ds[12h], ds:[14h], etc. Qu'en serait-il si vous deviez chercher l'instruction où vous avez accidentellement stocké le résultat d'un calcul dans j au lieu de i ? Vous devriez même faire attention à l'emplacement de chaque variable dans la mémoire et savoir par exemple que i correspond à 10h et j à 12h. Ne serait-ce pas mieux d'utiliser des noms comme i et j au lieu de se préoccuper de leurs adresses numériques ? Il semble donc raisonnable de réécrire le code de la manière suivante :

```
mov     ax, i
```

[1]Car, toutes les fois que vous multipliez deux valeurs de 16 bits, vous pouvez obtenir un résultat de 32 bits. Ce résultat est placé dans dx:ax, avec le mot le plus significatif dans dx et le mot le moins significatif dans ax.

[2]En réalité, l'ordinateur exécute le *code machine* émis par le compilateur Pascal pour cette instruction. Mais vous n'avez pas à vous préoccuper de ce fait ici.

[3]Mais n'essayez pas de faire cela vous-même ! Car il y aurait des détails syntactiques à affiner et MASM émettrait un message d'erreur si vous essayez d'assembler cette instruction particulière.

```
mul     j
mov     i, ax
mov     ax, j
mul     j
add     i, ax
```

Bien sûr que vous pouvez faire ceci en assembleur ! L'un des travaux primaires d'un assembleur comme MASM est de vous permettre de vous servir de noms symboliques à la place des emplacements de mémoire. Bien plus, l'assembleur fait automatiquement pour vous l'assignation des adresses aux noms. Le fait que la variable i correspond au mot en mémoire de l'adresse DS:10h ne vous concernera plus, sauf si vous êtes curieux.

Cela ne devrait pas vous surprendre que ds pointe sur le segment dseg dans le fichier SHELL.ASM. Bien au contraire, faire pointer ds sur dseg est l'une des premières choses qui se passent dans le programme principal de SHELL.ASM. Par conséquent, tout ce que vous devrez faire est d'indiquer à l'assembleur de réserver de l'espace pour vos variables dans dseg et d'associer l'offset de toute variable avec le nom de celle-ci. Il s'agit d'un processus très simple qui sera le sujet des prochaines sections.

5.3 Déclaration et accès des variables scalaires

Les variables scalaires contiennent des valeurs uniques. Les variables i et j de la section précédente sont des exemples de variables scalaires. Les exemples de structures de données qui ne sont pas de ce type incluent les tableaux, les structures (enregistrements), les ensembles et les listes. Ces derniers types sont construits à partir de variables scalaires et ils sont des *types de données composés*. Vous en verrez un peu plus tard dans ce chapitre ; avant, vous devez apprendre à manipuler des variables simples.

Pour déclarer une variable dans dseg, il faut utiliser une instruction comme :

```
ByteVar         byte    ?
```

ByteVar est une *étiquette*. Elle est censée commencer à la première colonne de la ligne se trouvant quelque part dans dseg (ceci entre les instructions *dseg* et *dseg ends*). Vous découvrirez tout ce qui concerne les étiquettes dans quelques chapitres ; pour l'instant, vous pouvez supposer que tout identificateur valide en Pascal, C ou Ada l'est aussi en assembleur.

Si vous avez besoin de plus d'une variable dans vos programmes, il vous suffira de placer des lignes additionnelles dans le segment dseg déclarant ces variables et MASM allouera automatiquement un espace de stockage unique pour chaque variable (car, avoir i et j associés à la même adresse ne serait pas trop bon n'est-ce pas ?). Après avoir déclaré ces dites variables, MASM vous permettra de vous référer à ces dernières par *nom* au lieu que par adresse. Par exemple, après avoir inséré l'instruction de l'exemple ci-dessus dans le segment de données (dseg), vous pourriez utiliser des instructions comme mov ByteVar, al dans votre programme.

La première variable que vous placez dans le segment de données est associée à l'adresse DS:0. La variable suivante obtient une autre adresse se trouvant juste après la première. Par exemple, si la variable à l'emplacement zéro était une variable de type byte, l'adresse de la variable successive sera DS:1. Et si la première variable était de type word, l'adresse de la seconde variable serait DS:2. MASM alloue des adresses avec beaucoup de soin et fait très attention à éviter les chevauchements (overlap). Considérez la définition suivante de dseg :

```
dseg            segment     para public 'data'
bytevar         byte        ?           ;byte alloue des octets
wordvar         word        ?           ;word alloue des mots
dwordvar        dword       ?           ;dword alloue des doubles-mots
byte2           byte        ?
word2           word        ?
dseg            ends
```

MASM alloue de l'espace pour bytevar à l'adresse DS:0, parce que bytevar mesure un octet. Ce qui fait que l'emplacement suivant disponible est DS:1. Ensuite, il fait l'allocation pour wordvar à l'adresse DS:1 ; puiqu'un mot requiert deux octets, l'adresse disponible suivante après wordvar sera DS:3, où MASM allouera de l'espace

pour la variable dwordvar. Cette dernière a une taille de quatre octets, donc byte2 sera allouée à l'emplacement DS:7. De la même façon, MASM allouera de la mémoire pour word2 à l'emplacement DS:8. Et si vous vouliez ajouter encore une autre variable, sa place commencerait à l'adresse DS:0A.

Quand vous faites référence à un des identificateurs ci-dessus, MASM leur substituera automatiquement l'offset approprié. Par exemple, l'instruction mov ax, wordvar sera traduite par mov ax, ds:[1]. Donc, vous pouvez à présent utiliser des noms symboliques pour vos variables et ignorer complètement le fait que ces dernières sont en fait des adresses de mémoire avec leurs offsets correspondants dans le segment de données.

5.3.1 Déclaration et utilisation des variables BYTE

Donc, quelle est l'utilité réelle des variables byte ? Simple, vous pouvez représenter tout type de donnée ayant moins de 256 valeurs différentes. Ceci inclut des données très importantes et souvent utilisées, comme le jeu de caractères, les valeurs booléennes, la plupart des types d'énumération et, naturellement, de petits entiers (signés on non signés) ; on n'a mentionné que quelques exemples.

Les caractères dans un ordinateur IBM ou compatible, se représentent par un jeu de caractères ASCII/IBM de huit bits (voir à ce propos l'annexe A[4]). Le jeu 80x86 comprend un riche ensemble d'instructions pour manipuler des données caractère ; et il n'est pas surprenant que la plupart des variables de type byte servent à stocker en général des caractères.

Le type booléen représente seulement deux valeurs : vrai ou faux. Par conséquent, un seul bit suffirait à représenter une valeur booléenne. Cependant, les systèmes 80x86 travaillent mieux avec des blocs de huit bits, car, il faut plus de code pour travailler avec un bit qu'avec un octet en bloc. Donc, il vaut mieux d'utiliser un byte pour représenter une valeur booléenne. La plupart des programmeurs utilisent la valeur 0 pour représenter faux et toute autre valeur (1 en général) pour représenter vrai. Le drapeau zéro du registre Flags rend le test zéro/non-zéro très simple. Notez que ce choix entre zéro et non zéro est uniquement dicté par convention ; vous pouvez utiliser toute paire de valeurs (ou de groupes de valeurs) pour représenter vrai ou faux.

La plupart des langages de haut niveau qui ont du support pour les types de données énumératifs convertissent ces données (de façon interne) en entiers non signés. Le premier élément de la liste est généralement l'élément zéro, le second, l'élément un, le troisième deux, etc. Par exemple, considérez le type d'énumération suivant, en Pascal :

```
couleurs = (rouge, bleu, vert, violet, orange, jaune, blanc, noir);
```

La plupart des compilateurs Pascal attribuent la valeur 0 à rouge, 1 à bleu, 2 à vert, etc.

Plus tard, vous verrez comment créer vos propres types énumératifs en langage assembleur. Tout ce que vous avez besoin d'apprendre maintenant est comment allouer de l'espace pour une variable qui contient des valeurs énumérées. Puisqu'il est peu probable d'avoir plus de 256 valeur de ce genre, vous pouvez utiliser une simple variable d'un octet pour stocker la valeur. Si vous avez une variable nommée *couleur* de type *couleurs* et vous utilisez l'instruction mov couleur, 2 c'est la même chose que dire couleur:=vert en Pascal (plus tard vous apprendrez aussi comment utiliser des instructions plus significatives comme mov couleur, vert pour attribuer la couleur vert à la variable couleur).

Sans doute, si vous avez une petite variable entière non signée (0...255) ou signée (-128...127), une variable de type byte est le meilleur choix dans la plupart des cas. Notez que la plupart des programmeurs considèrent tous les types de données (sauf les petits entiers signés) comme des valeurs non signées. C'est-à-dire, certains types comme les caractères, les valeurs booléennes, les types énumérés et les entiers non signés sont tous considérés comme des valeurs non signées. Dans certains cas spéciaux, vous pouvez considérer des caractères comme des valeurs signées, mais, le plus souvent, ce n'est pas le cas.

Il y a trois instructions permettant de déclarer des variables byte dans un programme :

```
identificateur          db      ?
identificateur          byte    ?
identificateur          sbyte   ?
```

[4]Cet annexe n'a jamais été publié dans la version originale de *Art of Assembly Language*. Toutefois, cette traduction en comprend une version rédigée par le traducteur, n.d.t.

où *identificateur* représente le nom de votre variable. "db" est un vieux terme, qui date d'avant MASM 6.x. Vous verrez certes cette déclaration encore en usage chez certains programmeurs (spécialement ceux qui utilisent encore des versions de MASM antérieures à la version 6), mais Microsoft considère ce terme comme obsolète ; vous devriez toujours utiliser les déclarations *byte* et *sbyte* à la place.

La déclaration byte déclare des valeurs non signées ; vous devriez toujours utiliser cette déclaration pour toutes les valeurs entières (sauf pour les petits entiers signés). Pour des valeurs signées, utilisez la directive sbyte (pour *signed byte*).

Une fois que vous avez déclaré quelques variables avec ces directives vous pouvez y faire référence dans votre programme par leur nom :

```
i               db      ?
j               byte    ?
k               sbyte   ?
                .
                .
                .
                mov     i, 0
                mov     j, 245
                mov     k, -5
                mov     al, i
                mov     j, al
                ;etc.
```

Bien que MASM effectue un certain contrôle de type, vous ne devriez pas vous imaginer que l'assembleur est un langage fortement typé. En fait, MASM 6.x vérifiera seulement la *capacité* des valeurs dans leur emplacement cible. Tout ceci suit est correct dans MASM 6.x :

```
                mov     k, 255
                mov     j, -5
                mov     i, 127
```

Puisque toutes ces variables sont de type byte et toutes les constantes associées sont assez petites pour entrer dans huit bits, MASM les accepte toutes. Mais si vous y jetez un regard, vous verrez qu'elles sont logiquement incorrectes. En effet, qu'est-ce que cela représente -5 dans une variable non signée ? Et, puisque des valeurs signées de type byte doivent se trouver dans la plage -128...127, qu'est-ce qu'il arrive quand vous stockez 255 dans une variable byte signée ? MASM convertira simplement ces valeurs dans leur équivalent de huit bits (-5 devient 0FBh et 255 devient 0FFh, c.à.d., -1).

Une version ultérieure de MASM effectuera peut-être un plus fort contrôle sur les types, peut-être non. Cependant vous devez toujours garder à l'esprit que ce type de contrôle peut faire défaut. C'est à vous d'écrire vos programmes correctement. L'assembleur ne vous viendra pas en aide autant que Pascal ou Ada. Et, même si un fort contrôle de type était parfaitement en fonction, il y aurait toujours le moyen de contourner la règle. Considérez la séquence suivante :

```
                mov     al, -5
                 .
        ; N'importe quel nombre d'instructions qui n'affectent pas AL
                 .
                mov     j, al
```

Malheureusement, il n'y a pas de moyen pour l'assembleur de déterminer si vous avez placé une valeur correcte dans j[5]. Les registres, par leur nature même, ne sont ni signés ni non signés ; et l'assembleur vous permet de placer le contenu d'un registre dans une variable, indépendamment de la valeur qui peut se trouver dans ce registre.

Bien que l'assembleur ne vérifie pas si les opérandes d'une instruction sont signées ou non, il peut certainement contrôler leur taille ; si celle-ci n'est pas appropriée, l'assembleur se plaindra avec un message d'erreur approprié. Les exemples suivants sont tous incorrects :

```
                mov     i, ax           ;pas possible de placer 16 bits dans 8
```

[5]Pour ce simple cas, vous pouvez maintenant modifier l'assembleur pour qu'il détecte le problème. Mais il est assez facile de tomber sur un exemple un peu plus complexe où l'erreur *ne pourra* pas être détectée.

```
mov     i, 300          ;300 dépasse la capacité d'un octet
mov     k, -130         ;-130 est hors de la plage -128...127
```

Vous vous demandez peut-être : « Si l'assembleur ne fait pas vraiment de différence entre valeurs signées et non signées, pourquoi s'en inquiéter ? Pourquoi ne pas utiliser *db* tout le temps ? ». Eh bien, il y a deux raisons à cela : en premier lieu si vous spécifiez le signe de vos variables, cela rend vos programmes plus lisibles et compréhensibles ; ensuite, qui a dit que l'assembleur est *toujours* incapable de voir si une variable est signée ou non ? Certes, l'instruction mov ignore la différence, mais il y a d'autres instructions qui ne le font pas.

Un dernier point qui vaut la peine de mentionner concerne la déclaration des variables byte. Dans toutes les déclarations vues jusqu'à présent, l'opérande de l'instruction a toujours été représentée par un point d'interrogation ; celui-ci indique à l'assembleur que la variable doit rester indéfinie quand DOS charge le programme en mémoire[6]. Vous pouvez spécifier une valeur initiale qui sera chargée en mémoire avant que le programme commence son exécution, simplement en plaçant cette valeur à la place du point d'interrogation. Considérez les déclarations suivantes :

```
i       db      0
j       byte    255
k       sbyte   -1
```

Dans cet exemple, l'assembleur initialisera i, j et k à 0, 255 et -1 respectivement pendant le chargement du programme en mémoire. Ce fait s'avérera très utile plus tard, spécialement quand on parlera des tableaux. Une fois encore, l'assembleur vérifie seulement la taille des opérandes sans considérer le signe ou la juste valeur ; MASM permet toute valeur de la plage -128...255 comme opérande pour chacune de ces déclarations.

Si vous avez l'impression qu'il n'y a pas encore de raison valide d'utiliser byte ou sbyte dans un programme, vous devriez noter que, si MASM ignore parfois la différence dans ces définitions, MS CodeView, de sa part, ne le fait jamais. Si vous avez déclaré une variable signée, CodeView l'affichera comme telle (en incluant le signe moins, si nécessaire). D'autre part, CodeView affichera toujours les variables db et byte en tant que valeurs positives (et ignorera les cas non signés).

5.3.2 Déclaration et Utilisation des Variables WORD

Les programmes 80x86 se servent des variables de type word (mot) pour trois choses : les entiers signés de 16 bits, les entiers non signés de 16 bits et les offsets (pointeurs) en assembleur 16 bits[7]. Oh, certainement, vous pouvez utiliser ces variables pour beaucoup d'autres choses, mais ces trois représentent les applications les plus communes.

Puisque le mot est le type de données le plus large sur les systèmes 8086, 8088, 80186, 80188 et 80286, vous trouverez que pour la majorité des programmes anciens ce type sert de base pour la plupart des utilisations. Sans doute, le processeur 80386 et ultérieurs permettent des opérations de 32 bits, mais certains programmes ne se servent pas d'instructions de 32 bits, car ceci les rendrait incompatibles avec les systèmes plus anciens.

Pour déclarer des variables word, on utilise les instructions dw, word et sword. L'exemple suivant illustre leur utilisation :

```
NoSignedWord            dw      ?
UnsignedWord            word    ?
SignedWord              sword   ?
Initialized0            word    0
InitializedM1           sword   -1
InitializedBig          word    65535
InitializedOfs          dw      NoSignedWord
```

La plupart de ces déclarations sont des modifications légères des déclarations de type byte qu'on a déjà vu. Bien sûr, vous pouvez initialiser toute variable word à une valeur de la plage -32768...65536 (l'union des plages des constantes signées et non signées). La dernière déclaration ci-dessus, cependant, est nouvelle : dans ce cas, une étiquette apparaît à la place de l'opérande (en référençant le nom de la variable *NoSignedWord*).

[6]En fait, DOS initialise ces variables à 0, mais vous ne devriez pas compter là-dessus.

[7]A ceci, il faut ajouter la représentation des caractères Unicode pour la programmation en assemblur 32 bits, qui fait l'objet d'un autre ouvrage de Randall Hyde, n.d.t.

Quand une étiquette apparaît à cet emplacement, l'assembleur remplace cette étiquette par l'offset de la variable qu'elle représente (à l'intérieur du segment de cette variable). Si c'étaient les seules déclarations dans dseg et qu'elles apparaissaient dans cet ordre, la dernière déclaration initialiserait *InitializedOfs* avec la valeur zéro, puisque c'est l'offset de *NoSignedWord* dans le segment de données. Cette forme d'initialisation est très utile pour initialiser des *pointeurs*. On y reviendra.

Le débogueur CodeView fait la différence entre les variables dw/word et sword. Il affiche toujours les valeurs non signées sous la forme d'entiers positifs. D'autre part, il affiche les variables sword comme des valeurs signées (complétées par le signe moins si la valeur est négative). Le support du débogueur est l'une des principales raisons pour lesquelles il faut utiliser les mots-clés word et sword de façon appropriée.

5.3.3 Déclaration et utilisation des variables DWORD

Pour déclarer des entiers de quatre octets et d'autres types de données, vous pouvez utiliser les directives dd, dword et sdword. De telles variables permettent des valeurs dans la plage -2 147 483 648 ... 4 294 967 295 (l'union des plages des entiers de quatre octets signés et non signés). On utilise ces déclarations de la même façon que celles pour des types word :

```
NoSignedDWord           dd      ?
UnsignedDWord           dword   ?
SignedDWord             sdword  ?
InitBig                 dword   4000000000
InitNegative            sdword  -1
InitPtr                 dd      InitBig
```

Le dernier exemple, initialise un pointeur de la taille d'un double-mot avec l'offset de type segment:offset de la variable *InitBig*[8].

Encore une fois, cela vaut la peine de noter que l'assembleur ne vérifie pas le type de ces variables selon les valeurs avec lesquelles elles sont initialisées. Si la valeur est contenue dans 32 bits, l'assembleur l'accepte. La vérification de la taille, cependant, est fortement assurée. Puisque les seules instructions de 32 bits dans des processeurs antérieurs au 80386 sont les et lds, vous obtiendrez une erreur si vous essayez d'accéder à des variables dword par l'instruction mov. Évidemment, même sur un 80386, vous ne pouvez pas transférer une valeur de 32 bits dans un registre de 16 bits ; vous devrez donc vous servir de registres de 32 bits. Plus tard vous apprendrez comment manipuler des variables de 32 bits, même sur un processeur de 16 bits. Mais, jusque-là faites comme si vous ne pouviez pas.

Gardez à l'esprit pourtant que CodeView fait toujours la différence entre dd/dword et sdword. Ceci aide à voir la valeur de vos variables quand vous êtes en train de déboguer vos programmes ; et CodeView fait ceci correctement seulement quand vous vous servez de déclarations correctes. Utilisez toujours sdword pour des valeurs signées et dd ou dword (dword est mieux) pour des valeurs non signées.

5.3.4 Déclaration et utilisation de variables FWORD, QWORD et TBYTE

MASM 6.x permet aussi la déclaration de variables de six, huit et dix octets en utilisant les instructions df/fword, dq/qword et dt/tbyte. Les déclarations qui utilisent ces directives ont été originairement conçues pour des valeurs en virgule flottante et en BCD (Décimal Codé Binaire). Il y a de meilleures directives pour les variables en virgule flottante et vous n'avez pas besoin de vous occuper des autres types de données que ces formats supportent. On les mentionne seulement pour donner un cadre complet.

La principale utilité des instructions df/fword est déclarer des pointeurs de 48 bits pour les systèmes de 32 bits en mode protégé (80386 et ultérieurs). Bien que vous pouvez utiliser ces directives pour créer une variable arbitraire de six octets, il y a de meilleures instructions pour le faire. Vous devriez utiliser celles-ci seulement pour des pointeurs *far* de 48 bits sur 80386 de format ssss:oooo oooo.

dq/qword vous permettent de déclarer des variables de huit octets (*quadword*). La fonction principale de ces directives est créer des variables de 64 bits en double précision ou des entiers de la même taille. Il y a de meilleures façons de créer des variables en virgule flottante. Pour ce qui concerne les entiers de 64 bits, ils ne

[8]En assembleur 32 bits, il s'agit d'un offset simple, n.d.t.

vous seront pas très utiles, au moins qu'Intel ne décide de lancer un système supportant des registres généraux de 64 bits[9].

Les directives dt/tbyte permettent d'allouer dix octets de stockage. Il y a deux types de données originaires de la famille 80x87 (des coprocesseurs mathématiques) qui se servent des types de données de dix octets : des valeurs BCD de cette taille et des valeurs en virgule flottante étendue à 80 bits. Ce livre ignorera délibérément le type de données BCD, et pour ce qui concerne les types à virgule flottante, il y a - encore une fois - une meilleure manière de faire.

5.3.5 Déclaration des variables en virgule flottante avec REAL4, REAL8 et REAL 10

Voici les directives que vous devez utiliser pour déclarer des variables en virgule flottante. Comme dd, dq et dt, ces instructions réservent quatre, huit et dix octets. Le champ de l'opérande pour ces instructions peut contenir un point d'interrogation (si vous ne voulez pas initialiser la variable tout de suite), ou bien une valeur initiale en format de virgule flottante. Les exemples suivants démontrent son utilisation :

```
x               real4   1.5
y               real8   1.0e-25
z               real10  -1.2594e+10
```

Notez que l'opérande doit être une constante valide en virgule flottante en notation décimale ou scientifique. En particulier, *des constantes entières pures ne sont pas permises*. L'assembleur émettra une erreur si vous utilisez une opérande comme :

```
x               real4   1
```

Pour corriger, changez le champ de l'opérande par "1.0".

Notez qu'effectuer des opérations en virgule flottante requiert des composants matériels spéciaux (par exemple, une puce 80x87 ou une 80x86 avec coprocesseur mathématique intégré). Si de tels composants ne sont pas disponibles, vous devrez écrire des composants logiciels pour effectuer des opérations comme l'addition, la soustraction, la multiplication, etc., en virgule flottante. En particulier, vous ne pouvez pas utiliser l'instruction add des processeurs 80x86 pour additionner deux valeurs réelles. Ce livre couvrira l'arithmétique flottante dans un chapitre ultérieur (voir le chapitre 14). Néanmoins, il est utile d'expliquer comment déclarer des variables de cette sorte dans un chapitre traitant les structures de données.

MASM vous permet aussi d'utiliser dd, dq et dt pour déclarer de variables en virgule flottante (puisque ces directives allouent les quatre, huit et dix octets nécessaires au stockage). Vous pouvez même initialiser de telles variables en plaçant des constantes réelles dans le champ de l'opérande. Mais il y a deux inconvénients à les déclarer de cette façon : d'abord, comme pour les octets, mots et doubles-mots, CodeView affichera correctement vos valeurs en virgule flottante seulement si vous les déclarerez avec real4, real8 et real10. Si vous utilisez dd, dq ou dt, CodeView affichera vos variables sous forme d'entiers de quatre, huit ou dix octets. Un autre problème potentiellement plus grand en utilisant dd, dq et dt est que ces directives permettent autant les entiers que les valeurs en virgule flottante (et souvenez-vous que real4, real8 et real10 ne le font pas). Maintenant, ceci pourrait paraître une bonne caractéristique de premier abord. Cependant, la représentation entière de la valeur 1 *n'est pas* la même que la représentation en virgule flottante de la valeur 1.0. Donc, si vous entrez accidentellement la valeur "1" dans le champ de l'opérande, alors qu'en réalité vous vous référiez à "1.0", l'assembleur l'acceptera sans réplique et vous donnera des résultats incorrects. Donc, déclarez toujours des variables en virgule flottante avec real4, real8 ou real10.

5.4 Création de types personnalisés avec TYPEDEF

Supposons simplement que vous n'aimez pas les types que Microsoft a définis pour déclarer des variables et que vous préférez les conventions C ou Pascal. Vous préférez vraisemblablement utiliser des termes comme *integer*, *float*, *double*, *char*, *boolean* et ainsi de suite. Si on travaillait en Pascal, on pourrait redéfinir les noms dans la section **type** du programme. Avec C, vous pourriez utiliser **#define** ou **typedef** pour faire la même chose. MASM 6.x a aussi sa propre instruction *typedef* pour créer des alias pour ces noms. L'exemple suivant démontre comment mettre au point des noms compatibles avec Pascal dans vos programmes assembleur :

[9]C'est fait !, n.d.t.

```
integer         typedef         sword
char            typedef         byte
boolean         typedef         byte
float           typedef         real4
colors          typedef         byte
```

Maintenant, vous pouvez déclarer vos variables avec des instructions plus significatives, comme :

```
i               integer         ?
ch              char            ?
FoundIt         boolean         ?
x               float           ?
HouseColor      colors          ?
```

Si vous êtes un programmeur C, Ada ou FORTRAN (ou tout autre langage), vous pourriez sélectionner les noms de types avec lesquels vous vous sentez plus à l'aise. Sans doute, cela ne change pas la façon dont la famille 80x86 ou MASM réagissent à ces variables, mais vous permet de créer des programmes qui vous seront plus faciles à lire et à comprendre, puisque les noms des types sont plus indicatifs de la nature des données sous-jacentes.

Notez que CodeView respecte encore ces types. Si vous définissez *integer* comme étant un type sword, CodeView les affiche sous forme d'entiers non signés. D'autre part, si vous définissez *float* pour vous référer à real4, CodeView affichera encore ces variables comme des valeurs à virgule flottante de quatre octets.

5.5 Les données pointeur

Certaines personnes se réfèrent aux pointeurs comme à des types de données scalaires, d'autres comme à des types composés. Ce livre les considérera comme des types scalaires, même s'ils ont la tendance à avoir les deux caractéristiques (pour une description complète des types composés, voir "Types de données composés", à la prochaine section).

Bien sûr, la première chose à se demander est « qu'est-ce que c'est un pointeur ? ». Probablement vous avez déjà fait l'expérience des pointeurs dans des langages comme Pascal, C ou Ada, et vous commencez peut-être à vous inquiéter. Presque tout le monde se souvient de ses très mauvaises expériences lors de sa première rencontre avec les pointeurs dans des langages de haut niveau. Eh bien, ce n'est pas le cas ici ! Les pointeurs sont plus faciles à comprendre en assembleur. De plus, tous les problèmes que vous avez pu avoir avec les pointeurs n'ont probablement rien à voir avec ces derniers, mais plutôt avec les listes chaînées ou les arbres que vous essayez de construire avec les pointeurs. D'autre part, les pointeurs ont beaucoup d'utilisations en assembleur qui n'ont rien à voir avec les listes chaînées, les arbres ou d'autres structures de données effrayantes. Souvent, de simples structures comme les tableaux ou les enregistrements impliquent l'usage des pointeurs. Donc, si les pointeurs vous ont toujours traumatisé, oubliez tout ce que vous savez à propos d'eux ; vous allez connaître leur intérêt réel.

Le meilleur point de départ est probablement la définition d'un pointeur. Qu'est-ce que c'est enfin ? Malheureusement, les langages de haut niveau comme Pascal tendent à cacher la simplicité des pointeurs derrière un mur d'abstraction. Cette complexité additionnelle (qui peut pourtant exister pour de bonnes raisons), inhibe les programmeurs parce qu'*ils ne comprennent pas ce qui se passe réellement.*

Maintenant, si vous avez peur des pointeurs, allons les ingorer pour l'instant et travaillons avec de simples tableaux. Considérez la déclaration suivante en Pascal :

```
M: array [0..1023] of integer;
```

Même si vous ne connaissez pas Pascal, le concept ici est raisonnablement facile à comprendre. M est un tableau (*array* en anglais) qui contient 1024 entiers, indexés de M[0] à M[1023]. Chacun de ces éléments peut stocker un entier qui est indépendant de tous les autres. En d'autres termes, ce tableau vous donne 1024 variables entières ; vous utilisez chacune d'elles via un numéro (l'index) plutôt que via un nom.

Si vous trouvez un programme ayant l'instruction M[0] := 100 vous n'allez pas avoir probablement des difficultés à comprendre la finalité de cette instruction : elle stocke la valeur 100 dans le premier élément[10] du tableau M. Maintenant, considérez les deux instructions suivantes :

```
i := 0; (*Supposez qu'"i" est une variable de type entier *)
M[i] := 100;
```

Vous conviendrez, sans beaucoup d'hésitations, que ces deux instructions effectuent exactement la même opération que M[0] := 100;. Et vous serez certainement d'accord aussi que toute expression entière de la plage 0...1023 est un index de ce tableau. Le bloc d'instructions qui suit effectue *encore* la même opération que notre simple affectation pour l'index zéro :

```
i := 5;          (*Supposez que toutes les variables sont des entiers*)
j := 10;
k := 50;
M[i*j-k] := 100;
```

« D'accord, mais quel est le point ? », vous penseriez peut-être. « Tout ce qui produit un entier de la plage 0...1023 est correct. Et alors ? ». D'accord, que diriez-vous de ceci :

```
M[1] := 0;
M[ M[1] ] := 100;
```

Whoa ! Maintenant il vous faudra quelque moment pour digérer. Cependant, si vous y allez lentement, ça commence à avoir du sens et vous découvrirez que ces deux instructions effectuent encore la même opération que dans l'exemple précédent. La première instruction stocke zéro dans l'élément M[1] du tableau. La seconde instruction charge la valeur de M[1], qui est un entier que vous pouvez utiliser comme index du tableau M et utilise sa valeur (zéro) pour placer la valeur 100 dans l'élément choisi.

Si vous trouvez raisonnable ce que vous venez de lire, peut-être bizarre, mais néanmoins utilisable, alors vous n'avez pas de problèmes avec les pointeurs. *Parce que M[1] est un pointeur !* Enfin, pas tout à fait, mais si vous substituez M par la mémoire et vous considérez ce tableau comme son contenu, alors c'est l'exacte définition d'un pointeur.

Un pointeur est simplement un emplacement de mémoire dont la valeur est l'adresse (ou l'index, si vous voulez) d'un autre emplacement de mémoire. Les pointeurs sont très faciles à déclarer et à utiliser dans un programme en assembleur. Vous n'avez même pas à vous préoccuper des index ou toute autre chose semblable. En fait, la seule complication que vous rencontrerez est que la technologie 80x86 supporte deux sortes de pointeurs : les pointeurs *near* (proches) et *far* (éloignés).

Un pointeur near est une valeur de 16 bits qui fournit un offset dans un segment. Il pourrait s'agir de tout segment, mais vous utiliserez généralement le segment de données (qui s'appelle *dseg* dans SHELL.ASM). Si vous avez une variable word *p* qui contient 1000h, alors *p* "pointe" sur l'adresse 1000h dans le segment de données. Pour accéder au mot sur lequel *p* pointe, vous pourriez utiliser un code comme le suivant :

```
mov     bx, p           ;Charge BX avec le pointeur
mov     ax, [bx]        ;Obtient la donnée pointée par p (indirection)
```

En récupérant la valeur de p dans bx, ce code charge la valeur 1000h dans bx (en présumant que p contient 1000h et, par conséquent, pointe à l'emplacement de mémoire 1000h dans dseg). La seconde instruction charge le registre ax avec le mot qui commence à l'emplacement dont l'offset apparaît dans bx. Puisque bx contient maintenant 1000h, ceci chargera ax des emplacements DS:1000 et DS:1001.

Pourquoi ne pas charger le registre ax directement avec l'adresse 1000h à l'aide d'une instruction comme mov ax, ds:[1000h] ? Eh bien, il y a beaucoup de raisons. Mais la plus importante est que cette instruction charge *toujours* le registre ax de l'emplacement 1000h. Sauf si vous recourez à un code automodificateur, vous ne pourrez pas changer l'emplacement que ax charge[11]. Les deux instructions précédentes, au contraire, chargent toujours ax avec l'emplacement pointé par *p*. Et la valeur de *p* est très facile à changer sous le contrôle du programme, sans utiliser de code automodificateur. En fait, la simple instruction mov p, 2000h fera en sorte que les deux instructions ci-dessus puissent charger ax de l'offset DS:2000 la prochaine fois qu'elles exécuteront. Considérez les instructions suivantes :

[10]Ou la première variable, si l'on continue avec l'analogie n.d.t.
[11]On peut donc considérer les pointeurs comme des modes d'adressage variables, n.d.t.

```
        lea     bx, i           ;Ceci peut maintenant être fait avec une seule
        mov     p, bx           ;instruction, comme vous le verrez au chapitre
         .                      ;huit.
         .
;Supposez ici la présence d'un code qui fait passer le contrôle après les
;deux instructions suivantes.

        lea     bx, j           ;le code ci-dessus a sauté ces deux instructions
        mov     p, bx           ;et on arrive ici depuis autre part
         .
         .
        mov     bx, p           ;Supposez que les deux chemins de code
        mov     ax, [bx]        ;finissent par aboutir ici
```

Ce court exemple démontre deux chemins différents d'exécution du programme. Le premier charge la variable *p* avec l'adresse de la variable *i* (souvenez-vous que lea charge bx avec l'offset de la seconde opérande), le second charge p avec l'adresse de la variable *j*. Les deux convergent vers les deux dernières instructions mov, qui chargent ax avec *i* ou avec *j*, selon l'instruction qui a été exécutée. En beaucoup d'aspects, ceci est comme un *paramètre* de procédure dans un langage de haut niveau comme Pascal. Dans l'exécution, les mêmes instructions peuvent accéder à deux variables différentes selon l'adresse (*i* ou *j*) qui a été chargée dans *p*.

Les pointeurs near de 16 bits sont plus petits et plus rapides et les processeurs 80x86 fournissent un accès efficace en les utilisant. Malheureusement, ils ont un très sérieux inconvénient : on ne peut accéder qu'à 64Ko de données[12]. Les pointeurs far n'ont pas cet inconvénient, mais leur taille est de 32 bits. Ces pointeurs vous permettent d'accéder par contre à toute donnée n'importe où dans l'espace de mémoire. Pour cette raison et pour le fait que la bibliothèque UCR standard utilise exclusivement des pointeurs far, ce livre se serivra de ces pointeurs presque tout le temps. Mais notez c'est une décision motivée pour garder la simplicité. Le code utilisant les pointeurs near reste plus petit et plus rapide.

Pour accéder à une donnée référencée par un pointeur de 32 bits, vous aurez besoin de charger la portion offset (le mot le moins significatif) du pointeur dans *bx*, *bp*, *si* ou *di* et la portion segment dans un registre de segment (typiquement *es*). Puis, vous pouvez accéder à l'objet via le mode d'adressage indirect par les registres. Étant donné que l'instruction *les* est si pratique pour cette opération, c'est le meilleur choix pour charger *es* en combinaison avec un des quatre registres ci-dessus. Le code exemple suivant stocke la valeur de *al* dans l'octet pointé par le pointeur far *p* :

```
        les     bx, p           ;Charge p dans ES:BX
        mov     es:[bx], al     ;Stocke al dans l'emplacement désigné par le pointeur
```

Puisque les pointeurs near sont de 16 bits et les pointeurs far sont de 32 bits, vous pouvez simplement utiliser les directives dw/word et dd/dword pour allouer de l'espace pour ces pointeurs (les pointeurs sont essentiellement non signés, donc vous n'utiliserez pas sword ou sdword pour déclarer un pointeur).

Cependant, il y a un meilleur moyen de faire ceci, en utilisant l'instruction typedef. Considérez la forme générale suivante :

```
nearptr         typedef near ptr typedebase
farptr          typedef far ptr typedebase
```

Dans ces deux exemples *nearptr* et *farptr* représentent le nom du nouveau type que vous êtes en train de créer et *typedebase* est le type vers lequel vous voulez créer un pointeur. Regardons quelques exemples spécifiques :

```
nbytptr         typedef near ptr byte
fbytptr         typedef far ptr byte
colorsptr       typedef far ptr colors
wptr            typedef near ptr word
intptr          typedef near ptr integer
intHandle       typedef near ptr intptr
```

(dans ces déclarations, on suppose que vous avez d'abord défini les types *colors* et *integer* avec une instruction typedef). Les instructions typedef avec *near ptr* comme opérande produisent des pointeurs near de 16 bits. Les

[12]Techniquement, ce n'est pas vrai. Un pointeur est limité à l'accès des données à un segment particulier à la fois, mais vous pouvez avoir différents pointeurs near, chacun desquels pointe à des données en différents segments. Malheureusement, vous devez garder trace de tous ces pointeurs et vous pouvez rapidement vous perdre à mesure que le nombre de pointeurs augmente.

instructions avec *far ptr* produisent des pointeurs far de 32 bits. MASM 6.x ignore le type de base fourni après les mots-clés *near ptr* ou *far ptr*. Cependant, CodeView utilise ce type pour afficher correctement l'objet vers lequel le pointeur pointe.

Notez que vous pouvez utiliser n'importe quel type comme type de base pour un pointeur. Comme le montre le dernier exemple ci-dessus, vous pouvez même définir un pointeur vers un autre pointeur (un *handle*, c'est-à-dire un *gestionnaire*, comme on pourrait dire en français). CodeView affichera l'objet comme une variable de type intHandle qui pointe sur une adresse.

Avec les types qu'on vient de créer, on peut maintenant générer des variables pointeur comme suit :

```
bytestr         nbytptr         ?
bytestr2        fbytptr         ?
CouleurCourante colorsptr       ?
itemCourant     wptr            ?
DernierInt      intptr          ?
```

Vous pouvez naturellement initialiser ces pointeurs pendant l'assemblage si vous savez où ils pointeront quand le programme exécute. Par exemple, vous pourriez initialiser la variable *bytestr* avec l'offset de *MaChaine* en utilisant la déclaration suivante :

```
bytestr         nbytptr MaChaine
```

5.6 Types de données composés

Les types de données composés sont ces types qui sont construits à partir d'autres types (généralement scalaires). Un tableau est un bon exemple de ce type de données : c'est un agrégat d'éléments ayant tous le même type. Notez qu'un type composé n'est pas nécessairement composé de types scalaires, il y a, par exemple, des tableaux de tableaux, mais, à la fin, vous pouvez décomposer un type composé en ses éléments primitifs scalaires.

Cette section couvrira deux des types composés le plus communs : les tableaux et les enregistrements. Il est quelque peu prématuré de mentionner des types de données encore plus complexes.

5.6.1 Tableaux

Les tableaux (*arrays*) sont probablement le type de données composé le plus utilisé. Et pourtant, beaucoup de programmeurs débutants ont une très faible compréhension de leur fonctionnement, ainsi que des moyens de les remplacer plus efficacement. Il est surprenant de voir comment beaucoup de programmeurs novices (et parfois avancés !) considèrent les tableaux d'une perspective complètement différente une fois qu'ils ont appris à les manipuler au niveau de la machine.

D'une façon abstraite, un tableau est un type de données agrégé dont les membres (éléments) sont tous du même type. La sélection d'un membre de tableau se fait par un index entier[13]. Différents index sélectionnent des éléments uniques. Ce texte suppose que les index entiers sont contigus (même si ce n'est pas du tout obligatoire). C'est-à-dire, si le nombre *x* est un index valide dans le tableau et *y* est aussi un index valide, avec x < y, alors tout *i* qui satisfait x < i < y est un index valide du tableau.

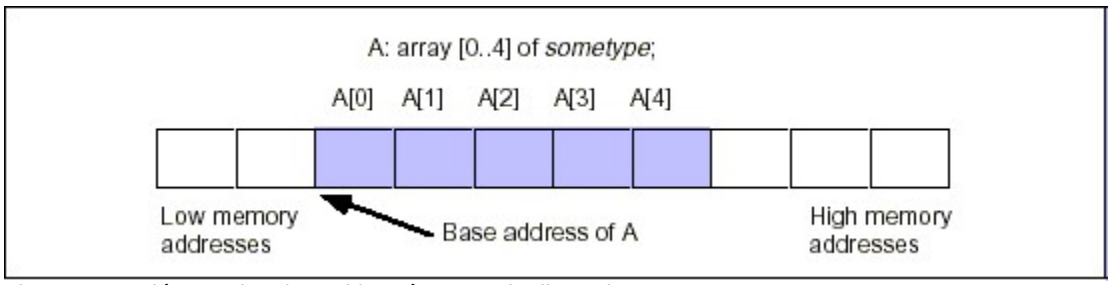

Figure 5.1 Implémentation d'un tableau à une seule dimension

[13]Ou quelque valeur dont la représentation sous-jacente est un entier, comme les types caractère, énumération ou booléen.

Quand vous appliquez l'opérateur d'indexage à un tableau, le résultat est l'élément spécifique du tableau choisi par cet index. Par exemple, A[i] choisit le $i^{ème}$ élément du tableau A. Notez qu'il n'y a pas une exigence formelle que l'élément i doive être de quelque manière proche de l'élément i+1 en mémoire. Tant que A[i] fait toujours référence au même emplacement et A[i + 1] se réfère toujours à l'emplacement suivant (et que les deux sont différents), la définition de tableau est satisfaite.

Dans ce livre, les tableaux occupent des emplacements de mémoire contigus. Un tableau de cinq éléments apparaît en mémoire comme à la figure 5.1.

L'adresse de base d'un tableau est l'adresse du premier élément et apparaît toujours à l'emplacement de mémoire le plus bas. Le second élément suit directement le premier, le troisième suit le second, etc. Notez qu'il n'y a pas d'obligation que les index commencent à zéro. Ils peuvent commencer à partir de n'importe quel numéro, à condition d'être contigus. Cependant, dans le cadre de notre discussion, c'est plus facile d'expliquer l'accès aux tableaux si le premier index est zéro. Ce livre fait commencer généralement la plupart des tableaux à zéro, à moins qu'il n'y ait une bonne raison de faire autrement. Cependant, ceci est requis seulement pour la cohérence de l'explication. Il n'y a pas de bénéfices d'efficacité sur le fait de faire commencer les tableaux à zéro ou à une autre valeur.

Pour accéder à un élément d'un tableau, on a besoin d'une fonction convertissant un index en l'adresse de l'élément indexé. Pour une tableau à une dimension, cette fonction est très simple :

```
Adresse_Element = AdressedeBase + ((Index - Index_Initial) * Taille_Element)
```

où *Index_Initial* est la valeur du premier index du tableau (valeur que vous pouvez ignorer si c'est zéro), et la valeur *Taille_Element* est la taille, en octets, d'un élément individuel du tableau.

5.6.1.1 Déclarer des tableaux dans le segment de données

Avant d'avoir accès à des éléments d'un tableau, il faut réserver suffisamment d'espace mémoire pour ce tableau. Heureusement, les déclarations de tableaux se bâtissent sur les déclarations qu'on a vue jusqu'à présent. Pour allouer *n* éléments dans un tableau, il faut utiliser une déclaration comme la suivante :

```
nomtab          typedebase      n dup(?)
```

nomtab est le nom de la variable tableau et *typedebase* est le type d'un élément du tableau. Ceci alloue l'espace nécessaire pour cet objet. Pour obtenir l'adresse de base du tableau, utilisez simplement son nom, dans ce cas, la variable *nomtab*.

L'opérande *n dup (?)* indique à l'assembleur qu'il faut *dup*liquer l'objet à l'intérieur des parenthèses *n* fois. Puisqu'un point d'interrogation apparaît à l'intérieur des parenthèses, la définition ci-dessus crée *n* occurrences d'une valeur indéfinie (non initialisée). Maintenant, observons quelques exemples spécifiques :

```
tabcar          char    128 dup(?) ;array[0..127] of char
tabint          integer 8 dup(?)   ;array[0..7] of integer
taboct          byte    10 dup(?)  ;array[0..9] of byte
tabptr          dword   4 dup(?)   ;array[0..9] of dword
```

Les deux premiers exemples impliquent que vous ayez déjà déclaré les types char et integer avec *typedef*.

Ces exemples allouent tous de l'espace pour des tableaux non initialisés. Vous pouvez aussi initialiser tous les éléments à une valeur spécifique avec les déclarations suivantes :

```
RealTableau             real4   8 dup(1.0)
IntegerTableau          integer 8 dup(1)
```

Ces définitions créent deux tableaux avec huit éléments. La première définition initialise chaque valeur réelle de quatre octets à la valeur 1.0, la seconde déclaration initialise chaque entier à 1.

Ce mécanisme d'initialisation marche bien si vous voulez que chaque élément d'un tableau ait la même valeur. Et si vous vouliez initialiser chaque élément d'un tableau avec des valeurs (éventuellement) différentes ? Eh bien, cela se fait aussi facilement et voici une autre technique d'initialisation :

```
nom     type    valeur1, valeur2, valeur3, ..., valeur_n
```

Cette forme alloue *n* variables de type *type*. Elle initialise le premier élément à *valeur1*, le second élément à *valeur2*, etc. Donc, simplement en énumérant chaque valeur dans le champ de l'opérande, vous pouvez créer un tableau avec les valeurs initiales désirées. Dans le tableau d'entiers suivant, par exemple, chaque élément contient le carré de son index :

```
Carres  integer 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100
```

Si votre tableau a plus d'éléments que ceux qu'une ligne peut contenir, il y a plusieurs moyens de le continuer sur la ligne qui suit. Le moyen le plus simple est d'utiliser une autre instruction *integer*, mais, *sans étiquette* :

```
Carres  integer 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100
        integer 121, 144, 169, 196, 225, 256, 289, 324
        integer 361, 400
```

Une autre option, qui est mieux en certaines circonstances[14], c'est d'utiliser une barre oblique inversée à la fin de chaque ligne indiquant à MASM 6.x de continuer à lire les données à la prochaine ligne :

```
Carres  integer 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, \
                121, 144, 169, 196, 225, 256, 289, 324, \
                361, 400
```

Sans doute, si votre tableau comporte plusieurs milliers d'éléments, les taper tous ne sera pas une partie de palisir. La plupart des tableaux initialisés de cette façon ne comportent pas plus de 200 éléments, et, en général, beaucoup moins de 100.

Il vous reste à apprendre une technique finale d'initialisation. Considérez la déclaration suivante :

```
GrandTableau    word    256 dup(0,1,2,3)
```

Ce tableau a 1024 éléments, non 256. L'opérande n dup(xxxx) duplique xxxx *n* fois, ce qui diffère de créer un tableau à *n* éléments. Si xxxx est un seul item, alors l'opérateur dup créera un tableau de *n* éléments ; si xxxx contient deux objets séparés par une virgule, alors dup créera un tableau avec *2*n* éléments. Si xxxx contient trois éléments, les éléments du tableaux seront *3*n*, et ainsi de suite. Puisque dans la déclaration ci-dessus il y a quatre valeurs séparées par une virgule, alors l'opérateur dup crée 256*4 éléments - soit 1024 - dans le tableau. Les valeurs dans le tableau commenceront par 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3...

Vous verrez d'autres possibilités avec l'opérateur dup quand on traitera les tableaux multidimensionnels.

5.6.1.2 Accéder à des éléments dans un tableau unidimensionnel

Pour accéder à un élément d'un tableau dont le premier index est zéro, on peut utiliser la formule simplifiée :

Adresse_Element = Adresse_Base + index * Taille_Element

Pour Adresse_Base, on peut utiliser le nom du tableau (puisque MASM associe l'adresse de la première opérande avec l'étiquette). Taille_Element est la taille en octets de chaque élément. Si l'objet est un tableau d'octets, alors le champ Taille_Element vaut 1 (et le calcul de l'offset devient très simple). Si chaque élément est un mot (ou un entier ou tout autre type de deux octets), alors la valeur de Taille_Elements est 2. Et ainsi de suite. Pour accéder à un élément du tableau Carres du paragraphe précédent, la formule à adopter est :

```
Adresse_Element = Carres + index * 2
```

Le code 80x86 équivalent à l'instruction AX := Carres[index] est

```
mov     bx, index
add     bx, bx          ;Manière détournée de calculer 2*bx
mov     ax, Carres [bx]
```

Il y a deux choses importantes à noter ici. Avant tout, ce code utilise l'instruction add, au lieu de mul pour calculer 2*index. La raison principale est qu'elle est plus pratique (souvenez-vous que l'instruction mul ne fonctionne pas avec des constantes et elle travaille seulement sur le registre ax). Il s'avère que *add* est *beaucoup* plus rapide que *mul* sur beaucoup de processeurs ; cependant, la vitesse n'est pas la seule raison de préférer add à mul, malgré que vous ne saviez probablement pas ce fait.

[14] A l'avis du traducteur, c'est mieux dans toutes les circonstances, n.d.t.

La seconde raison que a motivé cette préférence est que la séquence d'instructions ci-dessus ne calcule pas de façon explicite la somme entre l'adresse de base et l'index multiplié par deux. Au lieu de cela, elle fait appel au mode d'adressage indexé qui, de façon implicite, effectue cette somme. L'instruction mov ax, Carres [bx] charge AX avec l'emplacement Carre+bx qui est l'adresse de base plus index*2 (étant donné que bx, contient index*2). C'est sûr que vous auriez pu utiliser

```
lea     ax, Carres
add     bx, ax
mov     ax, [bx]
```

à la place de la dernière instruction, mais pourquoi utiliser trois instructions quand une suffit ? Ceci illustre bien pourquoi vous devez connaître à fond les modes d'adressage. Choisir le meilleur peut réduire la taille de votre programme, et, par conséquent, le rendre plus rapide.

Le mode d'adressage indexé est le mode naturel pour accéder à des éléments de tableaux à une dimension. Sa syntaxe même suggère un accès à un tableau. La seule chose à garder à l'esprit est qu'il faut se souvenir de multiplier l'index par la taille d'un élément. Ne pas le faire produit des résultats incorrects.

Si vous avez un 80386 ou ultérieur, vous pouvez prendre avantage de son mode d'adressage scalaire, afin de rendre encore plus rapide l'accès à un élément de tableau. Considérez les instructions suivantes :

```
mov     ebx, index              ; En supposant une valeur de 32 bits
mov     ax, Carres [ebx*2]
```

Ceci réduit le code à deux instructions. Vous verrez bientôt que deux instructions ne sont pas nécessairement plus rapides que trois, mais vous avez l'idée. Savoir vos modes d'adressage peut sûrement aider.

Avant de passer aux tableaux à plusieurs dimensions, deux points supplémentaires sur les modes d'adressage et les tableaux sont au programme. Les séquences ci-dessus fonctionnent parfaitement si vous accédez seulement à un seul élément du tableau Carres. Cependant, si dans une section de code vous accédez à plusieurs éléments et vous pouvez vous permettre de consacrer un autre registre à l'opération, vous pouvez sans doute accélérer le processus et probablement le rendre plus rapide. L'instruction mov ax, Carres[BX] a une taille de quatre octets (en supposant que vous avez besoin d'un déplacement de deux octets pour stocker l'offset de Carres dans le segment de données). Vous pouvez réduire ceci à une instruction de deux octets via le mode d'adressage basé/indexé comme suit :

```
lea     bx, Carres
mov     si, index
add     si, si
mov     ax, [bx][si]
```

Maintenant, bx contient l'adresse de base et si contient la valeur index*2. Sans doute, ceci a remplacé une instruction de quatre octets par une de trois et une autre de deux, vraiment une bonne affaire. Cependant, vous n'avez pas à recharger bx avec l'adresse de base de Carres lors du prochain accès. La séquence suivante est d'un octet plus courte par rapport à la séquence qui ne charge pas l'adresse de base dans bx :

```
lea     bx, Carres
mov     si, index
add     si, si
mov     ax, [bx][si]
 .
 .                      ;Supposez que bx n'est pas modifié entre-temps
 .
mov     si, index2
add     si, si
mov     cx, [bx][si]
```

Sans doute, plus vous aurez d'accès à Carres sans recharger bx, plus grand sera le nombre d'octets économisés. Recourir à des astuces comme celle-ci peut être parfois très pratique. Néanmoins, les gains en octets dépendent entièrement de la machine que vous utilisez. Des séquences de codes qui s'exécutent plus rapidement sur un 8086 pourraient se révéler *plus lentes* sur un 80486 (et vice-versa). Malheureusement, si la vitesse est votre but, il n'y a pas de recette universelle. Il est très difficile de prédire la vitesse de la plupart des instructions sur un simple 8086, et il l'est encore plus sur des processeurs comme le 80486 ou Pentium/80586, capables d'offrir des pipelines, des caches intégrés sur puce ou même des opérations superscalaires.

5.6.2 Tableaux multidimensionnels

Le matériel 80x86 peut facilement gérer les tableaux à une dimension. Malheureusement, il n'y a pas de mode d'adressage magique vous permettant d'accéder aisément à des éléments d'un tableau multidimensionnel ; cela prend du travail et beaucoup d'instructions.

Avant d'aborder la façon de déclarer des tableaux à plusieurs dimensions et d'y accéder, il serait une bonne idée de se représenter comment on les implémente en mémoire. Le premier problème est concevoir comment stocker un objet multidimensionnel dans l'espace de mémoire, qui est unidimensionnel.

Considérez pour un moment un tableau en Pascal de la forme A:array[0..3,0..3] of char. Ce tableau contient 16 octets organisés en quatre rangées de quatre caractères. D'une manière ou d'une l'autre, il faut établir une correspondance entre chacun des 16 octets de ce tableau et les 16 octets contigus dans la mémoire principale. La figure 5.2 montre un moyen de le faire.

Le référencement réel importe peu si les deux conditions suivantes sont remplies : (1) chaque élément fait référence à un emplacement de mémoire unique (c'est-à-dire, il n'y a pas deux éléments qui occupent la même adresse) et (2) le référencement est cohérent : un élément donné du tableau fait toujours référence à la même adresse. Donc, ce dont on a réellement besoin est une fonction avec deux paramètres d'entrée (rangée et colonne) produisant un offset dans un tableau linéaire de 16 octets.

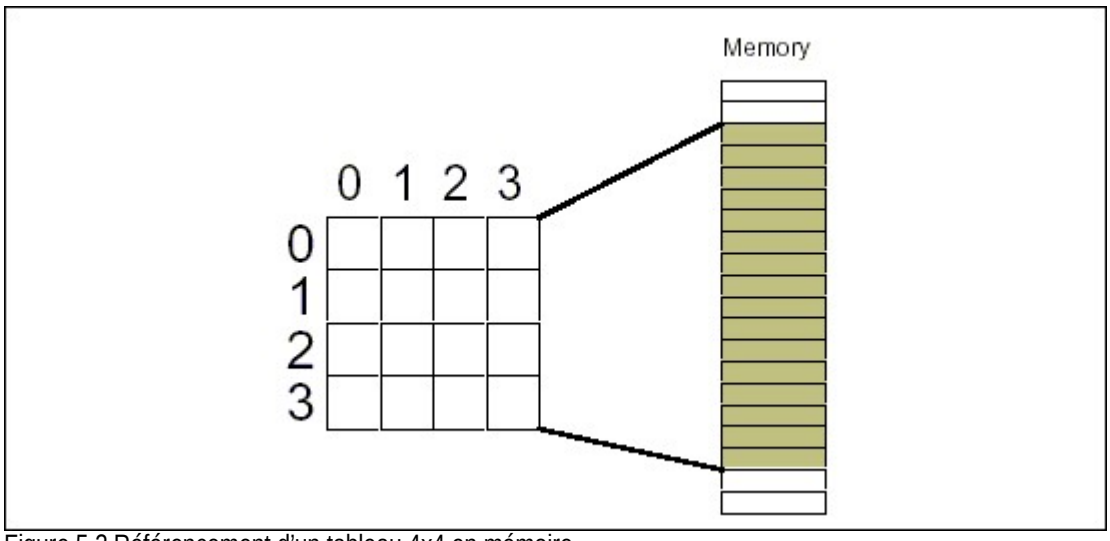

Figure 5.2 Référencement d'un tableau 4x4 en mémoire

Or, toute fonction qui satisfait les conditions ci-dessus est valide. Vous pouvez même choisir un adressage aléatoire tant qu'il est unique. Cependant, ce que vous voulez réellement est un référencement efficace à calculer pendant l'exécution et qui fonctionne avec des tableaux de toute taille (non seulement pour un tableau 4x4 ou même un tableau limité à deux dimensions). Bien qu'il y ait beaucoup de fonctions qui font l'affaire, il y en a deux en particulier que la plupart des programmeurs (et des langages de haut niveau) utilisent : *mode orienté rangée (row major ordering)* et *mode orienté colonne (column major ordering)*.

5.6.2.1 Mode orienté rangée

Ce mode assigne des éléments successifs à des emplacements de mémoire successifs, en parcourant chaque colonne d'une rangée avant de passer à la colonne successive. Ce procédé est mieux exposé par la figure 5.3.

Le mode orienté rangée est la méthode employée par la plupart des langages de programmation de haut niveau, en incluant Pascal, C, Ada, Modula-2, etc. Elle est très facile à implémenter et à utiliser dans un langage machine (spécialement avec un débogueur comme CodeView). La conversion d'une structure bidimensionnelle en un tableau linéaire est très intuitive. Vous commencez par la première rangée (la rangée numéro zéro) et puis

vous chaînez la seconde à la fin de la première. Ensuite, vous chaînez la troisième à la fin de la liste d'éléments et ainsi de suite (voir figure 5.4).

Pour ceux qui aiment penser en termes de code de programmation, la boucle imbriquée suivante en Pascal montre comment fonctionne le mode orienté rangée :

```
index := 0;
for colindex := 0 to 3 do
        for rangindex := 0 to 3 do
        begin
                mem[index] := orientrang[colindex][rangindex];
                index := index + 1;
        end ;
```

La chose importante à noter ici - qui est propre du mode orienté rangée, peu importe le nombre de dimensions - est que l'index le plus à droite augmente le plus rapidement. C'est-à-dire, lorsque vous allouez des emplacements de mémoire successifs, vous incrémentez cet index jusqu'à la fin de la rangée. Chaqe fois qu'on atteint la fin d'une rangée, l'index de gauche s'incrémente de 1 et l'index de droite recommence à zéro. Le principe est le même pour n'importe quel nombre de dimensions[15].

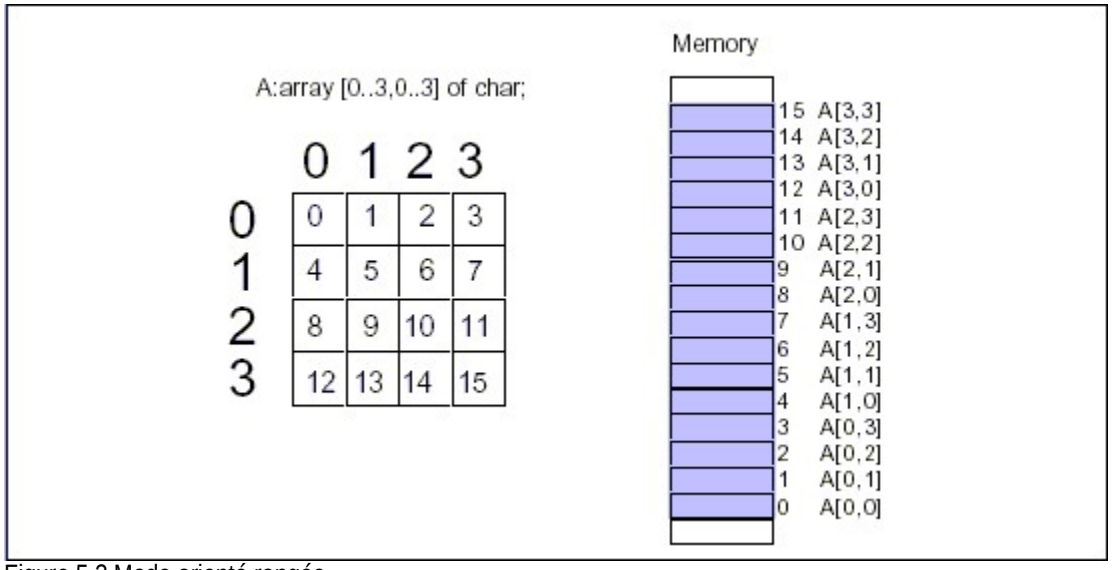

Figure 5.3 Mode orienté rangée

L'exemple suivant en Pascal montre un mode orienté rangée pour un tableau 4x4x4 :

```
index := 0;
for profindex := 0 to 3 do
        for colindex := 0 to 3 do
                for rangindex := 0 to 3 do begin
                    mem[index] := orientrang[profindex][colindex][rangindex];
                    index := index + 1;
end ;
```

La fonction réelle convertissant une liste de valeurs d'index en offsets n'implique pas de boucle ou d'autres calculs élaborés, mais une simple modification de la formule calculant l'adresse d'un élément de tableau à une seule dimension. L'équation permettant de calculer l'offset d'un tableau à deux dimensions, ordonné par rangée et déclaré comme *A:array[0..3,0..3] of integer* est :

adresse_element = adresse_base + (colindex + taille_rang * rangindex) * taille_element

[15]Pensez à la machine de Pascal, ou aux calculatrices des premières générations : les unités incrémentent le plus rapidement et quand une unité dépasse 9, les dizaines s'incrémentent et les unités recommencent de zéro et ainsi de suite. Le nombre de dimensions d'un tableau s'appelle son *degré*, n.d.t.

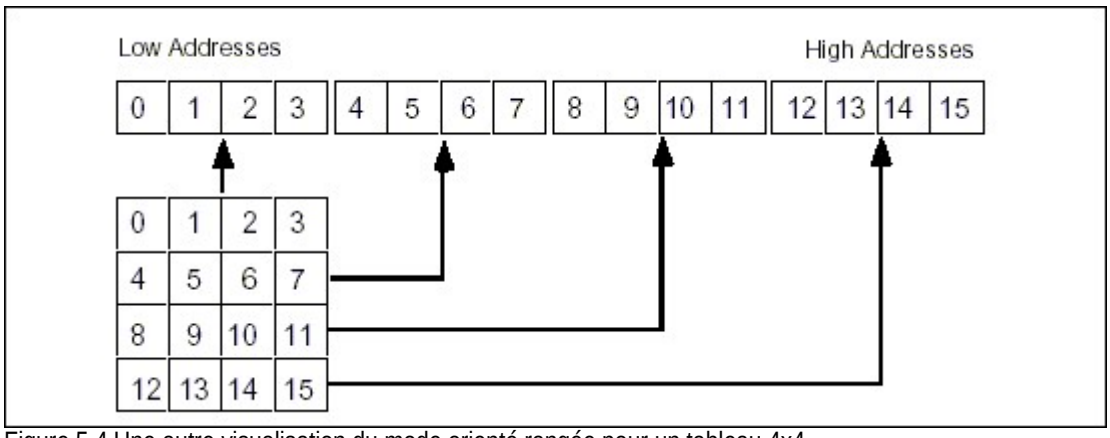

Figure 5.4 Une autre visualisation du mode orienté rangée pour un tableau 4x4

Comme d'habitude, adresse_base est l'adresse du premier élément du tableau (A[0][0] dans ce cas) et taille_element est la taille d'un élément individuel en octets. colindex est l'index de gauche, rangindex est l'index de droite (rangindex représente chaque rangée et colindex chaque élément d'une rangée, taille_rang est le nombre d'éléments de chaque rangée du tableau (quatre dans ce cas, puisque chaque ligne a quatre éléments)). En supposant que taille_element est 1, cette formule calcule les offsets suivants à partir de l'adresse de base :

Index colonne	Index rangée	Offset du tableau
0	0	0
0	1	1
0	2	2
0	3	3
1	0	4
1	1	5
1	2	6
1	3	7
2	0	8
2	1	9
2	2	10
2	3	11
3	0	12
3	1	13
3	2	14
3	3	15

Pour un tableau à trois dimensions, la formule pour calculer l'offset d'un élément en mémoire est :

Adress = Base + ((profindex * taille_colonne + colindex) * taille_rang + rangindex) * taille_element

taille_colonne est le nombre d'éléments pour chaque colonne et taille_rang est le nombre d'éléments pour chaque rangée. En Pascal, si vous avez déclaré un tableau "A:array[i..j][k..l][m..n] of type;", taille_rang est égale à n-m+1 et taille_colonne à l-k+1.

Pour un tableau de quatre dimensions, déclaré "A:array[g..h][i..j][k..l][m..n] of type;" la formule pour calculer l'adresse d'un élément est

Adresse = Base + (((index_gauche * taille_prof + profindex) * taille_colonne + colindex) * taille_rang + rangindex) * taille_element

taille_rangee est égale à i-j+1, taille_colonne et taille_ligne sont comme précédemment. index_gauche représente la valeur de l'index se trouvant à l'extrême gauche.

Maintenant, vous êtes probablement en train de commencer à voir un modèle. Il y a une formule générique permettant de calculer l'offset en mémoire pour un élément de tableau ayant *tout* nombre de dimensions ; cependant, vous utiliserez rarement plus de quatre dimensions.

Un autre moyen commode de se représenter un tableau ordonné par rangées est le tableau de tableau. Considérez la définition suivante d'un tableau à une seule dimension :

```
A: array[0..3] of type_x;
```

Présumez que *type_x* est un tableau de caractères.

A est un tableau à une dimension. Ses éléments individuels se trouvent être eux-mêmes des tableaux, mais vous pouvez ignorer ce fait pour le moment. La formule pour calculer l'adresse d'un élément serait

```
Adresse_Element = Base + Index * Taille_Element
```

Dans ce cas, Taille_Element est égale à 4, puisque chaque élément de A est un tableau de quatre caractères. Donc, qu'est-ce que la formule calcule ? Elle calcule l'adresse de base de chaque rangée dans ce tableau 4x4 de caractères (voir Figure 5.5).

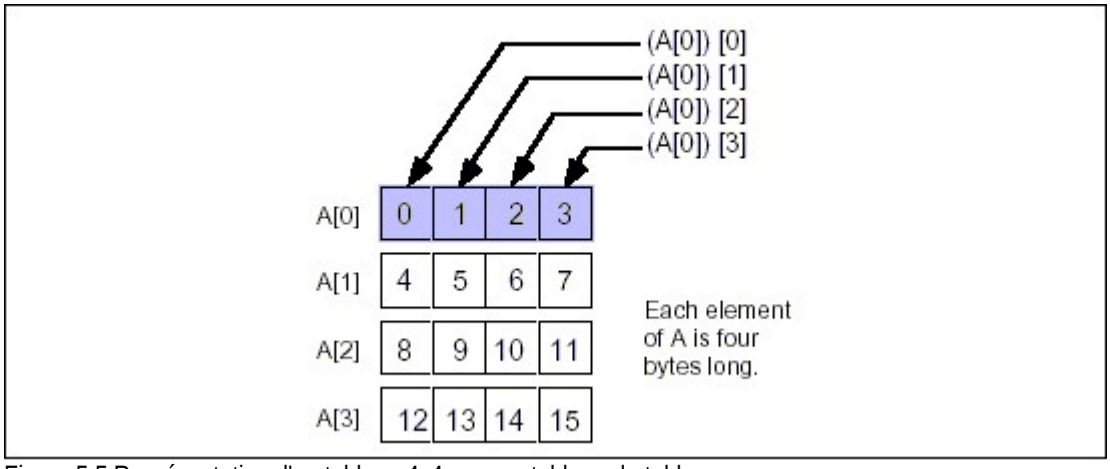

Figure 5.5 Représentation d'un tableau 4x4 comme tableau de tableau

Naturellement, une fois que vous obtenez l'adresse de base d'une rangée, vous pouvez réappliquer la même formule pour obtenir l'adresse d'un élément particulier. Alors que ceci n'affecte pas du tout la réussite du calcul, conceptuellement c'est peut-être un peu plus facile d'avoir affaire à plusieurs formules pour une seule dimension, plutôt que d'avoir recours au calcul complexe des adresses d'éléments des tableaux multidimensionnels.

Considérez un tableau Pascal défini comme "A:array[0..3][0..3][0..3][0..3] of char;" Vous pouvez considérer ce tableau à cinq dimensions comme un tableau de tableaux à une dimension :

```
type
        UneD = array[0..3] of char;
        DeuxD = array[0..3] of UneD;
        TroisD = array[0..3] of DeuxD;
        QuatreD = array[0..3] of TroisD;
var
        A: array[0..3] of QuatreD;
```

La taille de UneD est de quatre octets. Puisque DeuxD contient quatre tableaux UneD, sa taille est de 16 octets. De même, TroisD est composé par quatre DeuxD, donc il mesure 64 octets. Finalement, QuatreD est fait de quatre TroisD, donc, il est long 256 octets. Pour calculer l'adresse de "A[b][c][d][e][f]" il faudrait passer par les étapes suivantes :

- Calculer l'adresse de A[b] en tant que "Base + b * taille". Ici taille est de 256 octets. Utiliser ce résultat comme nouvelle adresse de base dans le prochain calcul.
- Calculer l'adresse de A[b][c] par la formule "Base + c * taille", où *Base* est la valeur obtenue ci-dessus et *taille* est 64. Utiliser le résultat comme nouvelle base pour le prochain calcul.

- Calculer l'adresse de A[b][c][d] via "Base + d * taille", où *Base* est obtenue du calcul précédent et *taille* est maintenant 16.
- Calculer l'adresse de A[a][b][c][d][e] par la formule "Base + e * taille" où *Base* vient du précédent calcul et *taille* vaut maintenant 4. Utiliser ce résultat comme base pour le prochain calcul.
- Finalement, calculer l'adresse de A[a][b][c][d][e][f] en utilisant la formule "Base + f * taille", où *Base* est le résultat obtenu lors du précédent calcul et *taille* est 1 (multiplication finale que vous pouvez évidemment ignorer). Le résultat obtenu est l'adresse de l'élément désiré.

Non seulement ce schéma est plus facile à utiliser que la formule élaborée précédemment, mais il est aussi plus facile à calculer (une seule boucle suffit). Supposez que vous avez deux tableaux initialisés comme suit

A1 = {256, 64, 16, 4, 1} et A2 = {b, c, d, e, f}

alors, le code Pascal qui effectue le calcul de l'adresse d'un élément devient :

```
for i := 0 to 4 do
        base := base + A1[i] * A2[i];
```

On présume que *base* contient l'adresse de base du tableau avait de l'exécution de la boucle. Notez que vous pouvez élargir ce code à tout nombre de dimensions en initialisant simplement A1 et A2 de manière appropriée et en changeant la valeur de fin de la boucle.

Cependant, il s'avère que la surcharge de calcul pour de telles boucles est trop grande pour être considérée en pratique. Vous pourriez utiliser un tel algorithme seulement si vous êtes capables de spécifier le nombre de dimensions pendant l'exécution. En fait, l'une des raisons principales pour lesquelles vous ne trouverez pas de tableaux avec un plus grand nombre de dimensions en assembleur est que l'assembleur démontre le manque d'efficacité associé à de tels accès. Il est facile d'écrire quelque chose comme "A[b, c, d, e, f]" en Pascal, sans réaliser ce que le compilateur est en train de faire avec le code. Les programmeurs en assembleur ne sont pas aussi cavaliers, ils se rendent compte immédiatement de la confusion qui découle des tableaux avec beaucoup de dimensions et ont souvent recours à des astuces pour accéder aux donnes dans de tels tableaux quand leur utilisation s'impose. On reparlera de ceci un peu plus loin dans ce chapitre.

5.6.2.2 Mode orienté colonne

Le mode orienté colonne reflète l'autre formule fréquemment utilisée pour calculer l'adresse d'un élément de tableau. Le FORTRAN et certains dialectes du BASIC (par exemple, celui de Microsoft), se servent de cette méthode pour indexer des tableaux.

Dans le mode orienté rangée, les index qui se trouvent le plus à droite, augmentent le plus rapidement à mesure qu'on avance dans des adresses consécutives. Dans le mode orienté colonne, par contre, ce sont les index le plus à gauche qui s'incrémentent le plus rapidement. De manière picturale, un tel mode est organisé comme le montre la Figure 5.6.

La formule pour trouver l'adresse d'un élément dans ce mode est très similaire à celle utilisée dans le mode orienté rangée. Il faut simplement inverser les index et les tailles :

Pour deux dimensions :

adresse_element = adresse_base + (index_raqngee + index_colonne * taille_colonne) * taille_element

où *index_rangee* correspond chaque rangée.

Pour trois dimensions :

Adresse = Base + ((index_rangee * taille_colonne + index_colonne) * taille_prof + index_prof) * taille_element

Pour quatre dimensions :

Adresse = Base + (((index_rangee * taille_colonne + index_colonne) * taille_prof + index_prof) * taille_gauche + index_gauche) * taille_element

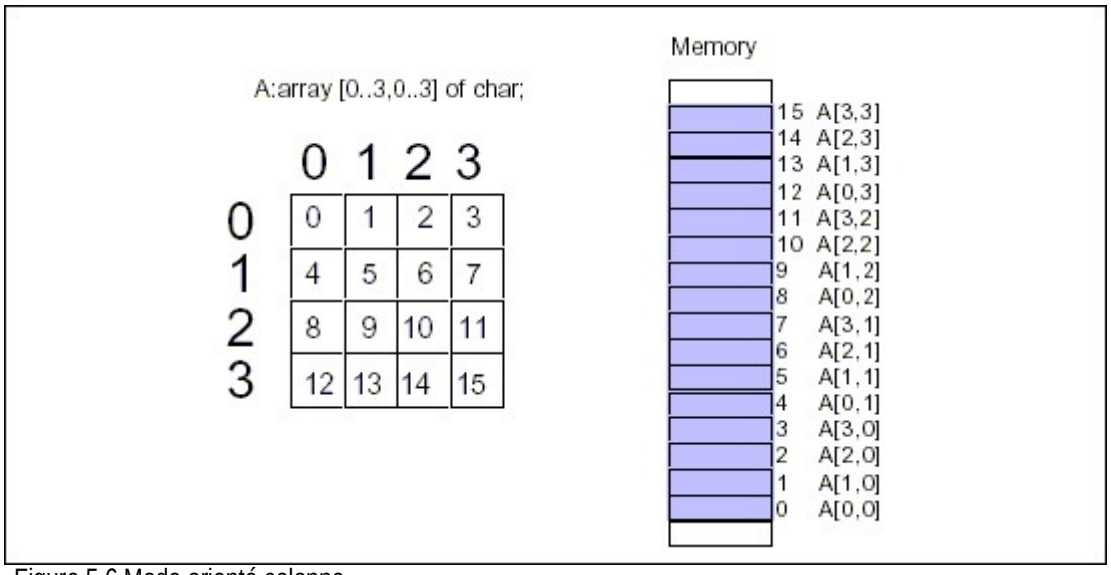

Figure 5.6 Mode orienté colonne

La boucle Pascal servant au mode orienté rangée reste inchangée (accès de A[b][c][d][e][f]) :

```
for i := 0 to 4 do
        base := base + A1[i] * A2[i];
```

De même, les valeurs initiales du tableau A1 restent inchangées :

```
A1 = {256, 64, 16, 4, 1}
```

La seule chose qui a besoin d'une modification, ce sont les valeurs initiales du tableau A2, et tout ce que vous avez à faire ici est de placer les index en ordre inverse :

```
A2 = {f, e, d, c, b}
```

5.6.2.3 Allouer de la mémoire pour des tableaux à plusieurs dimensions

Si vous avez un tableau *m x n* il aura *m * n* éléments et requerra *m*n*Taille_Element* d'espace. Pour allouer de la mémoire pour un tableau, vous devez réserver cette quantité de mémoire. Comme d'habitude, il y a plusieurs méthodes différentes pour accomplir cette tâche. Ce livre prendra en considération l'approche la plus facile à lire et à comprendre.

Vous souvenez-vous de l'opérateur dup, utilisé pour réserver de l'espace ? *n dup(xxxx)* duplique xxxx *n* fois. Comme vous avez vu précédemment, cet opérateur admet plus d'un élément à l'intérieur de ses parenthèses et il permet de dupliquer tout ce qu'il contient pour le nombre de fois spécifié. En fait, cet opérateur admet *absolument tout* ce que vous pouvez trouver dans le champ opérandes d'une instruction byte, y compris *d'autres occurrences additionnelles de l'opérateur dup*. Autrement dit, dup est imbricable. Veuillez considérer l'instruction qui suit :

```
A       byte    4 dup(4 dup(?))
```

Le premier opérateur dup répète quatre fois tout ce qui se trouve à l'intérieur des parenthèses. Dans ces dernières, 4 dup(?) indique à MASM de réserver de l'espace pour quatre octets. Quatre copies de quatre octets font seize octets, le nombre nécessaire pour un tableau 4x4. Sans doute, vous auriez pu faire la même chose avec

```
A       byte    16 dup(?)
```

en obtenant encore une fois les 16 octets contigus nécessaires à vos besoins. Pour ce qui concerne le microprocesseur, il n'y a aucune différence entre ces deux formes. Mais, d'autre part, la première version donne

au programmeur une meilleure indication que A est un tableau 4x4, alors que l'autre ferait plutôt penser à un tableau à une seule dimension avec 16 éléments.

Vous pouvez élargir ce concept à des tableaux de degré supérieur. La déclaration d'un tableau de trois dimensions du type *A:array[0..2, 0..3, 0..4] of integer* serait

```
A       integer 3 dup (4 dup (5 dup (?)))
```

(bien sûr, le type integer est censé d'avoir été précédemment créé avec l'opérateur typedef).

Comme c'était le cas avec les tableaux unidimensionnels, vous pouvez initialiser chaque élément du tableau à une valeur spécifique, en remplaçant le point d'interrogation par des valeurs particulières. Par exemple, pour initialiser le tableau ci-dessus, pour que chaque élément contienne la valeur 1, vous utiliseriez le code :

```
A      integer 3 dup (4 dup (5 dup (1)))
```

Si vous vouliez initialiser chaque élément avec une valeur différente, vous auriez à entrer chaque valeur individuellement. Si la taille d'une rangée est suffisamment petite, le meilleur moyen d'effectuer cette besogne est de placer les données de chaque rangée du tableau sur sa propre ligne. Considérez la déclaration suivante d'un tableau 4x4 :

```
A       integer         0, 1, 2, 3
        integer         1, 0, 1, 1
        integer         5, 7, 2, 2
        integer         0, 0, 7, 6
```

Encore une fois, l'assembleur ne se préoccupe pas de l'endroit où vous fractionnez les lignes, mais la représentation qu'on vient de voir est certainement plus facile à identifier comme un tableau 4x4 que ce qui suit, qui pourtant émet exactement les mêmes données :

```
        integer         0, 1, 2, 3, 1, 0, 1, 1, 5, 7, 2, 2, 0, 0, 7, 6
```

Sans doute, si vous avez un long tableau, un tableau avec des rangées vraiment longues, ou un tableau avec diverses dimensions, les espoirs de s'en sortir avec une représentation quelque peu raisonnable sont très réduits. C'est là que les commentaires particulièrement illustratifs deviennent vraiment pratiques.

5.6.2.4 Accéder aux éléments des tableaux multidimensionnels en assembleur

Maintenant que vous avez vu les formules pour calculer l'adresse d'un élément d'un tableau, vous êtes encore limité à quelques exemples en Pascal pour ce qui concerne l'accès effectif aux éléments des tableaux multidimensionnels. Maintenant, il est temps de voir comment le faire en langage assembleur.

Les instructions mov, add et mul transcrivent sans effort les formules calculant l'offset des éléments des tableaux à plusieurs dimensions. Considérez d'abord un tableau à deux dimensions :

```
; Notez que la largeur de chaque rangée de DeuxD est de 16 bits.

DeuxD           integer 4 dup (8 dup (?))
i               integer ?
j               integer ?

                .       .
                .       .
                .       .

; Pour effectuer l'operation DeuxD[i,j] := 5; il faut utiliser le code :
                mov     ax, 8   ;8 éléments par rangée
                mul     i
                add     ax, j
                add     ax, ax  ;Multiplier par la taille de l'élément (2)
                mov     bx, ax  ;Stockage dans un registre que l'on peut utiliser
                mov     DeuxD [bx], 5
```

Sans doute, si vous avez une puce 80386 (ou mieux), vous pourriez utiliser le code suivant[16] :

```
        mov     eax, 8  ;Les bits les plus significatifs de eax sont à zéro.
        mul     i
        add     ax, j
        mov     DeuxD[eax*2], 5
```

Notez que ce code ne requiert pas l'usage d'un mode d'adressage de deux registres. Même si un mode d'adressage comme DeuxD[bx][si] a l'air naturel pour l'accès d'un tableau à deux dimensions, ce n'est toutefois pas son but.

Maintenant, considérez un second exemple qui utilise un tableau à trois dimensions :

```
TroisD          integer 4 dup (4 dup (4 dup (?)))
i               integer ?
j               integer ?
k               integer ?
                .       .
                .       .
                .       .
; Pour effectuer l'opération TroisD[i,j,k] := 1; il faut utiliser ce code :
                mov     bx, 4   ;4 éléments par colonne
                mov     ax, i
                mul     bx
                add     ax, j
                mul     bx      ;4 éléments par ligne
                add     ax, k
                add     ax, ax  ;Multiplication par la taille de l'élément (2)
                mov     bx, ax  ;Stockage dans un registre que l'on peut utiliser.
                mov     TroisD[bx], 1
```

Sans doute, si vous disposez d'un processeur 80386 ou mieux, vous pourriez améliorer ce code de la façon suivante :

```
                mov     ebx, 4
                mov     eax, ebx
                mul     i
                add     ax, j
                mul     bx
                add     k
                mov TroisD[eax*2], 1
```

5.6.3 Structures

La deuxième principale structure de données composée est le *record* (enregistrement) en Pascal ou la *structure* en C[17]. La terminologie Pascal est probablement meilleure, puisqu'elle tend à éviter la confusion avec le terme plus générique *structure de données.* Cependant, MASM utilise "structure" et donc il est raisonnable de s'en tenir là. De plus, MASM utilise le terme *record* (enregistrement) pour quelque chose de légèrement différent, donc définitivement le choix du mot structure est approprié.

Là où un tableau est homogène - tous les éléments sont du même type - les éléments d'une structure peuvent être de types différents. Un tableau vous permet de sélectionner un élément particulier à l'aide d'un index entier, alors qu'avec les structures, vous devez sélectionner un *élément* (dit aussi *champ*) par son nom.

La finalité principale d'une structure est de permettre d'encapsuler des données différentes - mais logiquement liées - dans un seul ensemble. La déclaration d'un enregistrement Etudiant en Pascal est probablement l'exemple le plus classique :

```
Etudiant = record
        Non: string[64];
```

[16]En fait, il y a même une séquence d'instructions 80386 encore plus efficace, mais elle utilise des instructions qui n'ont pas encore été étudiées.

[17]Ceci peut avoir aussi d'autres noms dans d'autres langages de programmation, mais la plupart des gens reconnaît au moins un de ces noms.

```
        Licence: integer;
        SS: string[13];
        Examen1: integer;
        Examen2: integer;
        ExamenFinal: integer;
        Devoir: integer;
        Projets: integer;
end;
```

Beaucoup de compilateurs Pascal allouent, pour chaque membre d'un enregistrement, des emplacements de mémoire contigus ; dans l'exemple ci-dessus, Pascal allouera les premiers 65 octets pour le nom[18], les deux octets suivants pour le code de licence, les 14 suivants pour le numéro de sécurité sociale, etc.

En langage assembleur, vous pouvez aussi créer des types structure en utilisant la déclaration *struct* de MASM. Le code correspondant en assembleur est le suivant :

```
etudiant        struct
Nom             char    65 dup(?)
Licence         integer ?
SSN             char    12 dup(?)
Examen1         integer ?
Examen2         integer ?
ExamenFinal     integer ?
Devoir          integer ?
Projets         integer ?
etudiant        ends
```

Remarquez que la structure se termine par l'instruction ends (qui veut dire *end structure*). L'étiquette de ends doit être la même que celle de l'instruction struct.

Les noms des membres de la structure doivent être uniques, ce qui veut dire que le même nom ne peut pas paraître deux ou plusieurs fois dans la même structure. Cependant, tous les noms de champ ont une portée locale. Par conséquent, vous pouvez réutiliser les noms de ces membres ailleurs dans le programme[19].

La directive *struct* définit uniquement un type structure ; elle ne réserve pas d'espace pour une variable de type structure. Pour réserver cet espace, il faut déclarer une variable en utilisant le nom de la structure comme déclaration de MASM, par exemple :

```
Jean            etudiant        {}
```

Les accolades doivent apparaître dans l'opérande du champ. Toute valeur initiale doit paraître entre ces accolades. Cette déclaration réserve de la mémoire comme il est montré à la figure 5.7.

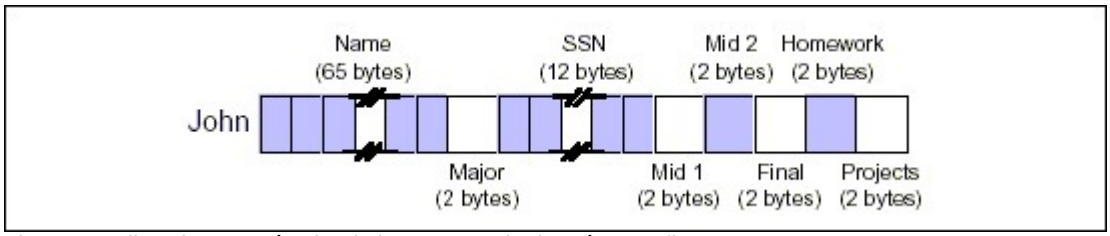

Figure 5.7 Allocation en mémoire de la structure de données *etudiant*

Si l'étiquette Jean correspond à l'adresse de base de cette structure, alors le champ Nom se trouve à l'offset Jean+0, le champ Licence est à l'offset Jean+65, le champ SS est à l'offset Jean+67 et ainsi de suite.

Pour accéder à un élément d'une structure, il faut connaître l'offset de cet élément par rapport au début de la structure. Par exemple, le membre Licence de la variable Jean est à l'offset 65 par rapport à l'adresse de base de Jean. Par conséquent, vous pourriez stocker la valeur de ax dans ce champ utilisant l'instruction mov

[18]Les chaînes de caractères (strings) requièrent un octet additionnel en plus de la longueur totale de la chaîne, pour encoder sa longueur.

[19]Vous ne pouvez pas redéfinir un nom de membre comme **equate** ou étiquette de macro. Vous pouvez cependant réutiliser un nom comme identificateur de déclaration. Notez aussi que les versions de MASM précédentes à la version 6.0 ne supportent pas la possibilité de réutiliser les noms de champ des structures.

Jean[65], ax. Malheureusement, mémoriser tous les offsets des champs d'une structure annule l'utilité première d'une structure. Après tout, si vous deviez manipuler tous ces offsets numériques, pourquoi ne pas utiliser un tableau de bytes à la place d'une structure ?

Eh bien, il avère assez naturellement que MASM vous fournit les mêmes facilités que C et Pascal pour accéder aux champs : l'opérateur point. Pour stocker ax dans le champ Licence, vous pouvez utiliser mov Jean. Licence, ax à la place de l'instruction précédente. Ceci est beaucoup plus lisible et certainement plus facile à utiliser.

Notez que l'usage de l'opérateur point ne présente pas un nouveau mode d'adressage. L'instruction *mov Jean.Majeur, ax* utilise toujours le mode d'adressage de déplacement seul. MASM additionne simplement l'adresse de base de Jean avec l'offset du champ Licence (65) pour obtenir la position correcte. Vous pouvez aussi spécifier des valeurs initiales par défaut. Dans l'exemple précédent les membres de la structure Jean avaient des valeurs initiales indéterminées, car vous aviez spécifié "?" dans le champ opérande de chaque déclaration de champ. Mais il y a deux moyens de spécifier une valeur initiale pour des champs de structure. Considérez la définition suivante d'une structure de données "Point" :

```
Point           struct
x               word    0
y               word    0
z               word    0
Point           ends
```

Quand vous déclarez une variable de type Point en utilisant une instruction comme :

```
        PointCourant    Point   {}
```

MASM initialise automatiquement les variables PointCourant.x, PointCourant.y et PointCourant.z à zéro. Ceci fonctionne à merveille quand vos objets ont tous initialement la même valeur[20]. Sans doute, vous pouvez vouloir préférer de donner des valeurs initiales différentes à la création de ces champs. Ceci peut se faire facilement en différenciant les valeurs à l'intérieur des accolades :

```
Point1          Point   {0, 1, 2}
Point2          Point   {1, 1, 1}
Point3          Point   {0, 1, 1}
```

MASM assigne des valeurs aux champs dans l'ordre où ces valeurs apparaissent en opérande. Pour Point1, MASM initialise le champ x à 0, y à 1 et z à 2.

Le type de la valeur initiale de l'opérande doit correspondre au type du champ correspondant dans la définition de la structure. Vous ne pouvez pas, par exemple, spécifier une constante entière pour un champ défini avec real4, pas plus qu'une valeur supérieure à 255 dans un champ de type byte.

MASM ne requiert pas l'initialisation de chaque valeur d'une structure. Si vous laissez un champ en blanc, MASM utilisera la valeur par défaut spécifiée (une valeur indéfinie si vous spécifiez "?" au lieu d'une valeur par défaut).

5.6.4 Tableaux de structures et tableaux (ou structures) comme champs de structures

Les structures peuvent contenir d'autres structures ou des tableaux comme membres. Considérez la définition suivante :

```
Pixel           struct
        Pt              point   {}
        Couleur         dword   ?
Pixel           ends
```

La définition qu'on vient de voir crée un point individuel avec un composant Couleur de 32 bits. En initialisant un objet de type Pixel, le premier initialiseur correspond au champ Pt et non au membre *x* de Pt. **La définition suivante est incorrecte :**

```
CePoint         Pixel   {5, 10}
```

[20]Notez cependant que la valeur initiale des champs x, y et z ne doit pas être nécessairement zéro. Vous auriez pu initialiser ces membres à 1, 2 et 3 de manière aussi facile.

La valeur du premier champ ("5") n'est pas un objet de type point. Par conséquent, l'assembleur émettra une erreur en rencontrant de cette initialisation. MASM permet d'initialiser les champs de CePoint via des déclarations comme :

```
CePoint          Pixel { ,10}
CePoint          Pixel {{}, 10}
CePoint          Pixel {{1,2,3},10}
CePoint          Pixel {{1,,1}, 12}
```

Le premier et le deuxième exemples utilisent les valeurs par défaut pour le membre Pt (x=0, y=0, z=0) de *point* et initialisent le membre Couleur à 10. Remarquez l'utilisation des accolades pour initialiser le champ Pt dans le second, troisième et quatrième exemple. Le troisième initialise les membres *x*, *y* et *z* de Pt à 1, 2 et 3 respectivement. Le dernier exemple initialise *x* et *z* et laisse *y* à sa valeur par défaut (dans ce cas, zéro).

Accéder aux membres de Pixel est très facile. Comme dans les langages de haut niveau, vous utilisez un premier point pour référencer le champ Pt et un second point pour accéder aux membres *x*, *y* et *z* de Pt :

```
                 mov     ax, CePoint.Pt.X
                  .
                  .
                  .
                 mov     CePoint.Pt.Y, 0
                  .
                  .
                  .
                 mov     CePoint.Pt.Z, di
                  .
                  .
                  .
                 mov     CePoint.Couleur, EAX
```

Vous pouvez aussi déclarer des tableaux comme des membres d'une structure. La structure suivante crée un type de donnée capable de représenter un objet avec huit points :

```
Objet8           struct
Pts              point   8 dup (?)
Couleur          dword   0
Objet8           ends
```

La structure alloue de l'espace pour huit différents points. Accéder à un élément du tableau Pts requiert la connaissance de la taille de l'objet de type point (souvenez-vous, vous multipliez l'index du tableau par la taille d'un élément, six dans ce cas particulier). Supposez, par exemple, que vous avez une variable CUBE de type Objet8. Vous pouvez accéder aux éléments de ce tableau comme suit :

```
; CUBE.Pts.X := 0;
                 mov     ax, 6
                 mul     i       ;6 octets par élément
                 mov     si, ax
                 mov CUBE.Pts[si].X, 0
```

L'aspect fâcheux de tout ceci est que vous devez connaître la taille de chaque élément du tableau Pts. Heureusement, MASM vous fournit un opérateur capable de calculer la taille (en octets) d'un élément de tableau pour vous ; on y reviendra.

5.6.5 Pointeurs sur des structures

Pendant l'exécution, votre programme peut se référer à des structures directement ou indirectement via un pointeur. Quand vous utilisez un pointeur pour accéder à des membres d'une structure, vous devez charger l'un des registres de pointeur (*si*, *di*, *bx* ou *bp* sur des processeurs inférieurs au 80386) avec l'offset de la structure désirée et *es*, *ds*, *ss* ou *cs*[21] avec l'adresse de segment de la même structure. Supposez que vous avez les déclarations de variables suivantes (en considérant la structure Objet8 de l'exemple précédent) :

```
Cube             Objet8  {}
CubePtr          dword   Cube
```

[21]Pour le 80386 et ultérieurs, il faut ajouter FS et GS à la liste.

CubePtr contient l'adresse de (c'est-à-dire est un pointeur sur) l'objet Cube. Pour accéder au champ Couleur de l'objet Cube, vous pouvez utiliser une instruction comme mov eax, Cube.Couleur. Mais quand vous accédez à un objet via un pointeur, vous devez charger l'adresse de cet objet dans une paire de registres segment:pointeur, comme es:bx. L'instruction les bx, CubePtr fait le travail. Après cela, vous pouvez accéder à des champs de l'objet Cube en utilisant le mode d'adressage disp+bx. Le seule question qui reste à poser est : « Comment spécifier le champ à accéder ? ». Considérez brièvement le code *incorrect* suivant :

```
les     bx, CubePtr
mov     eax, es:[bx].Couleur
```

Il y a un problème majeur ici. Étant donné que les noms de champs sont locaux à la structure et on peut réutiliser un nom de champ à l'intérieur de différentes structures, comment se fait-il que MASM détermine quel offset Couleur représenter ? Quand on accède à des membres de structure directement (par exemple, mov eax, Cube.Couleur), il n'y a pas d'ambiguïté, car Cube possède un type spécifique que l'assembleur peut vérifier. Par contre, es:bx peut pointer sur *n'importe quoi*. En particulier, il peut pointer sur n'importe quelle structure qui contient un champ Couleur. Par conséquent, l'assembleur ne peut pas, de sa part, décider quel offset utiliser pour le symbole Couleur.

Dans ces cas, MASM résout cette ambiguïté en exigeant que vous fournissiez un type. Probablement, le moyen le plus simple de le faire est de spécifier le nom de la structure comme *pseudo-champ* :

```
les     bx, CubePtr
mov     eax, es:[bx].Objet8.Couleur
```

En spécifiant le nom de la structure, MASM sait quelle valeur d'offset il faut utiliser pour le symbole Couleur[22].

5.7 Exemples de programmes

Les courts exemples de programmes suivants montrent divers concepts vus dans ce chapitre.

5.7.1 Déclarations de variables simples

```
;Pgm05_01.asm
;Exemple de déclaration de variables
;Ce fichier d'exemple démontre comment faire la déclaration et l'accès de certaines
;variables simples dans un programme en assembleur
;
; Randall Hyde
;
;
.xlist
include         stdlib.a
includelib      stdlib.lib
.list
.386
option segment:use16     ;Si vous ne vous servez pas d'un 80386 ou ultérieur


;Note : les déclarations des variables globales doivent se trouver dans le segment
"dseg"

dseg segment para public 'data'

;Déclaration de variables simples

character       byte    ?               ;"?" veut dire "non initialisé"
UnsignedIntVar  word    ?
DblUnsignedVar  dword   ?
```

[22]Les utilisateurs de MASM 5.1 et d'autres assembleurs devraient garder à l'esprit que les noms de champs *ne sont pas* locaux par rapport à la structure, mais ils doivent tous être uniques dans un fichier source. Comme résultat, de tels programmes ne requièrent pas le nom de la structure pour un champ particulier. Chose à tenir en considération quand vous convertissez du code plus ancien sous MASM 6.X.

```
;Utilisation de typedef pour créer des noms de types plus significatifs :
integer         typedef sword
char            typedef byte
FarPtr          typedef dword

;Déclarations de variables qui utilisent ces nouveaux types :
J               integer     ?
c1              char        ?
ptrVar          FarPtr      ?

;On peut indiquer à MASM et au DOS d'initialiser une variable quand le DOS charge le
;programme en mémoire en spécifiant la valeur initiale dans le champ de l'opérande
;de la déclaration de la variable :

K               integer     4
c2              char       'A'
PtrVar2         FarPtr      L   ;Initialisation de PtrVar2 avec l'adresse de L

;Avec ces directives, vous pouvez aussi réserver plus d'un octet, d'un mot ou d'un
;double-mot. Si on place diverses valeurs dans le champ de l'opérande, séparées par
;une virgule, l'assembleur émettra un octet, mot ou double-mot pour chaque
;opérande :

L               integer     0, 1, 2, 3
c3              char        'A', 0dh, 0ah, 0
PtrTbl          FarPtr      J, K, L

;La directive byte permet de spécifier une chaîne de caractères d'un octet,
;en mettant celle-ci entre guillemets ou apostrophes. La directive émet un octet
;pour chaque caractère de la chaîne (sans inclure les guillemets ou les apostrophes
;qui la contiennent) :

string          byte        "Hello Word", 0dh, 0ah, 0

dseg            ends


;Le programme suivant démontre comment accéder à chacune des variables ci-dessus.
cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg;Ces instructions sont fournies
                mov     ds, ax  ;par shell.asm pour initialiser
                mov     es, ax  ;le registre de segment.

;Des instructions simples qui démontrent comment accéder à la mémoire :
                lea     bx, L           ;bx pointe sur le premier mot de L
                mov     ax, [bx]        ;Charge le mot dans L
                add     ax, 2[bx]       ;Additionne le mot à L+2 (le "1")
                add     ax, 4[bx]       ;Additionne le mot à L+4 (le "2")
                add     ax, 6[bx]       ;Additionne le mot à L+6 (le "3")
                mul     K               ;Calcule (0+1+2+3) * 4
                mov     J, ax           ;Enregistre le résultat dans J
                les     bx, PtrVar2     ;Charge es:di avec l'adresse de L
                mov     di, K           ;Charge di avec 4
                mov     ax, es:[bx][di] ;Charge la valeur de L+4

;Exemples d'accès à des octets
                mov     c1, ' '         ;Place un espace dans la variable cl
                mov     al, c2          ;c3 := c2
                mov     c3, al

Quit:           mov     ah, 4ch         ;Nombre magique pour indiquer au DOS
                int 21h                 ;de dire à ce programme de quitter.
```

```
Main            endp
cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack     ")
sseg            ends

zzzzzzseg               segment para public 'zzzzzz'
LastBytes               byte    16 dup (?)
zzzzzzseg               ends

                end     Main
```

5.7.2 Utilisation des variables pointeur

```
;Pgm05_02.asm
;Utilisation de variables pointeur dans un programme en assembleur
;
;Ce petit exemple de programme démontre l'utilisation des pointeurs
;
;Randall Hyde

.xlist
include         stdlib.a
includelib      stdlib.lib
.list

.386
option          segment:use16   ;Si vous ne vous servez pas d'un 80386 ou
                                ;ultérieur

dseg            segment para public 'data'

;Quelques variables auxquelles on accédera indirectement (en utilisant des
;pointeurs) :

J               word    0, 0, 0, 0
K               word    1, 2, 3, 4
L               word    5, 6, 7, 8

;Les pointeurs near ont une taille de 16 bits et stockent un offset
;dans le segment de données courant (dseg dans ce programme). Les pointeurs
;far ont une taille de 32 bits et stockent une adresse complète de format
;segment:offset. Les définitions de types suivantes nous permettent de créer
;facilement des pointeurs near et far.

nWrdPtr         typedef         near ptr word
fWrdPtr         typedef         far ptr word

;Maintenant pour les véritables variables pointeur suivantes :

ptr1            nWrdPtr         ?
ptr2            nWrdPtr         K       ;Initialisé avec l'adresse de K
ptr3            fWrdPtr         L     ;Initialisé avec l'adresse segmentée de L

dseg            ends



cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
```

```
                mov     ax, dseg        ;Ces instructions sont fournies par
                mov     ds, ax          ;shell.asm pour initialiser les
                mov     es, ax          ;registres de segment

;Initialisation de ptr1 (un pointeur near) avec l'adresse de la variable J

                lea     ax, J
                mov     ptr1, ax

;Addition des quatre mots dans les variables J, K et L en utilisant des
;pointeurs sur ces mêmes variables

                mov     bx, ptr1        ;Obtenir le pointeur near du
                                        ;1er mot de J
                mov     si, ptr2        ;Obtenir le pointeur near du
                                        ;1er mot de K
                les     di, ptr3        ;Obtenir le pointeur far du
                                        ;1er mot de L

                mov     ax, ds:[si]     ;Obtenir des données à K+0
                                        ;(1er mot de K est 1)
                add     ax, es:[di]     ;Additionner avec les données à L+0
                mov     ds:[bx], ax     ;Conserver le résultat dans J+0

                add     bx, 2           ;Passer à J+2
                add     si, 2           ;Passer à K+2
                add     di, 2           ;Passer à L+2

                mov     ax, ds:[si]     ;Obtenir ce qu'il y a à k+2
                add     ax, es:[di]     ;Additionner avec les données à L+2
                mov     ds:[bx], ax     ;Conserver le résultat dans J+2

                add     bx, 2           ;Passer à J+4
                add     si, 2           ;Passer à K+4
                add     di, 2           ;Passer à L+4

                mov     ax, ds:[si]     ;Obtenir ce qu'il y a à k+4
                add     ax, es:[di]     ;Additionner avec les données à L+4
                mov     ds:[bx], ax     ;Conserver le résultat dans J+4

                add     bx, 2           ;Passer à j+6
                add     si, 2           ;Passer à K+6
                add     di, 2           ;Passer à L+6

                mov     ax, ds:[si]     ;Obtenir ce qu'il y a à k+6
                mov     ax, es:[di]     ;Additionner avec les données à L+6
                mov     ds:[bx], ax     ;Conserver le résultat dans J+6

Quit:           mov ah, 4ch             ;Nombre magique pour indiquer au DOS
                int 21h                 ;de dire à ce programme de quitter

Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack     ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

5.7.3 Accès à des tableaux unidimensionnels

```
;Pgm05_03.asm
;Exemple de déclaration de tableaux
;
;Ce programme montre comment déclarer et comment avoir accès à des tableaux à
;une seule dimension

;Randall Hyde

.xlist
include         stdlib.a
includelib      stdlib.lib
.list

.386
option          segment:use16   ;On a besoin d'utiliser des modes du 386

dseg            segment para public 'data'

J               word    ?
K               word    ?
L               word    ?
M               word    ?

JD              dword   0
KD              dword   1
LD              dword   2
MD              dword   3

;Quelques déclarations de tableaux non initialisés :
ByteAry         byte    4 dup(?)
WordAry         word    4 dup(?)
DwordAry        dword   4 dup(?)
RealAry         real8   4 dup(?)

;Quelques tableaux initialisés :
BArray          byte    0, 1, 2, 3
WArray          word    0, 1, 2, 3
DWArray         dword   0, 1, 2, 3
RArray          real8   0.0, 1.0, 2.0, 3.0

;Un tableau de pointeurs contenant l'adresse de ces pointeurs :
PtrArray        dword   ByteAry, WordAry, DwordAry, RealAry

dseg ends

;Le programme suivant montre comment accéder à chacune des variables déclarées
;ci-dessus :


cseg            segment para public 'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg        ;Ces instructions sont fournies
                mov     ds, ax          ;par shell.asm pour initialiser
                mov     es, ax          ;les registres de segment


;Initialisation des variables d'index. Notez que ces variables constituent
;des index logiques de tableaux. N'oubliez pas que, pour accéder à des éléments
```

```
;on doit multiplier ces valeurs par la taille d'un élément.

                mov     J, 0
                mov     K, 1
                mov     L, 2
                mov     M, 3

;Le code suivant montre comment accéder à des éléments de tableaux en
;utilisant des modes d'adressage simples du 80x86.

                mov     bx, J           ;AL := ByteAry[J]
                mov     al, ByteAry[bx]

                mov     bx, K           ;AX := WordAry[K]
                add     bx, bx          ;Index*2, car c'est un tableau de mots
                mov     ax, WordAry[bx]

                mov     bx, L           ;EAX := DwordAry[L]
                add     bx, bx          ;Index*4, car c'est un tableau de
                                        ;double-mots
                add     bx, bx
                mov     eax, DwordAry[bx]

                mov     bx, M           ;BX := adresse(RealAry[M])
                add     bx, bx      ;Index*8, car c'est un tableau de quad words
                add     bx, bx
                add     bx, bx
                lea     bx, RealAry[bx] ;Adresse base + index * 8


;Si vous avez un CPU 80386 ou supérieur, vous pouvez utiliser le mode
;d'adressage indexé scalaire pour simplifier l'accès aux tableaux.


                mov     ebx, JD
                mov     al, ByteAry[ebx]

                mov     ebx, KD
                mov     ax, WordAry[ebx*2]

                mov     ebx, LD
                mov     eax, DwordAry[ebx*4]

                mov     ebx, MD
                lea     bx, RealAry[ebx*8]

Quit:           mov     ah, 4ch         ;Nombre magique du DOS
                int     21h             ;pour indiquer au programme de quitter.
Main            endp

cseg            ends

sseg            segment  para stack 'stack'
stk             byte    1024 dup("stack   ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup(?)
zzzzzzseg       ends
                end     Main
```

5.7.4 Accès aux tableaux multidimensionnels

```
;Pgm05_04.asm
;Tableaux multidimensionnels : déclaration et accès
;
;Randall Hyde

.xlist
include         stdlib.a
includelib      stdlib.lib
.list

.386
option          segment:use16   ;on a besoin de ces deux déclarations pour
                                ;utiliser le jeu de registres du 80386


dseg            segment         para public 'data'

;Index qu'on va utiliser pour les tableaux
J               word    1
K               word    2
L               word    3

;Quelques tableaux à deux dimensions.
;Notez comment ce code utilise l'opérateur "dup" pour suggérer la taille de
;chaque dimension :

B2Ary           byte    3 dup(4 dup(?))
W2Ary           word    4 dup(3 dup(?))
D2Ary           dword   2 dup(6 dup(?))

;Tableaux bidimensionnels avec initialisation.
;Notez la disposition des données initiales pour suggérer la taille de
;chaque tableau :

B2Ary2          byte    0, 1, 2, 3
                byte    4, 5, 6, 7
                byte    8, 9, 10, 11

W2Ary2          word    0, 1, 2
                word    3, 4, 5
                word    6, 7, 8
                word    9, 10, 11

D2Ary2          dword   0, 1, 2, 3, 4, 5
                dword   6, 7, 8, 9, 10, 11

;Un exemple de tableau à trois dimensions :

W3Ary           word    2 dup(3 dup (4 dup(?)))

dseg            ends

cseg            segment para public 'code'
                assume cs:cseg, ds:dseg

Main            Proc
                mov     ax, dseg        ;Ces instructions sont fournies par
                mov     ds, ax          ;shell.asm pour initialiser
                mov     es, ax          ;les registres de segment.

;AL := B2Ary2[j,k]
                mov     bx, J           ;Index := (j*4+k)
                add     bx, bx          ;j*2
```

```
                add     bx, bx          ;j*4
                add     bx, K           ;j*4+k
                mov     al, B2Ary2[bx]

;AX := W2Ary2[j, k]
                mov     ax, J           ;Index := (j*3 + k)*2
                mov     bx, 3
                mul     bx              ;(j*3) -- Ceci détruit DX !
                add     ax, K           ;(j*3+k)
                add     ax, ax          ; (j*3+k)*2
                mov     bx, ax
                mov     ax, W2Ary2[bx]

;EAX := M2Ary[j, k]
                mov     ax, J           ;Index := (j*6 + k)*4
                mov     bx, 6
                mul     bx              ;DX:AX := J*6, en ignorant le
                                        ;dépassement de capacité dans DX
                add     ax, K           ;j*6 + k
                add     ax, ax          ;(j*6 + k)*2
                add     ax, ax          ;(j*6 + k)*4
                mov     bx, ax
                mov     eax, D2Ary[bx]

;Exemple d'accès à un tableau à trois dimensions
;
;AX := W3Ary[J,K,L]
                mov     ax, J           ;Index := ((j*3 + k)*4 + 1)*2
                mov     bx, 3
                mul     bx              ;*3
                add     ax, K           ;j*3 + k
                add     ax, ax          ;(j*3 + k)*2
                add     ax, ax          ;(j*3 + k)*4
                add     ax, 1           ;(j*3 + k)*4 + 1
                add     ax, ax          ;((j*3 + k)*4 + 1)*2
                mov     bx, ax
                mov     ax, W3Ary[bx]

Quit:           mov     ah, 4ch         ;Nombre magique pour le DOS
                int     21h             ;indiquant au programme de quitter

Main            endp

cseg            ends

sseg            segment         para stack 'stack'
stk             byte            1024 dup("stack    ")
sseg            ends

zzzzzzseg       segment         para public 'zzzzzz'
LastBytes       byte            16 dup(?)
zzzzzzseg       ends

                end             Main
```

5.7.5 Accéder à des structures simples

```
;Pgm05_05.asm
;Structures simples : déclaration et accès
;
;Randall Hyde
```

```
.xlist
include         stdlib.a
includelib      stdlib.lib
.list

.386
option          segment:use16   ;Si vous ne vous servez pas d'un 80386 ou
                                ;ultérieur

dseg            segment         para public 'data'

;La structure suivante contient les bits d'un octet mod-reg-r/m du 80x86

mode            struct
modbits         byte    ?
reg             byte    ?
rm              byte    ?
mode            ends

Instr1Adrs      mode    {}      ;Tous les champs non initialisés
Instr2Adrs      mode    {}

;Certaines structures avec des champs initialisés
axbx            mode    {11b, 000b, 000b}       ;Mode d'adressage "ax, ax"
axdisp          mode    {00b, 000b, 110b}       ;Mode d'adressage "ax, disp"
cxdispbxsi      mode    {01b, 001b, 000b}       ;Mode cx, disp8[bx][si]

;Pointeurs near sur des structures
sPtr1           word    axdisp
sPtr2           word    Instr2Adrs

dseg            ends


cseg            segment para public 'code'
                assume cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg        ;Ces instructions sont fournies par
                mov     ds, ax          ;shell.asm pour initialiser les
                mov     es, ax          ;registres de segment

;Pour accéder directement aux champs d'une variable de structure, utiliser
;simplement l'opérateur "." de Pascal ou de C
                mov     al, axbx.modbits
                mov     Instr1Adrs.modbits, al

                mov     al, axbx.reg
                mov     Instr1Adrs.reg, al

                mov     al, axbx.rm
                mov     Instr1Adrs.rm, al

;En accédant à des éléments d'une structure indirectement (en utilisant un
;pointeur), vous devez spécifier le type de structure comme premier champ, de
;sorte à ne pas embrouiller MASM.
                mov     si, sPtr1
                mov     di, sPtr2

                mov     al, ds:[si].mode.modbits
                mov     ds:[di].mode.modbits, al

                mov     al, ds:[si].mode.reg
```

```
                mov     ds:[di].mode.reg, al

                mov     al, ds:[si].mode.rm
                mov     ds:[di].mode.rm, al

Quit:           mov     ah, 4ch         ;Nombre magique du DOS
                int     21h             ;pour indiquer au programme de quitter
Main            endp

cseg            ends

sseg            segment         para stack 'stack'
stk             byte            1024 dup("stack    ")
sseg            ends

zzzzzzseg       segment         para public 'zzzzzz'
LastBytes       byte            16 dup(?)
zzzzzzseg       ends

                end             Main
```

5.7.6 Tableaux de structures

```
;Pgm05_06.asm
;Tableaux de structures
;
;Randall Hyde

.xlist
include         stdlib.a
includelib      stdlib.lib
.list

.386
option          segment:use16   ;Si vous ne vous servez pas d'un 80386 ou
                                ;ultérieur

dseg            segment         para public 'data'

;Voici une structure définissant une coordonnée (x, y). Notez que le type Point
;requiert quatre octets.

Point   struct
X               word    ?
Y               word    ?
Point   ends

;Un point non initialisé
Pt1             Point   {}

;Un point initialisé
Pt2             Point   {12, 45}

;Un tableau unidimensionnel de points non initialisés
PtAry1          Point   16 dup({}) ; Remarquez les "{}" entre parenthèses.

;Un tableau unidimensionnel de points tous initialisés à l'origine
PtAry1i         Point   16 dup({0, 0})

;Un tableau de points à deux dimensions
PtAry2          Point   4 dup(4 dup({}))
```

```
;Un tableau de points à trois dimensions, tous initialisés à l'origine
PtAry3          Point   2 dup(3 dup(4 dup({0, 0})))

;Un tableau de points unidimensionnel, initialisés à des valeurs différentes
iPtAry          Point   {0, 0}, {1, 2}, {3, 4}, {5, 6}

;Quelques index pour les tableaux
J               word    1
K               word    2
L               word    3

dseg            ends

;Le programme suivant démontre comment accéder aux variables ci-dessus

cseg            segment para public 'code'
                assume cs:cseg, ds:dseg

                Main    proc
                mov     ax, dseg        ;Ces instructions sont fournies par
                mov     ds, ax          ;shell.asm, afin d'initialiser les
                mov     es, ax          ;registres de segment.

;PtAry1[J] := iPtAry[J]
                mov     bx, J           ;Index := j*4 pusqu'il y a 4 octets par
                                        ;élément
                add     bx, bx
                add     bx, bx

                mov      ax, iPtAry[bx].X
                mov      PtAry1[bx].X, ax

                mov      ax, iPtAry[bx].Y
                mov      PtAry1[bx].Y, ax

;CX := PtAry2[K, L].X; DX:=PtAry2[K, L].Y
                mov      bx, J          ;Index := (K*4 + J)*4
                add      bx, bx         ;k * 2
                add      bx, bx         ;k * 4
                add      bx, J          ;k*4 + j
                add      bx, bx         ;(K * 4 + j) * 2
                add      bx, bx         ;(K * 4 + J) * 4

                mov     cx, PtAry2[bx].X
                mov     dx, PtAry2[bx].Y

;PtAry3[j,k,l].X := 0
                mov     ax, J           ;Index := ((j*3 + k) * 4 + 1) * 4
                mov      bx, 3
                mul     bx              ;j*3
                add     ax, k           ;j*3 + k
                add     ax, ax          ;(j*3 + k)*2
                add     ax, ax          ;(j*3 + k)*4
                add     ax, 1           ;(j*3 + k)*4 + 1
                add     ax, ax          ;((j*3 + k)*4 + 1)*2
                add     ax, ax          ; ((j*3 + k)*4 + 1)*4
                mov     bx, ax
                mov     PtAry3[bx].X, 0

Quit:           mov     ah, 4ch
                int     21h
Main            endp
```

```
cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024    dup("stack    ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzzz'
LastBytes       byte 16 dup(?)
zzzzzzseg       ends
                end Main
```

5.7.7 Structures et tableaux comme champs d'autres structures

```
;Pgm05_07.asm
;Structures contenant d'autres structures comme champs
;Structures contenant des tableaux comme champs
;
;Randall Hyde

.xlist
include         stdlib.a
includelib      stdlib.lib
.list

.386
option          segment:use16    ;Si vous ne vous servez pas d'un 80386 ou
                                 ;ultérieur

dseg            segment          para public 'data'

Point           struct
X               word    ?
Y               word    ?
Point           ends

;On peut définir un rectangle avec seulement deux points. Le champ de la
;Couleur contient seulement une valeur de couleur de 8 bits. Note : la
;taille d'un Rect est de 9 octets.

Rect            struct
HautGauche      Point   {}
BasDroit        Point   {}
Couleur                 byte    ?
Rect            ends

;Les pentagones ont cinq points, donc, on peut définir un tableau de points
;pour définir un pentagone. Sans doute, on a besoin aussi d'un champ Couleur.
;Note : la taille d'un pentagone est de 21 octets.

Pent            struct
Couleur                 byte    ?
Pts             Point   5 dup({})
Pent            ends

;Ok, voici quelques déclarations de variables :

Rect1           Rect    {}
Rect2           Rect    {{0, 0}, {1, 1}, 1}

Pentagone1      Pent    {}
Pentagones      Pent    {}, {}, {}, {}

Index           word    2
```

```
dseg            ends

cseg            segment para public 'code'
                assume cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg        ;Ces   instructions   sont   fournies   par
shell.asm
                mov     ds, ax          ;pour initialiser les registres de segment
                mov     es, ax

;Rect1.HautGauche.X := Rect2.HautGauche.X

                mov     ax, Rect2.HautGauche.X
                mov     Rect1.HautGauche.X, ax

;Pentagone1 := Pentagones[Index]
                mov     ax, Index       ;Il faut Index * 21
                mov     bx, 21
                mul     bx
                mov     bx, ax

;Copier le premier point :
                mov     ax, Pentagones[bx].Pts[0].X
                mov     Pentagone1.Pts[0].X, ax

                mov     ax, Pentagones[bx].Pts[0].Y
                mov     Pentagone1.Pts[0].Y, ax

;Copier le second point :
                mov     ax, Pentagones[bx].Pts[4].X[23]
                mov     Pentagone1.Pts[4].Y, ax

                mov     ax, Pentagones[bx].Pts[4].Y
                mov     Pentagone1.Pts[4].Y, ax

;Copier le troisième point :
                mov     ax, Pentagones[bx].Pts[8].X
                mov     Pentagone1.Pts[8].X, ax

                mov     ax, Pentagones[bx].Pts[8].Y
                mov     Pentagone1.Pts[8].Y, ax

;Copier le quatrième point :
                mov     ax, Pentagones[bx].Pts[12].X
                mov     Pentagone1.Pts[12].X, ax

                mov     ax, Pentagones[bx].Pts[12].Y
                mov     Pentagone1.Pts[12].Y, ax

;Copier le cinquième point :
                mov     ax, Pentagones[bx].Pts[16].X
                mov     Pentagone1.Pts[16].X, ax

                mov     ax, Pentagones[bx].Pts[16].Y
                mov     Pentagone1.Pts[16].Y, ax

;Copier la couleur :
                mov     al, Pentagones[bx].Couleur
                mov     Pentagone1.Couleur, al

Quit:           mov     ah, 4ch         ;Nombre magique du DOS
                int     21h             ;pour indiquer au programme de quitter
```

[23]Dans la version originale, à partir de ce point, les index de Pts de Pentagones sont erronés. L'index de Pts était multiplié par deux, alors qu'il fallait le multiplier par quatre, car quatre est la taille de chaque Pts. L'original, au lieu d'avoir 4, 8, 12, 16 comme index, avait 2, 4, 6, 8, ce qui évidemment ne marche pas, n.d.t.

```
Main            endp
cseg            ends

sseg            segment  para stack 'stack'
stk             byte     1024 dup("stack     ")
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       byte     16 dup(?)
zzzzzzseg       ends
                end      Main
```

5.7.8 Pointeurs sur des structures et sur des tableaux de structures

```
;Pgm05_08.asm
;Pointeurs sur des structures
;Pointeurs sur des tableaux de structures
;
;Randall Hyde

.xlist
include         stdlib.a
includelib      stdlib.lib
.list

.386
option          segment:use16    ;On a besoin de ces deux déclarations
                                 ;pour utiliser les registres du 80386

dseg            segment          para public 'data'

;Structure Sample
;Notez : sa taille est de 7 octets.

Sample          struct
b               byte     ?
w               word     ?
d               dword    ?
Sample          ends

;Quelques déclarations de variables :

OneSample       Sample           {}
SampleAry       Sample           16 dup ({})      ;Tableau de 16 structures

;Pointeurs sur les variables ci-dessus

OnePtr          word     OneSample                        ;Un pointeur proche
AryPtr          dword    SampleAry

;Index du tableau

Index           word     8

dseg            ends

;Le programme suivant montre comment accéder à chacune des variables ci-dessus.

cseg            segment  para public 'code'
                assume cs:cseg, ds:dseg

Main            proc
                mov      ax, dseg;Ces lignes sont fournies par shell.asm
                mov      ds, ax  ;pour initialiser les registres de segment.
                mov      es, ax

;AryPtr[Index] := OnePtr
```

```
                mov     si, OnePtr      ;Obtenir un pointeur sur OneSample
                les     bx, AryPtr      ;Obtenir un pointeur sur un tableau de
        Samples
                mov     ax, Index               ;Il faut Index * 7
                mov     di, 7
                mul     di
                mov     di, ax

                mov     al, ds:[si].Sample.b
                mov     es:[bx][di].Sample.b, al
                mov     ax, ds:[si].Sample.w
                mov     es:[bx][di].Sample.w, ax

                mov     eax, ds:[si].Sample.d
                mov     es:[bx][di].Sample.d, eax

Quit :          mov     ah, 4ch         ;Nombre magique du DOS
                int     21h             ;pour indiquer au programme de quitter.
Main            endp
cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024    dup("stack    ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzzz'
LastBytes       byte 16 dup(?)
zzzzzzseg       ends
                end Main
```

5.8 Exercices de laboratoire

Dans ces exercices de laboratoire vous allez apprendre comment faire la trace à l'intérieur d'un programme utilisant CodeView et observer les résultats. Il s'agit d'une technique importante à posséder. Il n'y a pas de meilleur moyen d'apprendre l'assembleur qu'en traçant et en observant les actions accomplies par chaque instruction. Même si vous connaissez déjà l'assembleur, tracer dans un programme avec un débogueur comme CodeView est le meilleur moyen pour vérifier si votre programme fonctionne correctement.

Dans ces exercices, vous allez assembler les programmes d'exemple fournis dans la section précédente ; puis, vous exécuterez les programmes assemblés sous CodeView et vous observerez chaque instruction du programme. **Pour votre rapport de laboratoire :** vous aurez à inclure un listing de chaque programme et à décrire l'opération de chaque instruction, en incluant les données chargées dans tout registre affecté ou bien stockées dans la mémoire.

Le paragraphe suivant décrit une exécution expérimentale : tracer dans le programme pgm5_1.asm. Votre rapport de laboratoire doit contenir des informations similaires pour les huit programmes d'exemple.

Pour assembler vos programmes, utilisez la commande ML avec l'option /Zi. Par exemple, pour assembler le premier programme, utilisez la commande DOS suivante :

```
ml /Zi pgm5_1.asm
```

Ce qui produit le fichier pgm5_1.exe qui contient les informations de débogage de CodeView. Vous pouvez charger ce programme dans le débogueur de CodeView, en utilisant la commande suivante :

```
cv pgm5_1
```

Une fois que vous vous trouvez à l'intérieur de CodeView, vous pouvez faire la trace en pressant la touche F8 de façon répétitive : chaque fois que vous pressez cette touche, CodeView exécute une instruction.

Afin de mieux observer les résultats pendant la trace, il faudrait ouvrir la fenêtre des registres. Si elle n'est pas encore ouverte, vous pouvez l'ouvrir en pressant la touche F2. A mesure que les instructions que vous exécutez modifieront les registres, vous pouvez observer les changements.

Tous les programmes d'exemple commencent par une séquence de trois instructions qui initialisent les registres DS et ES ; presser trois fois la touche F8 permet de faire la trace dans ces instructions et charge les registres AX, DS et ES avec la valeur 1927h (valeur qui peut changer d'un système à un autre).

En continuant la trace, l'instruction lea bx, L charge la valeur 0015h dans bx. Et la trace sur le groupe d'instructions qui suivent *lea* produit les résultats suivants :

```
mov     ax, [bx]                ;AX = 0
add     ax, 2[bx]                       ;AX = 1
add     ax, 4[bx]                       ;AX = 3
add     ax, 6[bx]                       ;AX = 6
mul     K                       ;AX = 18
mov     J, ax                   ;J est maintenant égal à 18h
```

Commentaires sur les instructions ci-dessus : ce code charge bx avec l'adresse de base du tableau L et ensuite calcule la somme de L[i], i = 0..3 (0+1+2+3). Puis, il effectue la multiplication entre la somme et K(4) et garde le résultat dans J. Notez que vous pouvez utiliser la commande "dw J" dans la fenêtre des commandes pour afficher la valeur courante de J[24] (où "J" doit être en majuscule parce que CodeView est sensible à la casse).

```
les     bx, PtrVar2             ;BX = 0015, ES = 1927
mov     di, K                   ;DI = 4
mov     ax, es:[bx][di]         ;AX = 2
```

Commentaires : l'instruction *les* charge es:bx avec la variable pointeur PtrVar2. Cette variable contient l'adresse de la variable L. Puis, ce code charge di avec la valeur de K et termine en chargeant le second élément de L dans ax.

```
mov     c1, ' '
mov     a1, c2
mov     c3, al
```

Ces trois instructions stockent simplement un espace dans la variable de type byte c1 (vérifier avec la commande "da c1" dans la fenêtre de commandes) et copient la valeur de c2 ("A") dans le registre AL et la variable c3 (vérifier avec "da c3").

Pour votre rapport de laboratoire : assemblez et faites la trace des programmes pgm5_2.asm à pgn5_8.asm. Décrivez les résultats de manière similaire à comme on l'a fait ici.

5.9 Projets de programmation

1) L'affichage vidéo du PC est un composant d'E/S mappées en mémoire. C'est-à-dire, l'adapteur d'affichage fait correspondre chaque caractère de l'affichage à un mot en mémoire. L'affichage est un tableau 80x85 de mots et il est déclaré comme suit :

```
affichage:array[0..24,0..79] of word;
```

affichage[0,0] correspond au coin supérieur gauche de l'écran, affichage[0,79] est le coin supérieur droit de l'écran, alors qu'affichage[24,0] constitue le coin inférieur gauche et affichage[24,79] le coin inférieur droit.

L'octet le moins significatif de chaque mot garde le code ASCII du caractère qui doit apparaître à l'écran, alors que l'octet le plus significatif contient l'attribut byte (voir le début du chapitre 23 pour plus de détails sur l'octet attribut). L'adresse de base est B000:0 pour les affichages monochromes et B800:0 pour les moniteurs couleur.

Le dossier du chapitre 5 contient un fichier nommé PROJ5_1.ASM. Ce fichier est un programme squellette, manipulant l'affichage vidéo. Ce programme, une fois complété, écrit une série de points à l'écran et puis il écrit une suite d'espaces bleus. Il contient un programme principal qui utilise une série d'instructions que vous n'avez probablement encore vues. Ces instructions exécutent une boucle for comme suit :

```
for i := 0 to 79 do
```

[24]Dans certaines versions de CodeView, ceci ne fonctionne pas de la façon qu'on vient de voir ici, n.d.t.

```
for j := 0 to 24 do
          putscreen(i, j, valeur);
```

A l'intérieur du programme, vous trouverez quelques commentaires vous invitant à écrire le code pour stocker une valeur donnée de AX dans l'emplacement d'affichage [i,j]. Modifiez le programme comme décrit dans les commentaires et testez le résultat.

Pour ce projet, vous avez besoin de déclarer deux variables de type word, I et J dans le segment des données. Puis, vous aurez à modifier la procédure *PutScreen*. Dans cette procédure, tel qu'indiqué par les commentaires de ce fichier, vous aurez à calculer l'index servant à référencer le tableau de l'écran et à stocker la valeur se trouvant dans le registre ax dans es:[bx+0]. Notez que es:[0] est l'adresse de base de l'affichage vidéo dans cette procédure. Vérifiez votre code soigneusement avant de l'exécuter ; si celui-ci fonctionne mal, il peut planter le système et vous aurez à redémarrer. Ce programme, à condition qu'il fonctionne bien, remplira l'écran avec des points jusqu'à ce que vous ne pressez une touche ; après cela, il remplira l'écran par des espaces bleus. Après l'exécution du programme, vous aurez probablement à exécuter la commande DOS, *CLS* (*CLear Screen*) de nettoyage de l'écran. Notez que dans le répertoire CH05, il y a une version qui fonctionne de ce programme, nommée p5_1.exe. Vous pouvez l'exécuter pour observer son fonctionnement, en cas de problèmes.

2) Le répertoire du chapitre 5 contient aussi un autre fichier nommé PROJ5_2.ASM. Ce fichier est un programme (sauf pour deux courtes sous-routines), qui génère des labyrinthes et les résout à l'écran. Ce programme demande de compléter les deux sous-routines MazeAdrs et ScrnAdrs. Ces deux procédures apparaissent au début du fichier ; vous devriez ignorer le reste du code. Quand le programme appelle la fonction MazeAdrs, il passe une coordonnée X au registre dx et une coordonnée Y au registre cx. Vous devrez calculer l'index d'un tableau 27x82, défini comme suit :

```
maze:array[0..26, 0..81] of word;
```

Retournez l'index dans le registre ax. *N'accédez pas au tableau maze (*labyrinthe*) ; le code appelant le fera pour vous*.

La fonction ScrnAdrs est presque identique à la fonction MazeAdrs, sauf qu'elle calcule un index pour un tableau 25x80 au lieu d'un tableau 27x82. Comme dans MazeAdrs, la coordonnée X sera dans le registre dx et la coordonnée Y dans le registre cx.

Complétez ces deux fonctions, assemblez le programme et exécutez-le. Assurez-vous très soigneusement que votre travail est correct, car même la plus petite erreur fera sans doute planter le système.

3) Créez un programme avec un tableau de structures unidimensionnel. Placez-y au moins quatre champs (de votre choix). Ecrivez un segment de code pour avoir accès à l'élément "i" (où *i* serait une variable word) dans le tableau.

4) Ecrivez un programme qui copie les données d'un tableau 3x3 et les garde dans un second tableau 3x3. Pour le premier tableau, stockez les données selon un ordre orienté rangée. Pour le second, stockez les données selon un mode orienté colonne. Utilisez neuf séquences d'instructions chargeant le mot à l'emplacement (i,j) (i=0..2,j=0..2).

5) Récrivez la séquence de code ci-dessus en n'utilisant que des instructions MOV. Lisez et écrivez les emplacements des tableaux directement, n'effectuez pas les calculs d'adresses des tableaux.

5.10 Résumé

Ce chapitre présente une vue de l'organisation de la mémoire et des structures de données centrée sur le 80x86. Ce n'est certainement pas un cours complet sur les structures de données. Il traite simplement des types de données simples et composés et il explique comment les déclarer et les utiliser dans vos programmes. Vous trouverez beaucoup d'autres informations sur la déclaration et l'usage des types simples dans les chapitres qui suivent.

L'un des buts principaux de ce chapitre a été de décrire comment déclarer et utiliser des *variables* dans un programme en assembleur. Dans un tel programme vous pouvez facilement réserver des octets, des mots, des

doubles-mots et d'autres types de variables. Ces types de données scalaires peuvent comprendre des types booléens, des entiers, des réels et d'autres types de données que vous trouvez couramment dans des langages de haut niveau. Voir :

• "Déclaration des variables dans un programme", au paragraphe 5.2
• "Déclaration et utilisation des variables BYTE", au paragraphe 5.3.1
• "Déclaration et utilisation des variables WORD", au paragraphe 5.3.2
• "Déclaration et utilisation des bariables DWORD", au paragraphe 5.3.3
• "Déclaration et utilisation de bariables FWORD, QWORD et TBYTE", au paragraphe 5.3.4
• "Déclaration des variables en virgule flottante avec REAL4, REAL8 et REAL 10", au paragraphe 5.3.5

MASM permet à ceux qui n'aiment pas utiliser des variables comme byte, word, etc., de créer des noms de types personnalisés. Voulez-vous les appeler *integer* au lieu que *word* ? Pas de problème, vous pouvez définir vos propres noms de types avec l'instruction typedef. Voir :

• "Création de types personnalisés avec TYPEDEF", au paragraphe 5.4

Un autre type de données important est le *pointeur*. Les pointeurs ne sont rien plus que des adresses de mémoire stockés dans des variables (généralement de type word ou dword). Les CPU 80x86 comprennent deux types de pointeur : *proches* (near) et *éloignés* (far). En mode réel, les pointeurs near ont une taille de 16 bits et contiennent l'offset d'un segment connu (typiquement, le segment de données). Les pointeurs far sont de 32 bits et contiennent une adresse logique complète de type segment:offset. Souvenez-vous que vous devez utiliser un des modes d'adressages indirects ou indexés pour accéder à une donnée référencée par un pointeur. Pour ceux qui veulent créer leurs propres types de pointeur (au lieu d'utiliser simplement word ou dword pour déclarer des pointeurs near ou far), l'instruction typedef permet encore une fois de personnaliser les déclarations. Voir :

• "Les données pointeur", au paragraphe 5.5

Un type de données composite est un type qui est formé à partir de types plus simples. Il y a abondance d'exemples de ces types, mais deux des types de données composites les plus populaires sont les tableaux et les structures (enregistrements). Un tableau est un groupe de variables, toutes du même type. Un programme sélectionne un élément d'un tableau à l'aide d'un index entier qui en contient l'offset. Les structures, d'autre part, peuvent contenir des membres dont les types sont différents. Dans un programme, vous sélectionnez le champ désiré en fournissant un nom de champ avec l'*opérateur point*. Voir :

• "Tableaux", au paragraphe 5.6.1
• "Tableaux multidimensionnels", au paragraphe 5.6.2
• "Structures", au paragraphe 5.6.3
• "Tableaux de structures et tableaux ou structures comme champs de structures ", au paragraphe 5.6.4
• "Pointeurs sur des structures ", au paragraphe 5.6.5

5.11 Questions

1. Dans quel segment (8086) placeriez- vous normalement vos variables ?
2. Quel segment dans le fichier SHELL.ASM correspond au segment contenant les variables ?
3. Décrivez comment déclarer des variables d'un octet. Donnez plusieurs exemples. Quelles sont les utilisations courantes de ces variables dans un programme ?
4. Décrivez comment déclarer des variables d'un mot. Donnez plusieurs exemples. Quelles sont les utilisations courantes de ces variables dans un programme ?
5. Répétez la question 4 pour des variables d'un double-mot.
6. Expliquez la finalité de l'instruction TYPEDEF. Donnez des exemples de son utilisation.
7. Qu'est-ce que c'est une variable pointeur ?
8. Quelle est la différence entre un pointeur *near* et *far* ?

9. De quelle manière accédez-vous à un objet pointé par un pointeur far ? Donnez un exemple en utilisant les instructions 8086.

10. Qu'est-ce que c'est un type de données composite ?

11. Comment déclare-t-on des tableaux en assembleur ? Fournissez le code pour les tableaux suivants :

    a) Un tableau d'octets bidimensionnel 4x4 b) Un tableau contenant 128 double-mots

    c) Un tableau contenant 16 mots d) Un tableau de mots tridimensionnel 4x5x6

12. Décrivez comment accéder à un élément des tableaux ci-dessus. Donnez les formules nécessaires et le code 8086 pour un tel accès (en présumant que la variable I est l'index dans un tableau unidimensionnel, que I et J sont les index d'un tableau bidimensionnel et I, J et K les index pour un tableau tridimensionnel). Utilisez le mode orienté rangée si approprié.

13. Donnez le code 80386, via le mode d'adressage scalaire, pour accéder aux éléments des tableaux ci-dessus.

14. Expliquez la différence entre le mode orienté rangée et le mode orienté colonne.

15. Supposez avoir un tableau à deux dimensions que vous voulez initialiser comme suit :

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

    Donnez la déclaration appropriée de la variable tableau pour obtenir ceci. Note : n'utilisez pas les instructions machine du 8086 pour l'initialisation. Initialisez-le dans votre segment de données.

16. Ecrivez les déclarations équivalentes en assembleur pour déclarer les structures suivantes :

```
Date=           Record
                        Mois:integer;
                        Jour:integer;
                        Annee:integer;
                end;

Heure=          Record
                        Heures:integer;
                        Minutes:integer;
                        Secondes:integer;
                end;

Videocassette=  Record
                        Titre:string[25];
                        DateParution:Date;
                        Prix:Real; (*Supposez des réels de 4 bytes*)
                        Longueur:Heure;
                        Cote:char;
                end;

Videotheque : array [0..127] of Videocassette; (*Ceci est une variable*)
```

17. Supposez que ES:BX pointe sur un objet de type Videocassette. Quelle est l'instruction permettant de charger convenablement le champ Cote dans AL ?