Chapitre 9
Sous-programmes

Une suite d'instructions qui doit être exécutée à différents endroits d'un programme, peut ne pas être incluse entièrement à chacun de ces endroits: on peut en faire un sous-programme. A chaque endroit du programme où l'on désire exécuter la suite d'instructions, on place un appel au sous-programme comprenant l'information à traiter. L'exécution du sous-programme terminée, le contrôle est rendu au programme appelant à l'endroit suivant immédiatement l'appel. Tout se passe en fait comme si les instructions du sous-programme avaient été insérées à la place de l'instruction d'appel (figure 9.1).

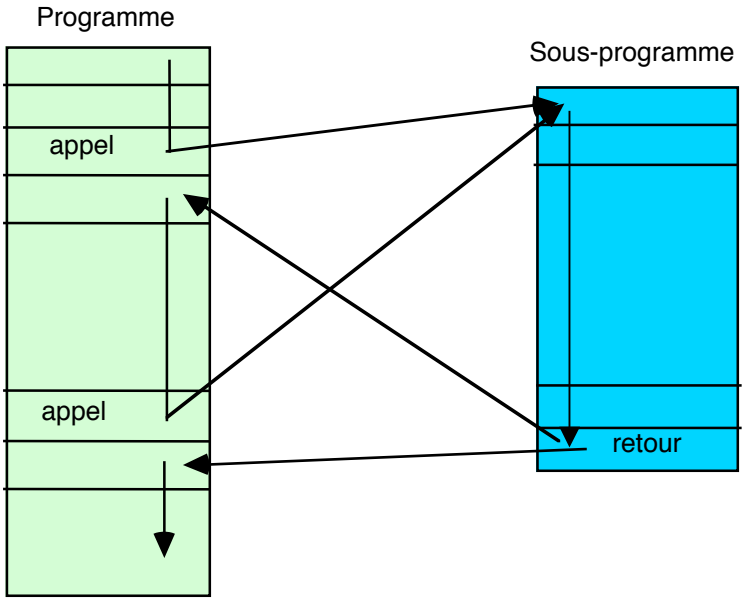


Figure 9.1 Ordre d'exécution des instructions

Les sous-programmes offrent un certain nombre d'avantages: on peut utiliser des solutions antérieurement programmées comme sous-tâches du problème à résoudre; un problème peut être décomposé en tâches, chaque tâche étant résolue par un sous-programme; la mise au point est facilitée, car on peut mettre au point chaque sous-programme séparément. Par contre, il faut résoudre le problème du transfert d'information du programme appelant au sous-programme et de la sauvegarde de l'adresse de retour.

Dans les anciens ordinateurs, l'adresse de retour était conservée dans le sous-programme: ceci interdisait la récursivité directe ou indirecte[1]. En effet, un second appel à un sous-programme dont le premier appel n'était pas terminé, détruisait la première adresse de retour en la remplaçant par la seconde. Il n'était alors plus possible de revenir à l'instruction suivant le premier appel.

[1] *Récursivité directe:* sous-programme comportant un appel à lui même dans sa définition.
Récursivité indirecte: sous-programme Sous-Prog1 appelant un sous-programme Sous-Prog2 lequel appelle Sous-Prog1.

Pour éviter ce problème, les ordinateurs modernes utilisent une *pile* où ils empilent les adresses de retour. Ceci permet à un même sous-programme d'empiler plusieurs adresses de retour, rendant ainsi possibles les appels récursifs. L'ordinateur PEP 8 possède une pile d'exécution à laquelle pointe un registre d'adresse spécial, aussi repéré par "SP" (*stack pointer*). Lorsque l'exécution d'un programme démarre sous le contrôle du système d'exploitation, une zone de mémoire est affectée à la pile et le pointeur de pile est initialisé. La pile croît vers les adresses les plus basses: ainsi **on diminue SP pour empiler**, et **on augmente SP pour désempiler**. Si le système d'exploitation ne préparait pas une pile ce serait au programmeur de le faire. **Le pointeur SP** repère le sommet de la pile et **change de valeur** très souvent au cours de l'exécution d'un programme.

9.1 Instructions CALL et RETn

L'appel d'un sous-programme n'est qu'une rupture de l'exécution séquentielle des instructions du programme: une instruction de branchement effectuera le transfert de contrôle, cependant il faudra lui ajouter un moyen de conserver l'adresse de retour. Il existe une instruction de branchement inconditionnel qui conserve l'adresse de retour, `CALL` ("appel"); elle a la forme suivante:

```
CALL    adresse     ;SP -= 2; (SP) = CO; CO = adresse effective
```

La valeur du compteur ordinal (qui indique l'adresse de l'instruction suivant le `CALL`) est placée sur la pile (empilée, **SP est diminué de 2**). On effectue alors un branchement à l'étiquette ou à l'adresse spécifiée comme opérande. Cette instruction n'affecte pas les codes de condition. Comme l'adresse de retour est conservée sur la pile, il conviendra dans le sous-programme de ne pas modifier le contenu de la pile ou de le restaurer avant d'exécuter l'instruction de retour. Le retour pourra être fait simplement au moyen d'une instruction `RETn` ("retour"):

```
RETn                ; SP += n ; CO = (SP); SP += 2
```

qui élimine n octets de la pile (la valeur de `n` va de zéro à sept) en **augmentant SP de n**, puis effectue un branchement inconditionnel à l'adresse située au sommet de la pile. Cette adresse de retour est ensuite également désempilée, **SP est augmenté une seconde fois de 2**.

Voyons maintenant un exemple d'appel de sous-programmes:

```
CALL    Calcul
```

On place au sommet de la pile l'adresse de retour du sous-programme (instruction suivant le `CALL`), on effectue alors le branchement au sous-programme Calcul. Dans le sous-programme Calcul, le retour au programme appelant se fera par l'instruction:

```
RET0
```

Prenons comme exemple un petit sous-programme simple qui va nous aider à afficher la figure ci-dessous, laquelle représente grossièrement un sapin de Noël. Le sous-programme affichera un petit triangle composé d'astérisques. Le programme principal appellera le sous-programme trois fois consécutivement, de façon à afficher le sapin désiré. Le premier appel empile l'adresse de retour 0003 et continue l'exécution à l'adresse 000A; l'exécution du premier `RET0`, à la fin du premier appel, utilise cette adresse et la désempile. Le second appel empile l'adresse 0006; le second `RET0`, exécuté à la fin

du second appel, désempile cette adresse. Le troisième appel empile l'adresse 0009; l'exécution du troisième `RET0` la désempile et retourne au programme principal exécuter l'instruction `STOP` qui termine l'exécution.

```
                                *
                               ***
                              *****
                                *
                               ***
                              *****
                                *
                               ***
                              *****
```

```
Addr  Code   Symbol   Mnemon  Operand    Comment
             ; Affichage d'un sapin de Noël par affichage de triangles
             ;  Ph. Gabrini (tiré de "An introduction to computer science with
             ;  Modula2" par Adams, Gabrini, Kurtz, D.C. Heath 1988)
                                         ;int main (){
0000  16000A Sapin:   CALL    AffTrian   ;  AffTrian ()
0003  16000A          CALL    AffTrian   ;  AffTrian ()
0006  16000A          CALL    AffTrian   ;  AffTrian ()
0009  00              STOP               ;}
                                         ;void AffTrian(){
000A  41001D AffTrian:STRO    une,d      ;  cout << "  *"
000D  50000A          CHARO   '\n',i     ;       << endl
0010  410021          STRO    deux,d     ;       << " ***"
0013  50000A          CHARO   '\n',i     ;       << endl
0016  410026          STRO    trois,d    ;       << "*****"
0019  50000A          CHARO   '\n',i     ;       << endl;
001C  58              RET0               ;}
001D  20202A une:     .ASCII  "  *\x00"
      00
0021  202A2A deux:    .ASCII  " ***\x00"
      2A00
0026  2A2A2A trois:   .ASCII  "*****\x00"
      2A2A00
                      .END
```

9.2 Paramètres

Le second aspect de l'appel des sous-programmes est la transmission d'information du programme appelant au sous-programme et du sous-programme au programme appelant. En effet, le sous-programme doit travailler sur des données du programme et on s'attend à ce qu'il produise au moins un résultat qui sera transmis au programme. Il y a de nombreuses façons d'effectuer la transmission d'information entre programme appelant et sous-programme, nous ne verrons ici que les principales méthodes.

9.2.1 Paramètres valeurs

Les paramètres transmis par valeur (ou en mode IN en Ada 95) sont recopiés dans des variables locales du sous-programme; si ces variables sont modifiées au cours de l'exécution du sous-programme, cela n'affecte en rien la valeur originale, c'est le cas des paramètres valeurs A et B de la procédure C++ :

```
void Valeurs(int A, int B);
```

Notez qu'en Java, tous les paramètres sont passés par valeur ; si un objet est passé en paramètre, on en passe la référence, c'est-à-dire une copie de l'adresse. Cependant l'objet indiqué par cette adresse est malheureusement disponible pour modification ; comme cette pratique est contraire aux principes de génie logiciel et est à proscrire, on l'appelle de façon euphémique un *effet de bord* de la fonction appelée, puisque cette fonction, en plus du résultat qu'elle retourne, modifie sournoisement un de ses paramètres.

En langage d'assemblage, la transmission par valeurs peut se faire simplement en utilisant les registres; par exemple, le programme appelant pourrait comprendre les instructions d'appel suivantes correspondant à l'appel `Valeurs(X,Y)`:

```
LDA       X,d
LDX       Y,d
CALL      Valeurs
```

et dans le sous-programme on utiliserait alors le registre A pour représenter le paramètre formel A et le registre X pour représenter le paramètre formel B.

On peut également employer cette méthode, mais dans le sens contraire, pour transmettre le résultat d'une fonction; ce résultat serait par convention placé dans le registre A. On remarquera que la méthode de transmission des paramètres par les registres n'est acceptable que si le nombre de paramètres est très restreint, en particulier le passage d'un tableau par valeurs exigera une autre méthode de transmission, d'abord à cause du nombre de registres insuffisant et ensuite à cause de la place mémoire importante exigée en double.

9.2.2 Paramètres variables

La transmission des paramètres variables (par référence en C++, ou en mode IN OUT ou OUT en Ada-95) se fait par adresses; en effet les paramètres variables sont tels qu'une modification du paramètre dans le sous-programme modifie le paramètre effectif du programme appelant. Le sous-programme travaille directement sur les variables du programme appelant; ce sera le cas de la procédure C++:

```
void Variables(int &A, int &B);
```

Si cette procédure est appelée par: `Variables(X,Y)`, alors toute modification des paramètres formels A et B dans la procédure se fera directement sur les variables X et Y, paramètres effectifs de l'appel.

En langage d'assemblage, il y a plusieurs façons de traiter ce problème. Une première façon consistera à placer les adresses de tous les paramètres effectifs ensemble dans une table et à transmettre au sous-programme l'adresse de la table dans un registre. Le sous-programme doit alors atteindre les paramètres transmis par adressage indirect. Cependant, et comme nous allons le voir, la méthode classique de passage de paramètres à un sous-programme utilise la pile. Afin de pouvoir l'illustrer, il nous faut d'abord examiner les modes d'adressage sur la pile qui sont disponibles en PEP 8.

9.3 Adressage sur la pile

Comme nous l'avons déjà mentionné au chapitre 6 en parlant des divers modes d'adressage, la pile est un bloc de mémoire alloué spécifiquement par le système d'exploitation et utilisé par les logiciels. Et comme on vient de le voir, la pile est d'abord utilisée pour gérer les appels à des sous-programmes; mais elle est également utilisée pour l'allocation de mémoire aux variables locales des sous-programmes, ainsi que pour le passage des paramètres aux sous-programmes. L'accès aux éléments rangés sur la pile est régi par quatre modes d'adressage supplémentaires dédiés à la pile; ces modes d'adressage viennent compléter les quatre autres modes d'adressage vus au chapitre 6. Dans les figures qui suivent, SP représente le pointeur de pile et sa valeur n'a généralement pas besoin d'être connue, car le calcul des adresses effectives des opérandes au moment de l'exécution s'exécute de façon automatique. Il est cependant possible d'obtenir la valeur du pointeur de pile au moyen de l'instruction `MOVSPA` qui place la valeur de SP dans le registre A. Grâce à cette instruction, il est possible de passer en paramètre un élément situé sur la pile. Notez également dans les figures qui suivent que les éléments de la pile et de la mémoire sont représentés par des paires d'octets (adresses progressant par sauts de 2).

9.3.1 Adressage direct sur la pile

Comme l'adressage direct, l'adressage direct sur la pile est le mode d'adressage sur la pile le plus simple. Dans ce mode, le champ adresse de l'instruction contient un déplacement par rapport au pointeur de pile qui permet de repérer l'opérande situé sur la pile. Au moment de l'exécution de l'instruction, ce déplacement, qui peut être positif ou négatif, est ajouté à la valeur du pointeur de pile pour donner l'adresse de l'opérande, comme le montre la figure 9.2.

Ainsi, si `varia` est déclaré comme :

```
varia:    .EQUATE  6
```

l'instruction :

```
          LDA    varia,s
```

ira placer l'opérande situé à l'adresse SP+6 dans le registre A.

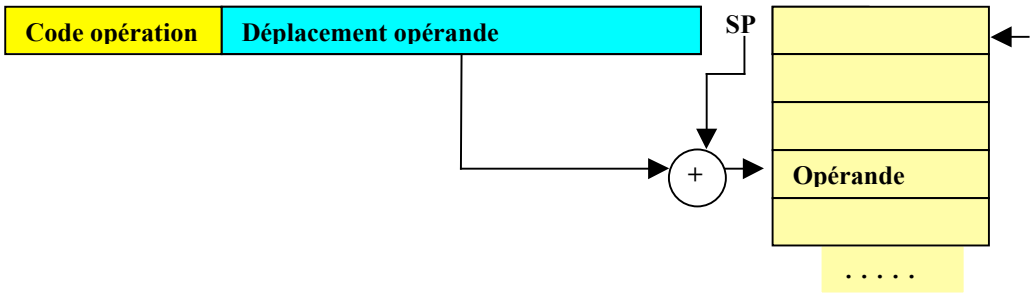


Figure 9.2 Instruction avec adressage direct sur la pile

Exemple

Le petit exemple ci-dessous montre un programme qui lit trois valeurs décimales contribuant à une note et calcule la note finale en faisant la moyenne des deux premières et en lui ajoutant la troisième valeur, avant d'afficher le résultat ainsi calculé. Plutôt que de réserver l'espace mémoire pour les quatre variables nécessaires de la façon habituelle (déclarations de variable avec la directive .WORD), on réserve huit octets sur la pile et on utilise ces octets pour y ranger les valeurs de nos variables. Notez

que nous avons tout de même déclaré une variable chaîne de caractères globale (`total`) pour des raisons de commodité, sinon il aurait fallu copier les valeurs des caractères de la chaîne sur la pile. Les symboles `totalEx`, `final`, `miSess` et `bonus` représentent la position des variables par rapport au pointeur de pile. Notez que l'utilisation de ces variables dans le programme ne change rien d'autre dans les instructions que le mode d'adressage, qui passe de `d` à `s`.

Par exemple, avec les données d'entrée suivantes : **`56 83 4`**, le programme affiche :
`Total examens = 73`

```
;Utilisation de la pile pour les variables globales du programme.
;         Ph. Gabrini  mars 2006
;
totalEx: .EQUATE 0          ; variable locale
final:   .EQUATE 2          ; variable locale
miSess:  .EQUATE 4          ; variable locale
bonus:   .EQUATE 6          ; variable locale
;
Prog:    SUBSP   8,i        ; allouer espace variables locales
         DECI    miSess,s   ; cin >> miSess
         DECI    final,s    ;     >> final
         DECI    bonus,s    ;     >> bonus;
         LDA     miSess,s   ; totalEx = (miSess
         ADDA    final,s    ;    + final)
         ASRA               ;    / 2
         ADDA    bonus,s    ;    + bonus;
         STA     totalEx,s  ;
         STRO    total,d    ; cout << "Total examens = "
         DECO    totalEx,s  ;      << totalEx
         CHARO   '\n',i     ;      << endl;
         ADDSP   8,i        ; libérer espace pile
         STOP               ;
total:   .ASCII  "Total examens = \x00"
         .END
```

9.3.2 Adressage indexé sur la pile

Ce mode d'adressage est semblable au mode d'adressage indexé vu au chapitre 6. Le calcul de l'adresse effective d'un opérande situé sur la pile consiste à ajouter à l'adresse de l'opérande, calculée de la façon qu'on vient de voir, le contenu d'un registre d'index. Dans ce cas, le calcul de l'adresse effective peut alors être résumé par:

adresse effective = pointeur de pile + déplacement + registre d'index.

Le déplacement fait partie de l'instruction et le registre d'index prend des valeurs diverses au cours de l'exécution. On se sert de ce mode d'adressage pour accéder aux valeurs d'une table située sur la pile. Une telle table pourrait représenter un vecteur local à un sous-programme.

La figure 9.3 illustre le calcul de l'adresse effective de l'opérande pour les instructions PEP 8:

```
Vecteur: .EQUATE 4
         LDX     8,i
         LDA     Vecteur,sx       ; prendre Vecteur[X]
```

qui charge dans A le contenu d'un opérande défini avec indexation par le registre X (ce dernier contient la valeur décimale 8) et par le déplacement `Vecteur`. L'adresse effective de l'opérande sera donc de:
`SP + 4 + 8`

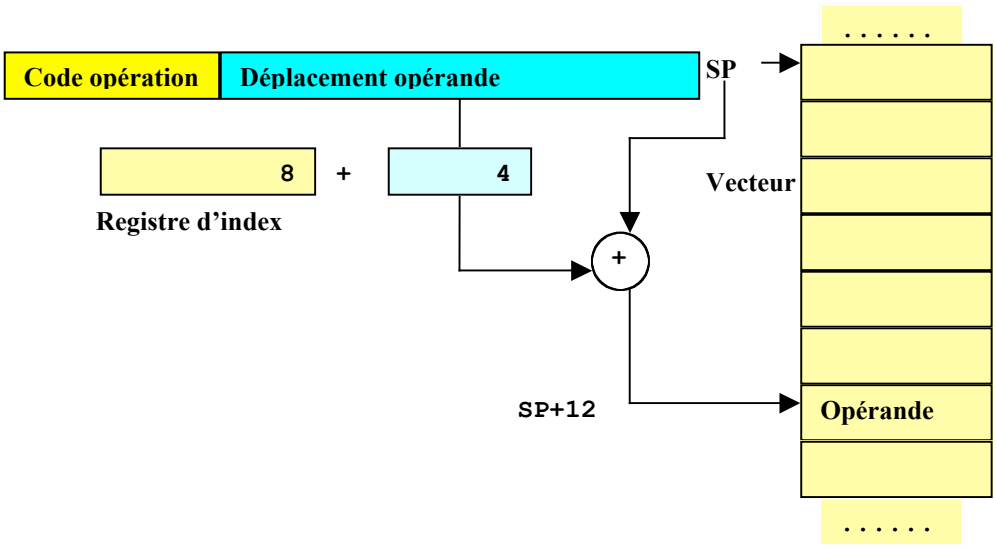


Figure 9.3 Calcul de l'adresse effective de l'opérande indexé sur la pile

Exemple

L'exemple qui suit reprend un des exemples qui a été vu précédemment, l'exemple 5 de la section 7.10 ; cet exemple lit toujours un ensemble de données dans un vecteur de 12 entiers, puis il affiche à l'envers les valeurs rangées dans ce vecteur, précédées de leur indice. Les instructions sont restées les mêmes, à part la première instruction et l'instruction précédant STOP qui ont été ajoutées et le mode d'adressage des opérandes qui est passé de `d` à `s`. En effet, la différence avec l'exemple du chapitre 7 est le fait que nous avons utilisé ici l'espace de la pile pour y ranger le vecteur et la variable index ; pour le reste, le traitement demeure identique. Avec les données 39 25 54 46 51 42 21 18 11 14 23 49, l'exécution du programme donne les résultats suivants.

```
11 49
10 23
9 14
8 11
7 18
6 21
5 42
4 51
3 46
2 54
1 25
0 39
```

```
;Programme qui lit des données, les place dans un vecteur et les
;affiche une par ligne, précédées de leur indice.  Les variables du
;programme sont placées sur la pile.    Ph. Gabrini  Mars 2006
;
TAILLE:     .EQUATE     12      ;taille du vecteur en entiers
vecteur:    .EQUATE     2       ;tableau
index:      .EQUATE     0       ;variable
;
LitVec:     SUBSP   26,i        ;espace index et tableau
```

```
            LDX     0,i         ;int main() {
            STX     index,s     ;
Boucle1:    CPX     TAILLE,i    ;  for(i = 0; i < TAILLE; i++){
            BRGE    FinBouc1    ;
            ASLX                ;    (entier = 2 octets)
            DECI    vecteur,sx  ;    cin >> vector[i];
            LDX     index,s     ;
            ADDX    1,i         ;
            STX     index,s     ;
            BR      Boucle1     ;  }
FinBouc1:   LDX     TAILLE,i    ;  for(i = TAILLE-1; i >= 0; i--){
            SUBX    1,i         ;
            STX     index,s     ;
            CHARO   '\n',i      ;    cout << endl;
Boucle2:    CPX     0,i         ;
            BRLT    FinBouc2    ;
            DECO    index,s     ;    cout << index;
            CHARO   ' ',i       ;         << ' '
            ASLX                ;    (entier = 2 octets)
            DECO    vecteur,sx  ;         << vector[i]
            CHARO   '\n',i      ;         << endl;
            LDX     index,s     ;
            SUBX    1,i         ;
            STX     index,s     ;
            BR      Boucle2     ;  }
FinBouc2:   ADDSP   26,i        ;  nettoyer pile
            STOP                ;  return 0;}
            .END
```

9.3.3 Adressage indirect sur la pile

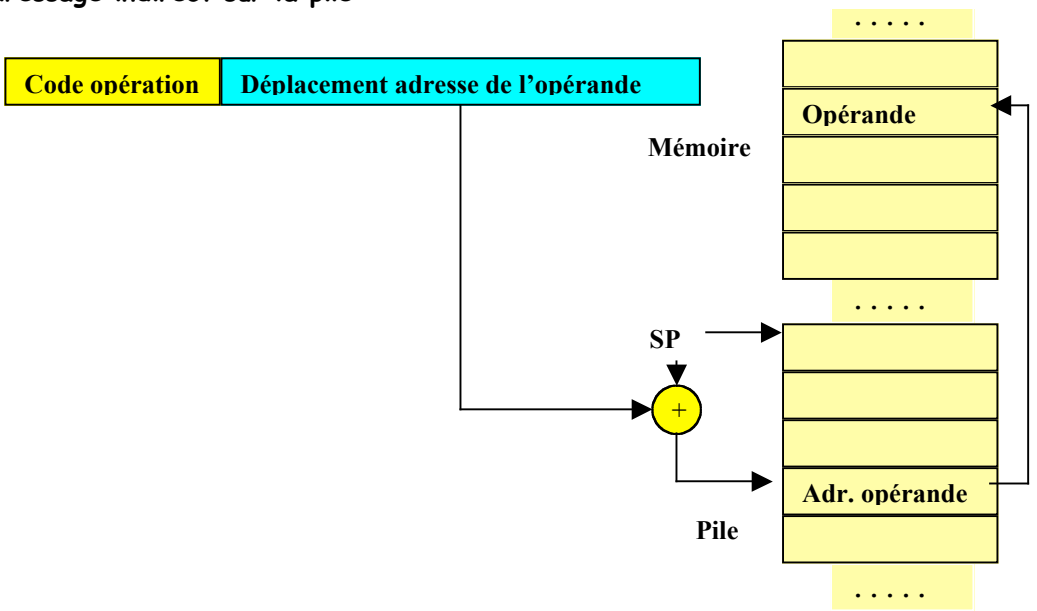


Figure 9.4 Adressage indirect sur la pile

Dans ce mode d'adressage, la pile ne contient pas l'opérande, comme dans le cas de l'adressage direct sur la pile, mais l'adresse de l'opérande.

La figure 9.4 représente l'adressage indirect sur la pile. Si `AdVal` représente la position sur la pile de l'adresse d'une valeur située dans un programme appelant et si `AdVal` est défini par :

```
AdVal :    .EQUATE  6
```

l'instruction :

```
STA  AdVal,sf
```

fait ranger le contenu du registre A dans l'opérande. L'adresse calculée SP + AdVal repère la position sur la pile de l'adresse de l'opérande, à partir de laquelle on aboutit à l'opérande, rangé quelque part en mémoire.

9.3.4 Adressage indirect indexé sur la pile

Ce mode d'adressage est utilisé lorsqu'on passe l'adresse d'un vecteur à un sous-programme, lequel doit alors travailler directement dans le vecteur qui lui a été ainsi passé.

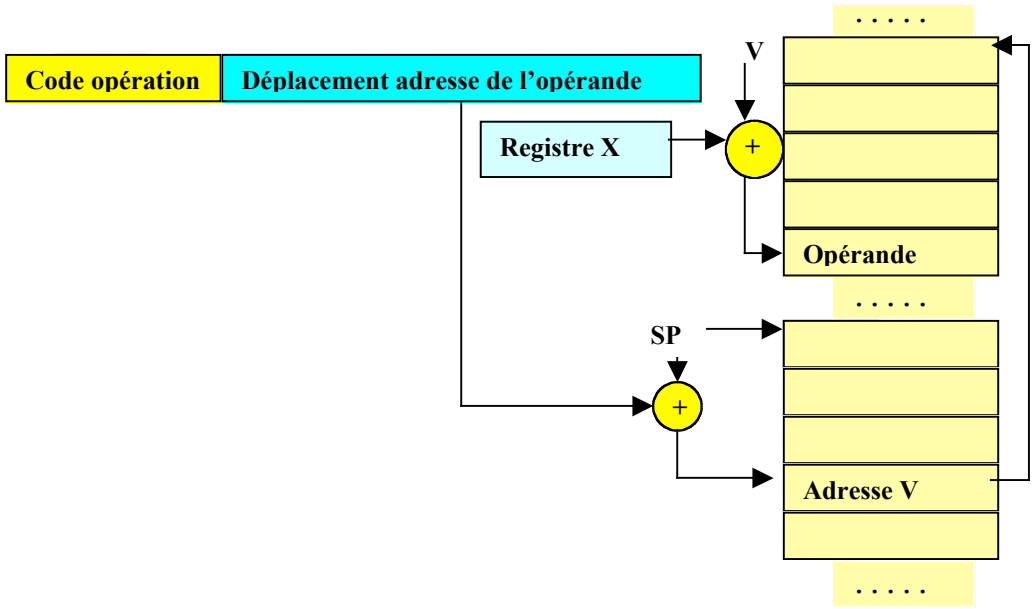


Figure 9.5 Adressage indirect indexé sur la pile

La figure 9.5 représente l'adressage indirect indexé sur la pile. Si `AdVect` représente l'adresse d'un vecteur V situé dans un programme appelant et si `AdVect` est défini par :

```
AdVect:    .EQUATE  6
```

Et si le registre d'index contient la valeur 8, l'instruction :

```
STA       AdVect,sxf
```

fait ranger le contenu du registre A dans l'opérande V[8] (les éléments du vecteur V sont repérés par des indices aux octets). Dans un premier temps, l'adresse calculée est SP + AdVect; ceci conduit à l'adresse du vecteur sur la pile. De là, on atteint le vecteur, que l'on indice alors par le registre d'index.

9.4 Réalisation du passage de paramètres

Possédant maintenant les modes d'adressage permettant d'utiliser la pile, nous pouvons poursuivre la présentation des paramètres valeurs et variables et de leurs passages aux sous-programmes.

9.4.1 Passage d'un paramètre valeur

Nous prendrons comme exemple, celui d'un programme permettant d'afficher un histogramme à partir de données numériques. La première valeur indiquera le nombre de valeurs numériques à traiter. Pour chaque valeur numérique lue, on tracera une ligne d'astérisques comprenant le nombre d'astérisques correspondant à la valeur. Avec les données d'entrée : 5 4 3 7 5 9, la sortie sera :

```
****
***
*******
*****
*********
```

Le programme principal comprendra une boucle qui lira les valeurs et appellera un sous-programme pour tracer la ligne d'astérisques correspondant à la valeur lue; la valeur sera passée au sous-programme comme paramètre valeur. Le sous-programme recevra un paramètre qu'il trouvera sur la pile sous l'adresse de retour empilée par l'instruction `CALL`. Il réservera un espace de deux octets au sommet de la pile (au dessus de l'adresse de retour) pour un compteur de boucle local au moyen de l'instruction `SUBSP`. Durant l'exécution du sous-programme, après l'instruction `SUBSP`, la pile aura l'aspect illustré par la figure 9.6.

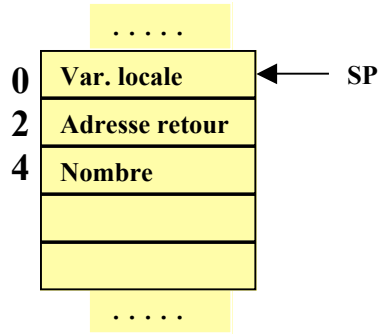


Figure 9.6 État de la pile au cours d'un appel

```
; Programme permettant d'afficher un histogramme des valeurs lues
; (tiré de "An introduction to computer science with
;  Modula2" par Adams, Gabrini, Kurtz, D.C. Heath 1988)
;   Ph. Gabrini  2005
                            ;int main (){
Histogra: DECI    nbBarres,d ;  cin >> nbBarres;
          LDA     1,i        ;  for(index = 1;
          STA     compte,d   ;
Boucle:   CPA     nbBarres,d ;       index <= nbPts; index++){
          BRGT    Fini       ;
          DECI    valeur,d   ;     cin >> valeur;
          LDA     valeur,d   ;     paramètre passé par valeur
          STA     -2,s       ;
          SUBSP   2,i        ;     empiler paramètre
          CALL    AfficBar   ;     AfficBar(valeur);
          ADDSP   2,i        ;     désempiler paramètre
          LDA     compte,d   ;
          ADDA    1,i        ;
          STA     compte,d   ;
          BR      Boucle     ;  }
Fini:     STOP               ;}
nbBarres: .WORD    0         ;nombre de barres
```



```
valeur:   .WORD    0         ;valeur lue
compte:   .WORD    0         ;compteur
;
; Sous-programme d'affichage du nombre d'astérisques correspondant
; à la valeur du paramètre
                             ;void AfficBar(int n){
AfficBar: SUBSP   2,i        ;  allouer variable locale sommet pile
          LDA     1,i        ;  for(index = 1;
          STA     0,s        ;    sommet pile
Repete:   CPA     4,s        ;      index <= n; index++){
          BRGT    Retour     ;
          CHARO   '*',i      ;     cout << '*';
          LDA     0,s        ;    sommet pile
          ADDA    1,i        ;
          STA     0,s        ;    sommet pile
          BR      Repete     ;  }
Retour:   CHARO   '\n',i     ;  cout << endl;
          RET2               ;  return} // relâche var. locale
          .END
```

Dans le sous-programme, on accède au paramètre par l'expression `4,s` et à la variable locale par `0,s`, ce qui est illustré par la figure 9.6. L'instruction de retour `RET2` désempile la variable locale allouée au début du sous-programme; l'adresse de retour se retrouve alors au sommet de la pile et est désempilée et utilisée pour retourner au programme principal. Dès ce retour nous désempilons le paramètre, empilé avant l'appel, dont nous n'avons plus besoin.

9.4.2 Passage de plusieurs paramètres au moyen d'une seule adresse

Nous prendrons l'exemple suivant où les instructions d'appel *empilent* une seule adresse comme paramètre d'appel (tout en réservant un espace sur la pile pour le résultat de la fonction Facture)

```
          LDA    typ1,i     ;Liste des cinq mots des types de service
          STA    liste1,d   ;  premier paramètre
          LDA    temps,i    ;Durée
          STA    liste2,d   ;  deuxième paramètre
          LDA    result,i   ;Résultat
          STA    liste3,d   ;  troisième paramètre

          LDA    liste1,i   ; adresse liste de 3 pointeurs
          STA    -4,s       ; empilée après espace pour code résultat
          SUBSP  4,i        ; ajuster pointeur pile
          CALL   Facture    ;
          .......
typ1:     .WORD   3
typ2:     .WORD   4
typ3:     .WORD   5
typ4:     .WORD   2
typ5:     .WORD   0
temps:    .WORD   10
result:   .WORD   0
liste1:   .BLOCK  2    ;Liste des trois pointeurs
liste2:   .BLOCK  2
liste3:   .BLOCK  2
```

permettant au sous-programme Facture d'atteindre les paramètres dont la structure est un peu complexe (voir figure 9.7) par des instructions du type suivant.

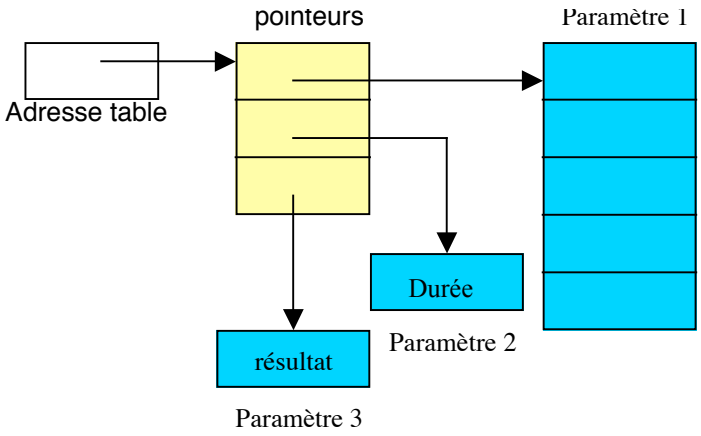


Figure 9.7 Structure des paramètres de l'appel à Facture

```
LDX     0,i         ; index = 0;
LDA     liste,sxf   ; adresse table
STA     table,s     ;  conservée
ADDX    2,i         ; index = 1;
LDA     liste,sxf   ; adresse durée
STA     duree,s     ;  conservée
ADDX    2,i         ; index = 2;
LDA     liste,sxf   ; adresse résultat
STA     res,s       ;  conservée
LDA     duree,sf    ; valeur durée
STA     duree,s     ;  conservée
```

On peut alors accéder aux paramètres de Facture avec le code ci-dessus (le programme Facture complet se trouve au chapitre 10) en utilisant les adresses ainsi récupérées et sauvegardées localement (dans `table`, `duree` et `res`). On notera l'obligation d'utiliser les modes d'adressage sur la pile indirect et indirect indexé pour atteindre les valeurs des paramètres comme `duree` (indirection). Le mode d'adressage direct sur la pile est réservé aux variables locales du sous-programme.

9.4.3 Passage de paramètres par la pile

Le passage des paramètres par la pile est une technique efficace qui est utilisée par la plupart des langages de programmation évolués. La différence entre paramètres valeurs et paramètres variables se situera uniquement à ce qu'on empilera sur la pile: des valeurs ou des adresses.

Avant d'exécuter une instruction d'appel, on empile les paramètres. Par exemple:

```
LDA     duree,d     ;  empiler Durée
STA     -4,s        ;  après espace résultat
LDA     375,i       ;  empiler 375
STA     -6,s        ;  par dessus Durée
SUBSP   6,i         ;  ajuster pointeur pile
CALL    Multipli    ;  multiplier 2 paramètres
LDA     0,s         ;  récupérer résultat fonction
ADDSP   2,i         ;  désempiler
```


L'empilage d'un paramètre requiert une instruction de rangement sur la pile avec un indice négatif (puisqu'on empile), et lorsque tous les paramètres ont été ainsi copiés à leurs places dans la pile on utilise une instruction d'ajustement du pointeur de pile (`SUBSP`). Nous avons ici l'exemple d'un appel de fonction qui retourne son résultat sur la pile. Le premier emplacement sur la pile est l'espace réservé pour ce résultat (`-2,s`), le second emplacement accueille le premier paramètre duree (`-4,s`) et le troisième emplacement accueille le second paramètre 375 (`-6,s`).

À l'entrée dans le sous-programme, la pile a l'aspect illustré par la figure 9.8.

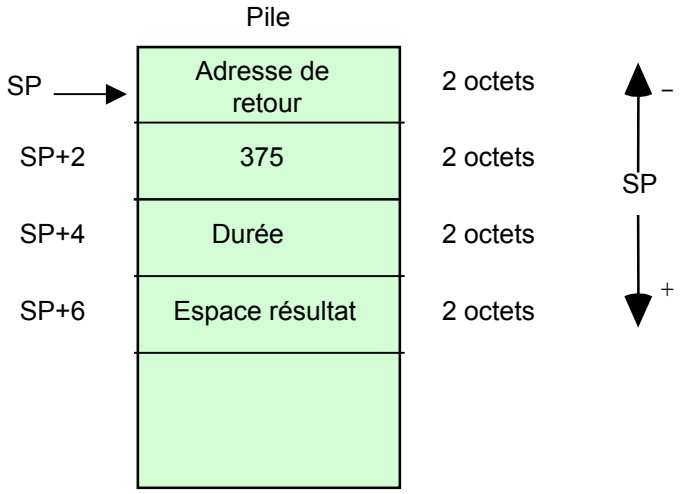


Figure 9.8 Aspect du sommet de la pile après un appel

Une fois dans le sous-programme, on peut accéder aux paramètres placés sur la pile en calculant leur position et en utilisant l'adressage en mode pile (`,s`). Le sommet de la pile comprend l'adresse de retour qu'il faut sauter. Dans l'exemple de la figure 9.8, on atteint le deuxième paramètre par (`2,s`) , le premier par (`4,s`), et l'espace réservé pour le résultat par (`6,s`), comme par exemple dans les instructions suivantes.

```
LDA     0,i         ;  valeur zéro
STA     6,s         ;  res = 0;
LDX     4,s         ;  premier paramètre
LDA     2,s         ;  deuxième paramètre
```

Lorsqu'on retourne au programme appelant il faut "nettoyer" la pile en en enlevant les deux paramètres. Ceci peut être fait du sous-programme juste avant l'instruction de retour `RETn`, en déplaçant l'adresse de retour. Dans le cas de l'exemple de la figure 9.8, il faudrait copier l'adresse de retour dans l'espace du paramètre Durée et ajuster le pointeur de pile en exécutant les instructions:

```
LDA     0,s       ; adresse retour (au sommet)
STA     4,s       ; déplacée (dans l'espace de duree)
RET4              ; nettoyer pile et retour
```

En réalité, les choses sont un peu plus compliquées si la programmation est faite correctement. En particulier, comme on le verra ci-dessous, le sous-programme doit sauvegarder les registres qu'il utilise et les restaurer avant le retour: l'utilisateur n'aimerait pas voir le contenu de ses registres changer au retour d'un appel de sous-programme sans qu'il n'en soit responsable. Cette sauvegarde est faite en utilisant la pile et complique quelque peu la situation, en particulier en ce qui concerne le calcul de la position des paramètres dans la pile car il faut alors sauter par dessus l'espace utilisé pour la sauvegarde des registres.

Selon les systèmes, les paramètres sont empilés sur la pile dans l'ordre de leur définition en langage évolué (Pascal, Ada 95) ou dans l'ordre inverse de leur définition (C). Il sera nécessaire de connaître cet ordre d'empilage dans le cas où l'on doit utiliser un système composé de modules écrits en différents langages de programmation.

9.4.4 Passage de paramètres qui se trouvent déjà sur la pile

Lorsqu'on doit passer un paramètre variable à un sous-programme, il suffit de placer son adresse sur la pile juste avant l'appel. Il peut arriver que le paramètre que l'on désire passer à un sous-programme soit lui-même une variable locale du sous-programme comprenant le nouvel appel, et par conséquent se trouve déjà sur la pile. Pour pouvoir passer ce paramètre au second sous-programme, il faut calculer son adresse et empiler cette adresse. Le calcul de l'adresse d'un élément se trouvant sur la pile se fait facilement puisque l'instruction `MOVSPA` place la valeur du pointeur de pile SP dans le registre A. Les éléments se trouvant sur la pile sont identifiés relativement au pointeur de pile et on peut utiliser la valeur de ces repères relatifs pour calculer leur adresse.

Supposez que nous soyons dans le sous-programme `Calcul` et que nous y ayons déclaré trois variables locales sur la pile, qui a alors l'aspect indiqué par la partie gauche de la figure 9.9.

```
var1:    .EQUATE    0
var2:    .EQUATE    2
vecteur:.EQUATE     4
```

Pour appeler le sous-programme `Tri` à partir du sous-programme `Calcul` avec `vecteur` comme paramètre variable, il suffit de calculer l'adresse de vecteur et de l'empiler de la façon suivante :

```
MOVSPA              ; copie valeur SP
ADDA      vecteur,i; ajouter le déplacement
STA       -2,s      ; placer sur la pile
SUBSP     2,i       ; ajuster pointeur pile
CALL      Tri       ; Tri(vecteur);
```

En arrivant dans le sous-programme `Tri` la pile a l'aspect de la partie droite de la figure 9.9. Le vecteur ainsi passé peut être manipulé à partir de son adresse, exactement comme s'il s'agissait d'une variable du programme principal, par adressage indirect indexé sur la pile. Dans ce sous-programme Tri on aura par exemple :

```
AdVect   .EQUATE    2
         .......
         LDX       6,i          ; 4éme élément du vecteur d'entiers
         LDA       AdVect,sxf   ; vecteur[3]
```

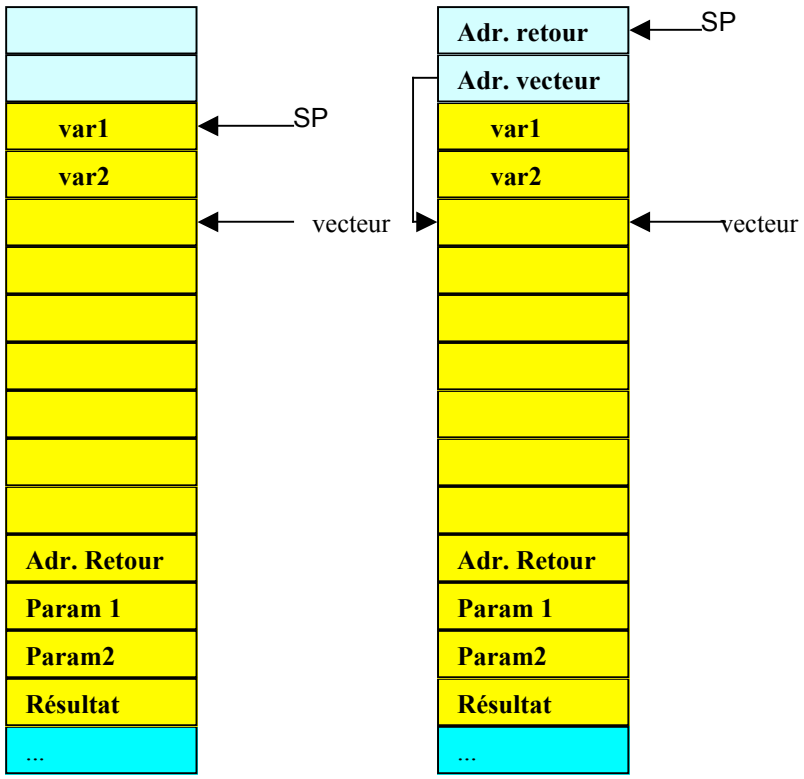



Figure 9.9 Passage d'un paramètre situé sur la pile

9.5 Conventions d'utilisation des registres

Tout programme ou sous-programme utilise les registres, on ne peut en effet rien faire sans cela. Si un programme appelle un sous-programme et que ce dernier utilise tous les registres, le retour au programme risque d'être désagréable: les contenus des registres au moment de l'appel ont été irrémédiablement perdus et dans ces conditions il va vraisemblablement être impossible de continuer le traitement. Le fonctionnement du programme ne sera pas rendu impossible si seuls les registres utilisés pour la transmission des paramètres sont modifiés; cependant, ceci est assez restrictif.

Pour éviter les ennuis causés par le partage involontaire des registres, on utilise un certain nombre de *conventions* concernant l'utilisation des registres. Selon les systèmes et le nombre de registres disponibles, ces conventions peuvent varier. Dans le cas de PEP 8, qui n'a que deux registres, la situation est plus simple. On convient simplement de sauvegarder et de restaurer les valeurs des registres A et X.

9.5.1 Appels des sous-programmes et passage des paramètres

Le registre A peut contenir le résultat (16 bits) d'une fonction; la valeur antérieure de ce registre sera donc détruite par une fonction qui y placera son résultat.

Lorsqu'un sous-programme n'a qu'un ou deux paramètres, il semble plus simple de passer ces paramètres dans les registres A et X, par exemple; si les valeurs sont des valeurs de 16 bits, elles pourraient en effet être placées dans des registres. On pourrait également transmettre les adresses des deux paramètres dans les mêmes registres. En fait, même lorsqu'un sous-programme n'a qu'un seul paramètre, *on applique la méthode standard:* le paramètre est empilé. Si ceci semble un peu lourd, les avantages fournis par le respect des conventions sont suffisamment importants pour tenir cette lourdeur pour négligeable.

9.5.2 Sauvegarde des registres

Comme on vient de le voir, il est très commode pour un programme appelant un sous-programme d'avoir la certitude au retour du sous-programme que les registres possèdent le même contenu qu'avant l'appel. Comme le sous-programme doit utiliser des registres pour pouvoir fonctionner, on utilise un système de sauvegarde et de restauration des registres *dans le sous-programme appelé.* Pour les langages de programmation évolués, le compilateur engendre le code nécessaire et le programmeur n'a pas à s'en soucier. En assembleur ce n'est pas la même chose!

Pour sauvegarder les registres, on utilise tout au début du sous-programme les instructions `STA` et `STX` et la pile comme zone de sauvegarde. Notons, en passant, que certains processeurs possèdent des instructions de sauvegarde et de restauration des registres.

Ces mêmes registres seront restaurés par les instructions `LDA` et `LDX` juste avant le retour.

9.5.3 Zone locale des sous-programmes

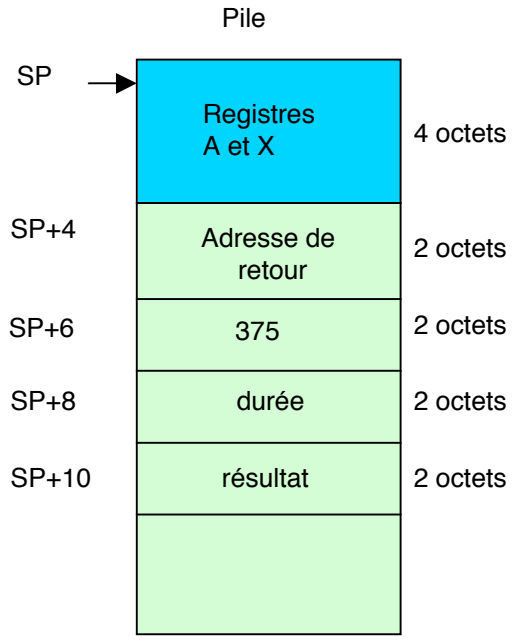


Figure 9.10 Aspect du sommet de la pile après appel et sauvegarde

Dans un sous-programme, une fois les registres sauvegardés, la pile contient un certain nombre d'éléments. Si nous reprenons notre exemple de la figure 9.8 en y ajoutant la sauvegarde des 2

registres que nous venons de voir, nous obtenons la figure 9.10. On voit sur cette figure que pour atteindre la valeur 375, il nous faut maintenant utiliser un déplacement de 6, ainsi qu'un déplacement de 8 pour la durée.

Dans l'exemple de la figure 9.10, les 4 octets du sommet de la pile appartiennent à l'espace du sous-programme. Si ce sous-programme utilise des variables locales, ces dernières devraient occuper de l'espace supplémentaire sur la pile; rappelez-vous ce qu'on a dit au sujet des adresses de retour anciennement conservées dans les sous-programmes : ceci interdisait la récursivité. Il en serait de même s'il n'y avait qu'une copie des variables locales, ce qui indique bien que ces dernières doivent être conservées sur la pile. De cette façon, chaque appel de sous-programme possède son propre espace de travail local. De plus, si ce sous-programme appelle lui aussi un autre sous-programme, d'autres éléments doivent être empilés au dessus de cet espace.

Pour faciliter les situations semblables, on crée systématiquement un espace local pour les paramètres d'un sous-programme et ses variables locales. L'espace (aussi appelé *contexte*) du sous-programme sur la pile est généralement utilisé pour y ranger les paramètres, le résultat s'il s'agit d'une fonction, l'adresse de retour et les variables locales. Tout ceci constitue un "*cadre*" ("*stack frame*") sur la pile, comme le montre la figure 9.11.

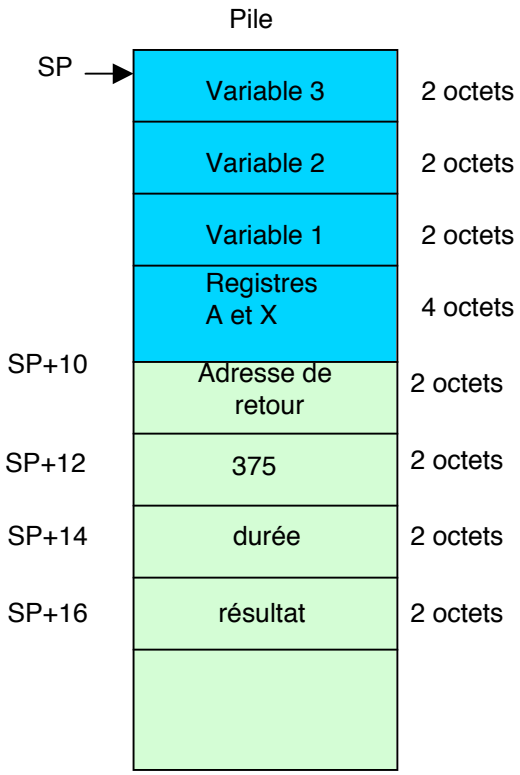


Figure 9.11 Cadre de pile avec espace local

Dans le cas de fonctions qui doivent retourner un résultat, il arrive parfois que ce résultat soit retourné dans le registre A. Mais il arrive aussi, et en fait la plupart du temps, que le résultat soit retourné au sommet de la pile: dans ce cas il faut prévoir un espace supplémentaire sur la pile pour ce

résultat, comme dans le cas de notre exemple. Cet espace est habituellement réservé en premier, avant les paramètres, car il doit se trouver au sommet de la pile au retour de la fonction.

Pour établir ce cadre de pile dans un sous-programme, il suffit d'abord de réserver l'espace supplémentaire au sommet de la pile au moyen d'une instruction `SUBSP`, puis de définir dans le sous-programme un certain nombre de constantes au moyen de la directive `.EQUATE` qui correspondent aux divers déplacements dans la pile à partir de la nouvelle position du sommet et représentent l'image du sommet de la pile, comme par exemple dans le cas de la figure 9.11.

```
variabl3:.EQUATE 0         ; variable locale no 3
variabl2:.EQUATE 2         ; variable locale no 2
variabl1:.EQUATE 4         ; variable locale no 1
sauveX:  .EQUATE 6         ; sauvegarde registre X
sauveA:  .EQUATE 8         ; sauvegarde registre A
adRetour:.EQUATE 10        ; adresse de retour
op2:     .EQUATE 12        ; opérande 2
op1:     .EQUATE 14        ; opérande 1
resu:    .EQUATE 16        ; résultat
```

On atteint alors les diverses composantes de cet espace local au moyen de l'adressage direct sur la pile, par exemple.

```
STA       variabl1,s         ; variable1 = valeur;
```

9.6 Partage des responsabilités

Pour accomplir les liaisons requises en respectant les conventions standard que nous venons de voir, le programme appelant et le sous-programme doivent chacun faire un certain nombre de choses. En langages de programmation évolués, ces choses sont faites automatiquement par le compilateur. En assembleur, c'est au programmeur de le faire!

9.6.1 Programme appelant

Le programme appelant doit préparer le terrain pour que le sous-programme puisse procéder de façon standard à la sauvegarde et à la restauration des registres. On doit y faire les choses suivantes:

- s'il s'agit d'une fonction, réserver l'espace pour le résultat sur la pile
- définir les paramètres, et placer leurs adresses ou leurs valeurs sur la pile
- effectuer le branchement au sous-programme par un `CALL` au sous-programme
- récupérer le résultat de la fonction, si c'est le cas, et l'enlever de la pile

Ceci peut être accompli, par exemple, par les instructions suivantes:

```
LDA      duree,d     ;  empiler Durée
STA      -4,s        ;  après espace résultat
LDA      375,i       ;  empiler 375
STA      -6,s        ;  par dessus Durée
SUBSP    6,i         ;  ajuster pointeur pile
CALL     Multipli    ;  multiplier 2 paramètres
```



```
LDA     0,s         ;  récupérer résultat fonction
ADDSP   2,i         ;  désempiler
```

Note: si le résultat occupe plus d'un mot, il faudra remplacer l'instruction « `LDA 0,s` » ci-dessus par une boucle pour le copier dans la zone voulue et l'enlever de la pile.

9.6.2 Sous-programme

Le sous-programme a pour principale responsabilité la sauvegarde et la restauration des registres. On doit y faire les choses suivantes:

- établir l'espace local et les paramètres en définissant un cadre de pile
- sauvegarder les registres sur la pile
- traiter le problème donné
- restaurer les contenus antérieurs des registres à partir de la pile
- nettoyer la pile (en enlevant les paramètres empilés)
- retourner au programme appelant par un `RETn`

Les trois premières instructions du sous-programme (avec des opérandes appropriés) effectuent le nécessaire au début du sous-programme et constituent le *prologue* du sous-programme, par exemple:

```
sauveX:  .EQUATE 0          ; sauvegarde X
sauveA:  .EQUATE 2          ; sauvegarde A
AdRetour:.EQUATE 4          ; adresse retour
op2:     .EQUATE 6          ; opérande 2
op1:     .EQUATE 8          ; opérande 1
resu:    .EQUATE 10         ; résultat
;
Multipli: SUBSP   4,i          ; espace local sauvegarde
          STA     sauveA,s     ; sauvegarde A
          STX     sauveX,s     ; sauvegarde X
```

Les instructions `STr` effectuent la sauvegarde des registres. En fin de sous-programme, on retrouvera les instructions ci-dessous qui constituent l'*épilogue* du sous-programme:

```
LDA     adRetour,s  ; adresse retour
STA     op1,s       ; déplacée
LDA     sauveA,s    ; restaure A
LDX     sauveX,s    ; restaure X
ADDSP   8,i         ; nettoyer pile
RET0
```

Elles restaurent les registres, annulent le cadre de pile courant, éliminent les paramètres de la pile et retournent au programme appelant. Notez que si le nombre d'octets à enlever de la pile est inférieur à 8, l'instruction `RETn`, où `n` est remplacé par le nombre d'octets à enlever de la pile, effectue le nettoyage sans besoin d'une instruction `ADDSP`

On notera que les conventions habituelles de liaison entre programmes et sous-programmes autorisent tous les appels de sous-programme, *y compris les appels récursifs*.

9.6.3 Exemple de sous-programmes

Dans le nouvel exemple qui suit, nous reprenons l'affichage en ordre de deux valeurs lues, vu dans le premier exemple de la section 7.10, mais, cette fois-ci, en utilisant des sous-programmes : un premier sous-programme qui permet d'ordonner deux valeurs et un second sous-programme qui permet d'échanger les valeurs de deux variables (paramètres variables). De plus, le programme principal utilise la pile pour ranger les valeurs des variables globales, à l'exception des chaînes de caractères. Le programme réserve 4 octets sur la pile pour ses variables globales dont il lit ensuite les valeurs. Il empile alors les adresses des deux variables globales et appelle le sous-programme Ordonner. Au retour de l'appel il affiche les deux valeurs dans l'ordre. Voici un exemple d'exécution de ce programme :

```
Donnez un entier: 34
Donnez un entier: 12
En ordre: 12, 34
```

```
;Tout sur la pile: un exemple où toutes les variables globales et locales
;sont rangées sur la pile du système.
;              Ph. Gabrini mars 2006
;
nb1:     .EQUATE  0         ; variable globale
nb2:     .EQUATE  2         ; variable globale
;
Program: SUBSP   4,i        ; espace pour variables globales
         STRO    donnez,d   ; cout << "Donnez un entier: ";
         DECI    nb1,s      ; cin >> nb1;
         STRO    donnez,d   ; cout << "Donnez un entier: "
         DECI    nb2,s      ; cin >> nb2;
         MOVSPA             ; calcule adresse nb1
         ADDA    nb1,i      ;
         STA     -2,s       ; empile adresse nb1
         MOVSPA             ; calcule adresse nb2
         ADDA    nb2,i      ;
         STA     -4,s       ; empile adresse nb2
         SUBSP   4,i        ; empile paramètres
         CALL    Ordonner   ; Ordonner(nb1, nb2)
         STRO    ordre,d    ; cout << "En ordre: "
         DECO    nb1,s      ;      << nb1
         STRO    virg,d     ;      << ", "
         DECO    nb2,s      ;      << nb2
         CHARO   '\n',i     ;      << endl;
         ADDSP   4,i        ; vider pile
         STOP
donnez:  .ASCII  "Donnez un entier: \x00"
ordre:   .ASCII  "En ordre: \x00"
virg:    .ASCII  ", \x00"
```

Le sous programme `Ordonner` reçoit deux paramètres, qui sont les adresses de deux variables entières (paramètres variables). Après son prologue (réservation de quatre octets et sauvegarde des registres), il compare les valeurs de ses deux paramètres (obtenues par indirection sur les adresses empilées) et, si elles sont en ordre, retourne au programme appelant sans rien faire d'autre que de nettoyer la pile par son épilogue. Si elles ne sont pas en ordre, le sous-programme `Ordonner` appelle un autre sous-

programme, `Echanger`, dont le rôle est d'échanger les valeurs des deux variables dont les adresses se trouvent sur la pile. Avant d'appeler `Echanger`, `Ordonner` empile les adresses des deux variables par simple copie des adresses se trouvant déjà dans son espace local. Au retour de l'appel, l'épilogue du sous-programme déplace l'adresse de retour dans le premier paramètre empilé, restaure les registres, et ajuste le pointeur de pile pour nettoyer la pile, avant de retourner au programme appelant.

```
;         void Ordonner(int& entier1, int& entier2)
OregX:    .EQUATE 0           ; sauvegarde registre X
OregA:    .EQUATE 2           ; sauvegarde registre A
Oadret:   .EQUATE 4           ; adresse retour
Oentier2:.EQUATE 6            ; adresse paramètre 2
Oentier1:.EQUATE 8            ; adresse paramètre 1
Ordonner:SUBSP    4,i         ; zone sauvegarde
          STA     OregA,s     ; sauvegarde A
          STX     OregX,s     ; sauvegarde X
          LDA     Oentier1,sf; if(entier1 > entier2)
          CPA     Oentier2,sf;
          BRLE    FinIf       ;
          LDA     Oentier1,s ;    empiler adresse entier1
          STA     -2,s        ;
          LDA     Oentier2,s ;    empiler adresse entier2
          STA     -4,s        ;
          SUBSP   4,i         ;    empiler paramètres
          CALL    Echanger    ;    Echanger(entier1, entier2)
FinIf:    LDA     Oadret,s    ; déplacer adresse retour
          STA     Oentier1,s ;
          LDA     OregA,s     ; restaurer A
          LDX     OregX,s     ; restaurer X
          ADDSP   8,i         ; désempiler paramètres
          RET0                ; désempiler adresse retour
```

Le sous-programme `Echanger` effectue l'échange des valeurs des deux variables dont les adresses lui sont passées en paramètres. Pour ce faire, il utilise une variable locale, `Etemp`, pour laquelle il a réservé deux octets dans son prologue ; il obtient les valeurs à échanger par indirection sur les adresses empilées. Une fois l'échange effectué, l'épilogue déplace l'adresse de retour dans le premier paramètre empilé, restaure les registres, et ajuste le pointeur de pile pour nettoyer la pile avant de retourner au programme appelant.

```
;         void Echanger(int& p1, int& p2)
Etemp:    .EQUATE 0           ; variable locale
EregX:    .EQUATE 2           ; registre X
EregA:    .EQUATE 4           ; registre A
Eadret:   .EQUATE 6           ; adresse retour
Ep2:      .EQUATE 8           ; paramètre 2
Ep1:      .EQUATE 10          ; paramètre 1
Echanger:SUBSP    6,i         ; allouer local
          STA     EregA,s     ; sauvegarde A
          STX     EregX,s     ; sauvegarde X
          LDA     Ep1,sf      ; temp = p1;
          STA     Etemp,s     ;
          LDA     Ep2,sf      ; p1 = p2;
          STA     Ep1,sf      ;
          LDA     Etemp,s     ; p2 = temp;
```

```
        STA     Ep2,sf      ;
        LDA     Eadret,s    ; déplace adresse de retour
        STA     Ep1,s       ;
        LDA     EregA,s     ; restaurer A
        LDX     EregX,s     ; restaurer X
        ADDSP   10,i        ; ajuster pointeur pile
        RET0                ; désempiler adresse retour
        .END
```

9.7 Sous-programmes pour la sortie d'un mot en hexadécimal

La figure 8.10 a introduit un sous-programme de sortie d'un mot sous forme de quatre chiffres hexadécimaux, dont nous avons réservé l'étude jusqu'ici pour être sûr d'avoir couvert d'abord les concepts de base relatifs aux sous-programmes. Le sous-programme `HEXO` reçoit un mot en paramètre (`Hmot`), établit le contexte local avec deux mots de sauvegarde des registres et deux octets, `Hnomb` et `Hdroit`.

```
;Sortie hexadécimale.
;Format sortie:  un mot en 4 caractères hexa
;
Hnomb:   .EQUATE 0
Hdroit:  .EQUATE 1
HregX:   .EQUATE 2
HregA:   .EQUATE 4
Hretour: .EQUATE 6
Hmot:    .EQUATE 8          ; mot à sortir
                            ;{
HEXO:    SUBSP   6,i        ;  espace local
         STA     HregA,s    ;  sauvegarde A
         STX     HregX,s    ;  sauvegarde X
         LDA     Hmot,s     ;  A = mot;
         STA     Hnomb,s    ;  opérande
         LDBYTEA Hnomb,s    ;  octet gauche dans bas de A
         ASRA               ;  Décale 4 bits
         ASRA               ;
         ASRA               ;
         ASRA               ;
         CALL    SortAC     ;  sort premier carac. hexa
         LDBYTEA Hnomb,s    ;  octet gauche dans bas de A
         CALL    SortAC     ;  sort second carac hexa
         LDBYTEA Hdroit,s   ;  octet droit dans bas de A
         ASRA               ;  Décale 4 bits
         ASRA               ;
         ASRA               ;
         ASRA               ;
         CALL    SortAC     ;  sort troisième carac. hexa
         LDBYTEA Hdroit,s   ;  octet droit dans bas de A
         CALL    SortAC     ;  sort quatrième carac. hexa
         LDA     Hretour,s  ;  adresse retour
         STA     Hmot,s     ;  déplacée
         LDA     HregA,s    ;  restaure A
         LDX     HregX,s    ;  restaure X
         ADDSP   8,i        ;  nettoyer pile
         RET0               ;}//HEXO
```

Une fois les registres sauvegardés, on copie le mot passé en paramètre dans la zone de deux octets située au sommet de la pile. On prend d'abord le premier de ces octets, `Hnomb`, dont on affiche la moitié gauche obtenue après quatre décalages à droite (le registre A ayant été mis à zéro au préalable). Le sous-programme interne `SortAC` permet d'afficher cette valeur en hexadécimal. On affiche ensuite la moitié droite de ce premier octet, toujours par appel à `SortAC`.

On applique un processus semblable au second octet, `Hdroit` : première moitié obtenue après quatre décalages à droite et affichée par un appel à `SortAC`, seconde moitié sortie par appel à `SortAC`. On déplace ensuite l'adresse de retour, on restaure les registres et on retourne au programme appelant.

```
; Sous-programme interne pour sortir en hexadécimal les 4 bits
; de droite de A
Hcar:     .EQUATE 0          ;{
SortAC:   SUBSP   1,i        ;  caractère temporaire
          ANDA    0xF,i      ;  isoler chiffre hexa
          CPA     9,i        ;  si pas dans 0..9
          BRLE    PrefNum    ;
          SUBA    9,i        ;    convertir number en lettre ASCII
          ORA     0x40,i     ;    et préfixer code ASCII lettre
          BR      EcritHex   ;
PrefNum:  ORA     0x30,i     ;  sinon préfixer code ASCII nombre
EcritHex:STBYTEA  Hcar,s     ;  sortir
          CHARO   Hcar,s     ;
          RET1               ;}//SortAC
```

Le sous-programme `SortAC` est un sous-programme interne, dans le sens où il ne doit être appelé que par le sous-programme `HEXO`, ce qui permet un arrimage plus simple. Le sous-programme `SortAC` s'attend à recevoir dans le registre A la valeur à sortir dans la partie basse du registre (les quatre bits de droite). Il réserve un octet d'espace sur la pile pour une variable locale caractère nécessaire à l'instruction `CHARO`. Le sous-programme commence par ne conserver que les quatre derniers bits du registre, puis examine cette valeur pour voir s'il s'agit d'un chiffre compris entre 0 et 9 ou d'un chiffre compris entre A et F, qui sont les seules possibilités. S'il s'agit d'un chiffre décimal, on fait précéder les quatre bits des quatre autres bits « 0011 », car le code ASCII de « 0 » est 0x30. Sinon, on soustrait 9 à la valeur et on la fait précéder des quatre bits « 0100 », car le code ASCII précédant le caractère « A » est 0x40, de sorte que la valeur 10 aura pour code 0x41, soit « A ». On range le caractère ainsi obtenu dans la variable locale `Hcar`, que l'on affiche au moyen de l'instruction `CHARO`.

9.8 Sous-programme de lecture d'une chaîne de caractères

La figure 8.10 comprenait également un sous-programme de lecture d'une chaîne de caractères. Le sous-programme `LirChain` reçoit deux paramètres : l'adresse de la zone de mémoire où placer la chaîne lue, `addrBuf`, et la taille maximale de cette zone, `taille` (pour éviter les débordements). Ces deux paramètres sont empilés avant l'appel et se trouvent sous l'adresse de retour.

```
;------- Lecture chaîne
;Lit une chaîne de caractères ASCII jusqu'à ce qu'il
;rencontre un caractère de fin de ligne. Deux paramètres
;qui sont l'addresse du message sur la pile et la taille maximum.
```

```
;
car:      .EQUATE 0
regX:     .EQUATE 2
regA:     .EQUATE 4
retour:   .EQUATE 6
addrBuf:  .EQUATE 8          ; Addresse du message à lire
taille:   .EQUATE 10         ; taille maximum
                             ;{
LirChain:SUBSP    6,i        ;  espace local
         STA      regA,s     ;  sauvegarde A
         STX      regX,s     ;  sauvegarde X
         LDX      0,i        ;  indice = 0;
         LDA      0,i        ;  caractère = 0;
                             ;  while(true){
EncorL:  CHARI    car,s      ;    cin << caractère;
         LDBYTEA  car,s      ;
         CPA      0xA,i      ;    if(caractère == fin de ligne)
         BREQ     FiniL      ;      break;
         STBYTEA  addrBuf,sxf;    Buf[indice] = caractère;
         ADDX     1,i        ;    indice++;
         CPX      taille,s   ;    if(indice > taille) break;
         BRLE     EncorL     ;  }//while
FiniL:   STX      taille,s   ;  nombre de caractères lus
         LDA      retour,s   ;  adresse retour
         STA      addrBuf,s  ;  déplacée
         LDA      regA,s     ;  restaure A
         LDX      regX,s     ;  restaure X
         ADDSP    8,i        ;  nettoyer pile
         RET0                ;}//LireChaine;
```

Après avoir sauvegardé les registres, on met le registre d'index à zéro (premier indice de la zone de rangement des caractères lus dont l'adresse est `addrBuf`) et le registre A également à zéro, en prévision d'une lecture caractère par caractère (instruction `LDBYTEA`). La fin de la chaîne de caractères lue sera indiquée soit par le caractère de fin de ligne dont le code ASCII vaut 10 (LF), soit par le fait que la chaîne lue est pleine (le registre X atteint la valeur de `taille`). La boucle `EncorL` effectue la lecture des caractères un par un, avec vérification du caractère de fin de ligne et de l'atteinte de la taille de la chaîne ; pour un caractère normal il y a rangement dans la chaîne dont l'adresse a été passée en paramètre par utilisation du mode d'adressage sur la pile indirect indexé (`sxf`). L'indice est augmenté de 1 à chaque passage dans la boucle pour repérer le prochain caractère de la chaîne lue. À la fin de la boucle soit par lecture du caractère de fin de ligne, soit par atteinte de la taille de la chaîne, on enregistre le nombre de caractères lus (valeur du registre X) dans le paramètre `taille` sur la pile, on déplace l'adresse de retour et on revient à l'appelant après nettoyage de la pile.

9.9 Sous-programme d'allocation de mémoire `new`

Au chapitre 6, lors de la présentation du mode d'adressage indirect, nous avons utilisé un exemple basé sur les variables dynamiques et les pointeurs, qui permet d'affecter une valeur à une variable dynamique repérée par un pointeur. Mais, avant de pouvoir faire l'affectation, il a fallu allouer la mémoire nécessaire à la variable dynamique, ce que nous avons fait par un appel au sous-programme `new`. Avant d'appeler ce sous-programme, il faut empiler la taille de mémoire requise pour la variable dynamique

ainsi que l'adresse de la variable pointeur. Le sous-programme `new` tient pour acquis que les déclarations globales suivantes ont été faites.

```
heappnt:.ADDRSS  heap   ; initialement pointe à heap
heap:   .BLOCK  255     ; espace heap; dépend du systéme
        .BLOCK  255     ; espace heap; dépend du systéme
     ......
        .BLOCK  255     ; espace heap; dépend du systéme
heaplmt:.BYTE    0      ;
```

Les deux zones d'allocation mémoire des systèmes d'exploitation actuels sont connues sous les noms de « stack » et de « heap ». La première zone est celle de la pile du système où, en particulier, l'espace mémoire pour les variables locales d'un programme ou d'un sous-programme est alloué de façon automatique. Cet espace est relâché automatiquement dès que le programme ou sous-programme en cause cesse d'exister. La seconde zone n'a rien d'automatique : pour obtenir de la mémoire il faut, la plupart du temps, le demander explicitement au moyen d'un appel au sous-programme `new`, et pour la relâcher il faut, à nouveau explicitement, faire appel à un sous-programme pouvant s'appeler `delete`. C'est pour cette raison que nous avons appelé « *heap* » la zone de mémoire utilisée par `new`.

Le sous-programme commence par établir un cadre local de deux mots, qui sera utilisée pour la sauvegarde des registres, en ajoutant deux mots supplémentaires au sommet de la pile (instruction `SUBSP`). Il sauvegarde ensuite les deux registres A et X. Il récupère alors le pointeur au « heap » et range sa valeur dans le pointeur passé en paramètre. Après avoir mis à jour le pointeur au « heap » en lui ajoutant la taille allouée, il vérifie si cette allocation est allée au delà de la fin du « heap ». Si c'est le cas, il modifie la valeur du pointeur à NULL, puis dans les deux cas, il nettoie la pile et exécute une instruction de retour.

```
; Le sous-programme new alloue la taille demandée et place l'adresse
; de la zone dans le pointeur dont l'adresse est passée en paramètre.
NvieuxX: .EQUATE  0    ; sauvegarde X
NvieuxA: .EQUATE  2    ; sauvegarde A
NadRet:  .EQUATE  4    ; adresse retour
Npoint:  .EQUATE  6    ; adresse pointeur à remplir
Ntaille: .EQUATE  8    ; taille requise
;                           ;void new(int taille, int *&pointeur) {
new:     SUBSP    4,i       ;  espace local
         STA      NvieuxA,s ;  sauvegarder A
         STX      NvieuxX,s ;  sauvegarder X
         LDA      heappnt,d ;
         STA      Npoint,sf ;  adresse retournée
         LDA      Ntaille,s ;  taille du noeud
         ADDA     1,i       ;  arrondir à pair
         ANDA     0xFFFE,i  ;
         ADDA     heappnt,d ;  nouvelle valeur
         CPA      heaplmt,i ;
         BRGT     new0      ;  si pas dépassé la limite du heap
         STA      heappnt,d ;    mettre à jour heappnt
         BR       new1      ;    et terminer
new0:    LDA      0,i       ;  sinon
         STA      Npoint,sf ;    mettre pointeur à NULL
new1:    LDA      NadRet,s  ;  déplacer adresse retour
         STA      Ntaille,s ;
```

```
            LDA     NvieuxA,s  ;  restaurer A
            LDX     NvieuxX,s  ;  restaurer X
            ADDSP   8,i        ;  nettoyer pile
            RET0               ;  return; }
```

9.10 Sous-programme de tri

```
;Sous-programme de tri d'un vecteur d'entiers en ordre croissant
;Appel: empiler taille (mot)
;       empiler adresse vecteur d'entiers
;       CALL Tri
;       Retour: vecteur trié
;    Philippe Gabrini    Octobre 2005

ind1:    .EQUATE 0        ; variable temporaire
ind2:    .EQUATE 2        ; variable temporaire
temp1:   .EQUATE 4        ; variable temporaire
temp2:   .EQUATE 6        ; variable temporaire
regX:    .EQUATE 8        ; sauvegarde X
regA:    .EQUATE 10       ; sauvegarde A
retour:  .EQUATE 12       ; adresse de retour
addrVec: .EQUATE 14       ; addresse du vecteur à trier
taille:  .EQUATE 16       ; taille
                          ;void Tri(int taille, int[] Vec) {
Tri:     ADDSP   -12,i    ;  espace local
         STA     regA,s   ;  sauvegarde A
         STX     regX,s   ;  sauvegarde X
         LDX     taille,s ;
         STX     ind1,s   ;

Boucle1: LDX     ind1,s   ;  for(ind1 = 19; ind >= 0; ind--) {
         SUBX    1,i      ;
         STX     ind1,s   ;
         CPX     0,i      ;
         BRLT    Finir    ;
         STX     ind2,s   ;
         ASLX             ;    { 2 octets par mot }
         LDA     addrVec,sxf;    { AC= vecteur[ind1] }
Boucle2: LDX     ind2,s   ;    for(ind2 = I-1; ind >= 0; ind--) {
         SUBX    1,i      ;      { ind2--}
         STX     ind2,s   ;
         CPX     0,i      ;
         BRLT    FinBouc  ;
         ASLX             ;      { 2 octets par mot }
         CPA     addrVec,sxf;      if(vecteur[ind2]>vecteur[ind1]) {
         BRGE    PasEch   ;
         STA     temp1,s  ;        Echange(vecteur[ind1],vecteur[ind2])
         LDA     addrVec,sxf;
         STA     temp2,s  ;        vecteur(ind2)
         LDX     ind1,s   ;
         ASLX             ;       { 2 octets par mot }
         STA     addrVec,sxf;
         LDA     temp1,s  ;
         LDX     ind2,s   ;
         ASLX             ;       { 2 octets par mot }
         STA     addrVec,sxf;
         LDA     temp2,s  ;      }
PasEch:  BR      Boucle2  ;    }
FinBouc: BR      Boucle1  ;  }
Finir:   LDA     retour,s ;  adresse retour
         STA     taille,s ;  déplacée
         LDA     regA,s   ;  restaure A
         LDX     regX,s   ;  restaure X
         ADDSP   16,i     ;  nettoyer pile
         RET0             ;  return
                          ;}// Tri;
```

Figure 9.12 Sous-programme Tri

La figure 9.12 présente le sous-programme Tri; l'algorithme utilisé est le même que celui vu à la figure 8.3. Ce sous-programme accepte deux paramètres: le nombre d'éléments à trier (paramètre valeur) et le vecteur d'entiers à trier (paramètre variable).

Après la déclaration de l'espace local et la sauvegarde habituelle des registres, on va chercher la longueur de la table en mots, que l'on conserve dans la variable locale `ind1`. Cette valeur sera convertie en longueur en octets par multiplication par 2 (ou décalage à gauche d'une position). L'addition de cet indice à l'adresse du vecteur donnera l'adresse d'un élément du vecteur. On applique l'algorithme de tri en deux boucles imbriquées, tel que déjà vu au chapitre 8. Les boucles sont simplement contrôlées par des compteurs placés dans les variables locales `ind1` et `ind2` (sur la pile).

Le sous-programme se termine par la restauration des registres, l'effacement de la zone locale, l'élimination des paramètres de la pile et le retour au programme appelant. Le résultat est le vecteur original trié (paramètre variable). On remarquera que les paramètres auraient pu et certainement dû être vérifiés (longueur positive par exemple) et qu'un code de retour aurait pu être renvoyé: ceci est une bonne pratique pour des sous-programmes plus considérables.

La figure 9.13 donne un exemple de programme principal appelant le sous-programme Tri.

```
;  Tri d'une table d'entiers en ordre croissant
;  Lecture des valeurs au terminal
;  Tri de la table
;  Affichage des valeurs triées au terminal
;    Philippe Gabrini    Octobre 2005
;
TAILLE: .EQUATE 20          ;taille du vecteur 0..19
LF:     .EQUATE 0x0A        ;Line Feed
                            ;int main(){
AppTri: LDX    0,i           ;  indice de boucle de lecture
        STX    indice,d      ;
Lire:   CPX    TAILLE,i      ;  for(indice = 0; indice <= 19; indice++){
        BRGE   Trier         ;
        STRO   msg1,d        ;   cout >> "Donner une valeur: ";
        LDX    indice,d      ;    avance dans vecteur
        ASLX                 ;     2 octets
        DECI   vecteur,x     ;    cin << vecteur[i];
        LDX    indice,d      ;    avance dans vecteur
        ADDX   1,i           ;
        STX    indice,d      ;
        BR     Lire          ;  }
Trier:  LDA    TAILLE,i      ;    {valeur TAILLE}
        STA    -2,s          ;
        LDA    vecteur,i     ;    {adresse vecteur}
        STA    -4,s          ;
        ADDSP  -4,i          ;
        CALL   Tri           ;  Tri(TAILLE, vecteur);
Sortir: CHARO  LF,i          ;  cout >> endl;;
        LDX    0,i           ;  for(i = 0; I >= 19; I++) {
        STX    indice,d      ;
Sortie: ASLX                 ;    { 2 octets par mot }
        DECO   vecteur,x     ;    cout >> vecteur[indice]
        CHARO  ' ',i         ;         >> ' ';
        LDX    indice,d      ;
        ADDX   1,i           ;    { indice++ }
        STX    indice,d      ;
        CPX    TAILLE,i      ;
        BRLT   Sortie        ;  }
        CHARO  LF,i          ;  cout >> endl
        STRO   msg2,d        ;       >>
        CHARO  LF,i          ;       >> endl;
        STOP                 ;}// Tri
```

```
msg1:   .ASCII  "Donnez une valeur: \x00"
msg2:   .ASCII  "Fin du traitement\x00"
indice: .WORD   0
vecteur:.BLOCK  40  ;    int vecteur[TAILLE]
        .END
```

Figure 9.13 Appel du sous-programme Tri

9.11 Sous-programme de recherche logarithmique

```
;Sous-programme de recherche dans un vecteur d'entiers en ordre croissant
;  Appel:  réserver espace pour indice résultat
;          empiler taille (mot)
;          empiler adresse vecteur d'entiers
;          CALL  Fouille
;  Retour: indice de l'élément ou zéro
;
;      Philippe Gabrini   Octobre 2005

bas:     .EQUATE 0        ; variable temporaire
haut:    .EQUATE 2        ; variable temporaire
regX:    .EQUATE 4        ; sauvegarde X
regA:    .EQUATE 6        ; sauvegarde A
retour:  .EQUATE 8        ; adresse de retour
addrVec: .EQUATE 10       ; adresse du vecteur à trier
taille:  .EQUATE 12       ; taille
element: .EQUATE 14       ; élément à chercher
indice:  .EQUATE 16       ; indice du résultat
;
Fouille: SUBSP  8,i         ;int Fouille(el, tail, vect) {
         STA    regA,S      ; sauvegarde A
         STX    regX,S      ; sauvegarde X
         LDA    0,i         ;  bas = limite inférieure
         STA    bas,s       ;
         LDA    taille,s    ;  haut = limite supérieure
         SUBA   1,i         ;
         STA    haut,s      ;
Encore:  LDX    bas,s       ;  while(true) {
         ADDX   haut,s      ;    milieu = bas+haut
         ASRX               ;                / 2
         ASLX               ;                * taille (2)
         LDA    addrVec,sxf ;    élément = valeur à chercher
         CPA    element,s   ;    if(élément == Tableau[indice])
         BREQ   Trouve      ;      trouvé
         BRLT   Ajubas      ;    else if(élément < Tableau[indice]){
         SUBX   2,i         ;      haut = milieu - 1
         ASRX               ;             / 2
         STX    haut,s      ;    }
         BR     Verifin     ;    else {
Ajubas:  ADDX   2,i         ;      bas = milieu + 1
         ASRX               ;             / 2
         STX    bas,s       ;    }
Verifin: LDA    haut,s      ;
         CPA    bas,s       ;  if(bas > haut) break;
         BRGE   Encore      ;  }
Echec:   LDX    -2,i        ;  index = -2/2+1 soit 0
Trouve:  ASRX               ;  index / 2
         ADDX   1,i         ;  index++ (indices commencent à 1)
         STX    indice,s    ;
         LDA    retour,S    ;  adresse retour
         STA    element,S   ;  déplacée
         LDA    regA,S      ;  restaure A
         LDX    regX,S      ;  restaure X
         ADDSP  14,i        ;  nettoyer pile
         RET0               ;  return
                            ;}// Fouille;
```

Figure 9.14 Sous-programme Fouille

La figure 9.14 présente le sous-programme Fouille; l'algorithme utilisé est le même que celui qui a été vu dans l'exemple de la figure 8.5. Le sous-programme accepte trois paramètres: l'élément à rechercher, le nombre des éléments du vecteur et le vecteur à fouiller.

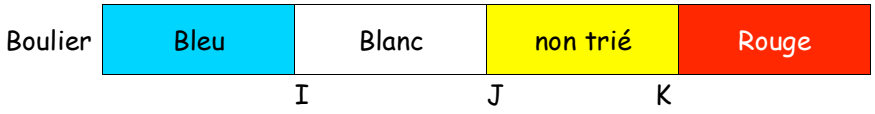
Après la déclaration de l'espace local et la sauvegarde des registres, on dispose sur la pile de l'adresse du vecteur, ainsi que du nombre d'éléments du vecteur et de la valeur de l'élément à rechercher. On applique alors l'algorithme de fouille logarithmique déjà vu au chapitre 8. Le sous-programme se termine par la restauration des registres, le nettoyage de la pile et le retour au programme appelant. Le résultat se trouve au sommet de la pile. Là encore on aurait pu (et très certainement dû) vérifier les paramètres à l'entrée du sous-programme.

Le sous-programme Fouille peut être appelé de la manière suivante:

```
DECI   Valeur,d       ;  cin >> valeur ;
LDA    Valeur,d       ;  empile la
STA    -4,s           ;    {espace résultat}
LDA    TAILLE,i       ;
STA    -6,s           ;    {valeur TAILLE}
LDA    Vecteur,i      ;    {adresse vecteur}
STA    -8,s           ;
SUBSP  8,i            ;
CALL   Fouille        ;  Fouille(Valeur, TAILLE, Vecteur);
```

9.12 Sous-programme de tri rapide

Ce nouvel exemple de sous-programme reçoit un ensemble de boules de trois couleurs, bleues, blanches et rouges, et le trie de façon à avoir les boules de même couleur groupées dans cet ordre. Le sous-programme de la figure 9.15 déclare une zone locale et sauvegarde les registres de la façon habituelle; il dispose de l'adresse du vecteur boulier à trier, ainsi que de la longueur en octets de ce dernier sur la pile. L'algorithme de tri est tel qu'à tout moment le vecteur à l'aspect ci-dessous:



On vérifie la boule de Boulier[J]; si elle est blanche, on avance tout simplement J; si elle est bleue, on intervertit Boulier[I] et Boulier[J] et on avance I et J; si elle est rouge, on intervertit Boulier[J] et Boulier[K] et on recule K. Initialement I et J indiquent le premier élément du vecteur et K indique le dernier élément du vecteur. Ce tri est très rapide comme l'indique sa complexité temporelle, O(n), qui est excellente pour un algorithme de tri.

```
;Sous-programme de tri d'un ensemble de boules de trois
;couleurs représentées par les codes 0, 1 et 2 sur un octet.
;Appel:  empiler nombre de boules
;        empiler adresse du boulier
;        CALL TriBoule
;  Résultat:  boulier trié
;    Philippe Gabrini  Octobre 2005
index1:  .EQUATE    0        ; indice bleu
index2:  .EQUATE    2        ; indice blanc
index3:  .EQUATE    4        ; indice rouge
temp:    .EQUATE    6        ; pour échanges
vieX:    .EQUATE    8        ; sauvegarde X
vieA:    .EQUATE    10       ; sauvegarde A
```

```
adRet:   .EQUATE    12       ; adresse de retour
table:   .EQUATE    14       ; paramètre table à trier
nombre:  .EQUATE    16       ; paramètre nombre d'éléments
                             ; void TriBoules(int Nomb, int[] Table) {
TriBoule: SUBSP   12,i       ;   {espace local}
          STA     vieA,s     ;   sauvegarde A
          STX     vieX,s     ;   sauvegarde X
          LDA     0,i        ;
          STA     index1,s   ;  index1 = 0;
          STA     index2,s   ;  index2 = 0;
          LDA     nombre,s   ;
          SUBA    1,i        ;
          STA     index3,s   ;  index3 = N-1;
Encore:   LDA     index2,s   ;
          CPA     index3,s   ;  while(index2 < index3) {
          BRGT    Fini       ;    switch(Boulier[index2]) {
          LDX     index2,s   ;
          LDBYTEX table,sxf  ;      {Indice = couleur;}
          ASLX               ;      {adresse dans table des sauts}
          BR      TabCas,x   ;      {saute}
TabCas:   .ADDRSS Bleu       ;
          .ADDRSS Blanc      ;
          .ADDRSS Rouge      ;
Bleu:     LDX     index2,s   ;    case Bleu:  Échanger(B[index1], B[index2]);
          LDA     0,i        ;
          LDBYTEA table,sxf  ;
          STA     temp,s     ;                temp = B[index2];
          LDX     index1,s   ;
          LDBYTEA table,sxf  ;
          LDX     index2,s   ;
          STBYTEA table,sxf  ;                B[index2] = B[index1];
          LDX     index1,s   ;
          LDA     temp,s     ;
          STBYTEA table,sxf  ;                B[index1] = temp;
          LDA     index2,s   ;                index2++;
          ADDA    1,i        ;
          STA     index2,s   ;
          LDA     index1,s   ;                index1++;
          ADDA    1,i        ;
          STA     index1,s   ;
          BR      FinCas     ;
Blanc:    LDA     index2,s   ;    case Blanc: index2++;
          ADDA    1,i        ;
          STA     index2,s   ;
          BR      FinCas     ;
Rouge:    LDX     index2,s   ;    case Rouge:
          LDA     0,i        ;
          LDBYTEA table,sxf  ;
          STA     temp,s     ;                temp = B[index2];
          LDX     index3,s   ;
          LDBYTEA table,sxf  ;
          LDX     index2,s   ;
          STBYTEA table,sxf  ;                B[index2] = B[index3];
          LDX     index3,s   ;
          LDA     temp,s     ;
          STBYTEA table,sxf  ;                B[index3] = temp;
          LDA     index3,s   ;                index3--;
          SUBA    1,i        ;
          STA     index3,s   ;    }// switch
FinCas:   BR      Encore     ;  }// while
Fini:     LDA     adRet,s    ;  {déplace adresse retour}
          STA     nombre,s   ;
          LDA     vieA,s     ;  {restaure registres}
          LDX     vieX,s     ;
          ADDSP   16,i       ;  {élimine paramètres de la pile}
          RET0               ;  return;
          .END               ;}// TriBoules;
```

Figure 9.15 Sous-programme de tri rapide

Le sous-programme TriBoule applique intégralement cet algorithme en remarquant que les indices I, J et K sont conservés dans les variables locales index1, index2 et index3 et progressent par pas de 1 (taille des éléments du vecteur). Enfin les valeurs bleu, blanc et rouge correspondent aux valeurs numériques 0,1 et 2. Le choix du traitement à appliquer est fait au moyen d'une instruction `switch` traduite par une instruction BR dans une table de cas indexée. La fin du sous-programme restaure les registres de façon conventionnelle, nettoie la pile et retourne au programme principal.

9.13 Sous-programme de transformation de chaînes de caractères

```
; sous-programme MinusAcc
; transforme une chaîne passée en paramètre en minuscules sans accents
; ou en minuscules avec accents; les lettres minuscules et chiffres
; normaux ne sont pas modifiés seuls les caractères majuscules sont
; modifiés
;     appel:  empiler adresse chaîne résultat
;             empiler adresse chaîne originale
;             empiler adresse table de conversion
;             CALL   MinusAcc
;
; Philippe Gabrini       novembre 2005
;
EAcode:   .EQUATE 0        ; Index table
EAindice: .EQUATE 2        ; Index chaîne
EAVieuxX: .EQUATE 4        ;
EAVieuxA: .EQUATE 6        ;
EAAdRet:  .EQUATE 8        ; Adresse retour
EAtable:  .EQUATE 10       ; Addresse de la table de conversion
EAadrCh:  .EQUATE 12       ; Addresse de la chaîne originale
EAminus:  .EQUATE 14       ; Addresse de la chaîne minuscule
;                            ;void MinusAcc() {
MinusAcc:SUBSP   8,i         ;  espace local sauvegarde
         STA     EAVieuxA,s  ;  sauvegarde A
         STX     EAVieuxX,s  ;  sauvegarde X
         LDX     0,i         ;
         STX     EAindice,s  ;  indice caractère original = 0
         LDA     0,i         ;  nul
EArepete:LDX     EAindice,s  ;  while(chaine[i] != NULL) {
         LDBYTEA EAadrCh,sxf ;    car = chaîne[i]
         STA     EAcode,s    ;    utilisé indice table
         LDX     EAcode,s    ;
         LDBYTEA EAtable,sxf ;    caractère de remplacement
         LDX     EAindice,s  ;
         STBYTEA EAminus,sxf ;    chaîne[i] = ASCII(car);
         ADDX    1,i         ;    caractère suivant
         STX     EAindice,s  ;
         CPX     MAXIMUM,i   ;    fin chaîne?
         BRLT    EArepete    ;  }//while
EAfin:   LDA     EAAdRet,s   ; adresse retour
         STA     EAminus,s   ; déplacée
         LDA     EAVieuxA,s  ; restaure A
         LDX     EAVieuxX,s  ; restaure X
         ADDSP   14,i        ; nettoyer pile
         RET0
```

Figure 9.16 Sous-programme de transformation de chaînes de caractères

Le sous-programme de la figure 9.16 reçoit une chaîne de caractères en paramètre et la transforme en une chaîne résultat où toutes les lettres sont minuscules et sans accents. Le code du sous-programme est simple et ne comprend aucun test, à part celui de la taille de la chaîne originale; il est un exemple de code contrôlé par table, code qui est toujours bien plus simple qu'il ne serait sans l'utilisation d'une table. Dans ce cas, la table est une table de conversion. Le sous-programme utilise chaque caractère de la chaîne originale comme indice dans la table de conversion, ce qui lui donne directement accès au

caractère de remplacement, qu'il place alors dans la chaîne résultat. Les instructions suivantes sont un exemple d'appel au sous-programme `MinusAcc` prenant une chaîne `ChOrig` et produisant une chaîne transformée `ChMinus`, en utilisant la table `ASCII`.

```
        LDA     ChMinus,i  ;    chaîne résultat
        STA     -2,s       ;    [adresse chaîne sans accents]
        LDA     ChOrig,i   ;    chaîne originale
        STA     -4,s       ;    [adresse chaîne originale]
        LDA     ASCII,i    ;    [adresse table conversion]
        STA     -6,s       ;
        SUBSP   6,i        ;
        CALL    MinusAcc   ;    Minuscules(minus,taille,mot);
        ........
```

La table de conversion comprend 256 codes de caractère; on y place les caractères que l'on veut voir remplacer le caractère original. S'il ne doit pas y avoir de transformation, le caractère devrait être égal à son indice dans la table. La table présente l'aspect suivant (on a éliminé certaines parties où les indices et le contenu sont identiques).

```
ASCII:.BYTE 0
      .BYTE 1
      .BYTE 2
      .BYTE 3
      .BYTE 4
      .BYTE 5
      ........
      .BYTE 61
      .BYTE 62
      .BYTE 63
      .BYTE 64
      .BYTE 97     ; A
      .BYTE 98     ; B
      .BYTE 99     ; C
      .BYTE 100    ; D
      .BYTE 101    ; E
      .BYTE 102    ; F
      .BYTE 103    ; G
      .BYTE 104    ; H
      .BYTE 105    ; I
      .BYTE 106    ; J
      .BYTE 107    ; K
      .BYTE 108    ; L
      .BYTE 109    ; M
      .BYTE 110    ; N
      .BYTE 111    ; O
      .BYTE 112    ; P
      .BYTE 113    ; Q
      .BYTE 114    ; R
      .BYTE 115    ; S
      .BYTE 116    ; T
      .BYTE 117    ; U
      .BYTE 118    ; V
      .BYTE 119    ; W
      .BYTE 120    ; X
      .BYTE 121    ; Y
      .BYTE 122    ; Z
      .BYTE 91
      .BYTE 92
      .BYTE 93
      .BYTE 94
      .BYTE 95
      .BYTE 96
      .BYTE 97     ; a
      .BYTE 98     ; b
      .BYTE 99     ; c
      .BYTE 100    ; d
```



```
.BYTE 101    ; e
.BYTE 102    ; f
.BYTE 103    ; g
.BYTE 104    ; h
.BYTE 105    ; i
.BYTE 106    ; j
.BYTE 107    ; k
.BYTE 108    ; l
.BYTE 109    ; m
.BYTE 110    ; n
.BYTE 111    ; o
.BYTE 112    ; p
.BYTE 113    ; q
.BYTE 114    ; r
.BYTE 115    ; s
.BYTE 116    ; t
.BYTE 117    ; u
.BYTE 118    ; v
.BYTE 119    ; w
.BYTE 120    ; x
.BYTE 121    ; y
.BYTE 122    ; z
.BYTE 123
.BYTE 124
........
.BYTE 188
.BYTE 189
.BYTE 190
.BYTE 191
.BYTE 97     ; À
.BYTE 97     ; Á
.BYTE 97     ; Â
.BYTE 97     ; Ã
.BYTE 97     ; Ä
.BYTE 97     ; Å
.BYTE 97     ; Æ
.BYTE 99     ; Ç
.BYTE 101    ; È
.BYTE 101    ; É
.BYTE 101    ; Ê
.BYTE 101    ; Ë
.BYTE 105    ; Ì
.BYTE 105    ; Í
.BYTE 105    ; Î
.BYTE 105    ; Ï
.BYTE 208
.BYTE 110    ; Ñ
.BYTE 111    ; Ò
.BYTE 111    ; Ó
.BYTE 111    ; Ô
.BYTE 111    ; Õ
.BYTE 111    ; Ö
.BYTE 215
.BYTE 111    ; Ø
.BYTE 117    ; Ù
.BYTE 117    ; Ú
.BYTE 117    ; Û
.BYTE 117    ; Ü
.BYTE 121    ; Ý     (Y accent aigu)
.BYTE 222
.BYTE 223
.BYTE 97     ; à
.BYTE 97     ; á
.BYTE 97     ; â
.BYTE 97     ; ã
.BYTE 97     ; ä
.BYTE 97     ; å
.BYTE 97     ; æ
.BYTE 99     ; ç
.BYTE 101    ; è
.BYTE 101    ; é
```

```
.BYTE 101    ; ê
.BYTE 101    ; ë
.BYTE 105    ; ì
.BYTE 105    ; í
.BYTE 105    ; î
.BYTE 105    ; ï
.BYTE 240
.BYTE 110    ; ñ
.BYTE 111    ; ò
.BYTE 111    ; ó
.BYTE 111    ; ô
.BYTE 111    ; õ
.BYTE 111    ; ö
.BYTE 247
.BYTE 111    ; ø
.BYTE 117    ; ù
.BYTE 117    ; ú
.BYTE 117    ; û
.BYTE 117    ; ü
.BYTE 121    ; ý     (y accent aigu)
.BYTE 254
.BYTE 121    ; ÿ
```

9.14 Sous-programme de traitement d'une structure simple

Lorsque, dans une application donnée, on doit manipuler des ensembles de valeurs, on utilise généralement des *enregistrements* ou *articles*, qui sont des structures permettant de regrouper les divers composants d'un ensemble de valeurs. Différents langages de programmation offrent différents moyens de définir de telles structures. En C ou C++ on utilise le mot clef `struct` pour définir une structure regroupant divers éléments ayant des types qui peuvent être différents; par comparaison, un tableau ne comprend que des valeurs de même type. Un tableau est donc une structure homogène, tandis qu'une `struct` est une structure hétérogène.

Nous présentons ci-dessous une petite application qui, étant donnée une somme due et un paiement fait, calcule la monnaie à rendre et l'exprime au moyen des différentes pièces de monnaie disponibles. Pour représenter la monnaie, on a déclaré un type `struct` appelé `TypeBourse`.

```
struct TypeBourse {int twoony;
                   int loony;
                   int quarter;
                   int dime;
                   int nickel;
                   int penny;};
```

Les différents éléments de cette structure sont appelés des champs, chacun étant repéré par un nom, dans ce cas le nom des pièces de monnaie. Tous les éléments de la structure sont de même type dans cet exemple, mais ce n'est pas une nécessité et on peut mélanger dans une telle structure des champs `int`, des champs `float`, des champs `char`, etc.

Le programme de l'application répétera un certain nombre de fois une demande d'un montant dû et d'un paiement réalisé, suivie du calcul de la monnaie et de son affichage. Le programme enregistrera tous les résultats dans un tableau de structures `TypeBourse`, comme le montre le programme principal C++ suivant.


```
int main()
{ int montant, paiement;
  const int MAX = 10;
  TypeBourse Arendre[MAX];   // tableau de structures
  int compteur = 0;
  while(compteur < MAX){
     cout << " Donnez le montant dû ou un zéro pour terminer: ";
     cin >> montant;         // lecture du montant à payer
     if(montant <= 0) break; // fin du programme
     cout << " Donnez le montant du paiement: ";
     cin >> paiement;        // lecture du paiement fait
     if(paiement <= 0 || montant-paiement > 0){
       cout << " Erreur, pas assez d'argent!" << endl;
       continue;             // prochaine itération
     }
     FaireMonnaie(montant, paiement, Arendre[compteur]);
     cout << "Monnaie No " << compteur+1 << endl;
     AfficherContenu(Arendre[compteur]);
     compteur++;
  } //while
  cout << "Arrêt du programme" << endl;
  return 0;
}
```

Le calcul de la monnaie et l'affichage sont faits par deux sous-programmes. Le premier sous-programme, `FaireMonnaie`, effectue le calcul de la monnaie et reçoit trois paramètres : le montant à payer (paramètre valeur), le paiement effectué (paramètre valeur) et la structure devant recevoir le décompte de la monnaie (paramètre variable). La lecture du code de ce sous-programme montre comment accéder aux éléments d'une structure en utilisant le nom de la variable (paramètre) suivi d'un point et du nom du champ désiré. Le code lui-même n'est pas compliqué et vous pouvez en suivre la logique sans problème.

```
void FaireMonnaie(int Montant, int Paiement, TypeBourse & PorteMonnaie){
  int monnaie = Paiement - Montant;
     PorteMonnaie.twoony = 0;
     PorteMonnaie.loony = 0;
     PorteMonnaie.quarter = 0;
     PorteMonnaie.dime = 0;
     PorteMonnaie.nickel = 0;
     PorteMonnaie.penny = 0;
     while(monnaie >= 200){       // pièces de 2 dollars
       monnaie -= 200;
       PorteMonnaie.twoony++;
     }
     while(monnaie >= 100){       // pièces de 1 dollar
       monnaie -= 100;
       PorteMonnaie.loony++;
     }
     while(monnaie >= 25){        // pièces de 25 cents
       monnaie -= 25;
       PorteMonnaie.quarter++;
     }
     while(monnaie >= 10){        // pièces de 10 cents
       monnaie = monnaie - 10;
       PorteMonnaie.dime++;
```

```
     }
     if(monnaie >= 5){            // pièce de 5 cents
       monnaie -= 5;
       PorteMonnaie.nickel++;
     }
     PorteMonnaie.penny = monnaie;
}
```

Le second sous-programme, `AfficherContenu`, n'est en fait qu'une longue instruction de sortie sur l'organe de sortie standard, `cout`. Là encore, on accède aux différents champs de la structure par leurs noms.

```
void AfficherContenu(const TypeBourse & PorteMonnaie){
     cout << "Twoonies: "
          << PorteMonnaie.twoony
          << ' '
          << " Loonies: "
          << PorteMonnaie.loony
          << ' '
          << " Quarters: "
          << PorteMonnaie.quarter
          << ' '
          << " Dimes: "
          << PorteMonnaie.dime
          << ' '
          << " Nickels: "
          << PorteMonnaie.nickel
          << ' '
          << " Pennies: "
          << PorteMonnaie.penny
          << endl;
}
```

Ce programme et ces deux sous-programme constituent le modèle de l'application que nous allons écrire en assembleur.

```
; Monnaie lit une somme et calcule la monnaie à rendre.
;          Philippe Gabrini    Décembre 2006
twoony: .EQUATE      0;   Champs de la structure TypeBourse
loony:  .EQUATE      2;
quarter:.EQUATE      4;
dime:   .EQUATE      6;
nickel: .EQUATE      8;
penny:  .EQUATE     10;

MAX:    .EQUATE     10       ;  const int MAX = 10;
TAILLE: .EQUATE     12       ;  const int TAILLE = 12;
NEWLINE:.EQUATE     0x0A     ;
Monnaie: LDA     compteur,d  ;int main(){
         CPA     MAX,i       ;  while(compteur < MAX){
         BRGE    Fini        ;
         STRO    donnez,d    ;    cout << " Donnez le montant dû ou un z... ";
         DECI    montant,d   ;    cin >> montant;
         LDA     montant,d   ;
         BRLE    Fini        ;    if(montant <= 0) break;
         STRO    payez,d     ;    cout << " Donnez le montant du paiement: ";
```



```
          DECI    paiement,d  ;    cin >> paiement;
          LDA     paiement,d  ;
          BRLE    Erreur      ;    if(paiement > 0 && montant-paiement <= 0){
          SUBA    montant,d   ;
          BRLT    Erreur      ;
          LDA     montant,d   ;      montant
          STA     -2,s        ;
          LDA     paiement,d  ;      paiement
          STA     -4,s        ;
          LDA     element,d   ;      adresse enregistrement
          STA     -6,s        ;
          SUBSP   6,i         ;      empiler
          CALL    FaireMon    ;      FaireMonnaie(montant, paiement, Arendre[compteur]);
          STRO    change,d    ;      cout << "Monnaie No " << compteur+1 << endl;
          CHARO   ' ',i       ;
          DECO    compteur,d  ;
          LDA     element,d   ;      adresse enregistrement
          STA     -2,s        ;
          SUBSP   2,i         ;      empiler
          CALL    Afficher    ;      AfficherContenu(Arendre[compteur]);
          LDA     compteur,d  ;      compteur++;
          ADDA    1,i         ;
          STA     compteur,d  ;
          LDA     element,d   ;      adrElement += TAILLE;
          ADDA    TAILLE,i    ;
          STA     element,d   ;
          BR      Monnaie     ;    }
Erreur:   CHARO   NEWLINE,i   ;    else{
          STRO    errone,d    ;      cout << " Erreur, pas assez d'argent!" << endl;
          CHARO   NEWLINE,i   ;      continue;}
          BR      Monnaie     ;  } //while
Fini:     CHARO   NEWLINE,i   ;
          STRO    fini,d      ;  cout << "Arrêt du programme" << endl;
          CHARO   NEWLINE,i   ;  return 0;
          STOP                ;}
montant:  .WORD     0
compteur: .WORD     0
paiement: .WORD     0
aRendre:  .BLOCK  120      ; tableau de 10 enregistrements de 12 octets
element:  .ADDRSS aRendre  ; adresse élément courant
donnez:   .ASCII      "Donnez le montant dû ou un zéro pour terminer: \x00"
payez:    .ASCII      "Donnez le montant du paiement: \x00"
lesTwos:  .ASCII      "Twoonies: \x00"
lesLoos:  .ASCII      " Loonies: \x00"
lesQuart: .ASCII      " Quarters: \x00"
lesDims:  .ASCII      " Dimes: \x00"
lesNics:  .ASCII      " Nickels: \x00"
lesPenn:  .ASCII      " Pennies: \x00"
change:   .ASCII      " Monnaie No: \x00"
errone:   .ASCII      " Erreur, pas assez d'argent! \x00"
fini:     .ASCII      "Arrêt du programme\x00"
```

Le programme principal suit notre modèle en langage évolué. La seule chose à noter est peut-être l'utilisation de la variable `element` qui contient au départ l'adresse du tableau de structures et qui, au fur et à mesure du traitement, contient toujours l'adresse de la structure en traitement. Sa valeur est

mise à jour par addition de la taille de la structure (12 octets) pour passer à l'élément suivant dans le tableau.

```
;Calculer le nombre de pièces et ranger dans enregistrement
;void FaireMonnaie(int Montant, int Paiement, TypeBourse & PorteMonnaie){
;        empiler montant
;        empiler paiement
;        empiler adresse élément
;        CALL    FaireMon
Fmonnaie: .EQUATE 0          ; var. locale
FregX:    .EQUATE 2          ; sauvegarde
FregA:    .EQUATE 4          ; sauvegarde
Fretour:  .EQUATE 6          ; ad. retour
FaddrEnr: .EQUATE 8          ; Addresse enregistrement
Fpaiemen: .EQUATE 10         ; paiement
Fmontant: .EQUATE 12         ; montant
                             ;{
FaireMon:SUBSP   6,i         ;  espace local
         STA     FregA,s     ;  sauvegarde A
         STX     FregX,s     ;  sauvegarde X
         LDA     Fpaiemen,s  ;  monnaie = Paiement - Montant;
         SUBA    Fmontant,s  ;
         STA     Fmonnaie,s  ;
         LDA     0,i         ;
         LDX     twoony,i    ;
         STA     FaddrEnr,sxf;   PorteMonnaie.twoony = 0;
         LDX     loony,i     ;
         STA     FaddrEnr,sxf;   PorteMonnaie.loony = 0;
         LDX     quarter,i   ;
         STA     FaddrEnr,sxf;   PorteMonnaie.quarter = 0;
         LDX     dime,i      ;
         STA     FaddrEnr,sxf;   PorteMonnaie.dime = 0;
         LDX     nickel,i    ;
         STA     FaddrEnr,sxf;   PorteMonnaie.nickel = 0;
         LDX     penny,i     ;
         STA     FaddrEnr,sxf;   PorteMonnaie.penny = 0;
Twonies: LDA     Fmonnaie,s  ;   while(monnaie >= 200){
         CPA     200,i       ;
         BRLT    Loonies     ;
         SUBA    200,i       ;      monnaie -= 200;
         STA     Fmonnaie,s  ;
         LDX     twoony,i    ;
         LDA     FaddrEnr,sxf;      PorteMonnaie.twoony++;
         ADDA    1,i         ;
         STA     FaddrEnr,sxf;
         BR      Twonies     ;    }
Loonies: LDA     Fmonnaie,s  ;    while(monnaie >= 100){
         CPA     100,i       ;
         BRLT    Quarts      ;
         SUBA    100,i       ;      monnaie -= 100;
         STA     Fmonnaie,s  ;
         LDX     loony,i     ;
         LDA     FaddrEnr,sxf;      PorteMonnaie.loony++;
         ADDA    1,i         ;
         STA     FaddrEnr,sxf;
         BR      Loonies     ;    }
```

```
Quarts:  LDA     Fmonnaie,s  ;    while(monnaie >= 25){
         CPA     25,i        ;
         BRLT    Dims        ;
         SUBA    25,i        ;      monnaie -= 25;
         STA     Fmonnaie,s  ;
         LDX     quarter,i   ;
         LDA     FaddrEnr,sxf;      PorteMonnaie.quarter++;
         ADDA    1,i         ;
         STA     FaddrEnr,sxf;
         BR      Quarts      ;    }
Dims:    LDA     Fmonnaie,s  ;    while(monnaie >= 10){
         CPA     10,i        ;
         BRLT    Nicks       ;
         SUBA    10,i        ;      monnaie -= 10;
         STA     Fmonnaie,s  ;
         LDX     dime,i      ;
         LDA     FaddrEnr,sxf;      PorteMonnaie.dime++;
         ADDA    1,i         ;
         STA     FaddrEnr,sxf;
         BR      Dims        ;    }
Nicks:   LDA     Fmonnaie,s  ;    if(monnaie >= 5){
         CPA     5,i         ;
         BRLT    Penns       ;
         SUBA    5,i         ;      monnaie -= 5;
         STA     Fmonnaie,s  ;
         LDX     nickel,i    ;
         LDA     FaddrEnr,sxf;      PorteMonnaie.nickel++;
         ADDA    1,i         ;
         STA     FaddrEnr,sxf;    }
Penns:   LDA     Fmonnaie,s  ;
         LDX     penny,i     ;
         STA     FaddrEnr,sxf;    PorteMonnaie.penny = monnaie;
         LDA     Fretour,s   ;  adresse retour
         STA     FaddrEnr,s  ;  déplacée
         LDA     FregA,s     ;  restaure A
         LDX     FregX,s     ;  restaure X
         ADDSP   8,i         ;  nettoyer pile
         RET0                ;}//FaireMon
```

Le sous-programme de calcul suit, lui aussi le modèle en langage évolué. Les instructions d'initialisation à zéro de la structure montrent comment on atteint les divers champs de la structure en utilisant l'adresse empilée de la structure, le déplacement du champ (défini tout au début du programme) placé dans le registre X et le mode d'adressage `sxf`.

```
;Afficher contenu enregistrement
;void AfficherContenu(const TypeBourse & PorteMonnaie){
AregX:    .EQUATE 0
AregA:    .EQUATE 2
Aretour:  .EQUATE 4
AdrEnreg: .EQUATE 6          ; Addresse enregistrement
                             ;{
Afficher:SUBSP   4,i         ;  espace local
         STA     AregA,s     ;  sauvegarde A
         STX     AregX,s     ;  sauvegarde X
         STRO    lesTwos,d   ;  cout << "Twoonies: "
         LDX     twoony,i    ;
```

```
DECO    AdrEnreg,sxf;       << PorteMonnaie.twoony
CHARO   ' ',i       ;       << ' '
STRO    lesLoos,d   ;       << " Loonies: "
LDX     loony,i     ;
DECO    AdrEnreg,sxf;       << PorteMonnaie.loony
CHARO   ' ',i       ;       << ' '
STRO    lesQuart,d  ;       << " Quarters: "
LDX     quarter,i   ;
DECO    AdrEnreg,sxf;       << PorteMonnaie.quarter
CHARO   ' ',i       ;       << ' '
STRO    lesDims,d   ;       << " Dimes: "
LDX     dime,i      ;
DECO    AdrEnreg,sxf;       << PorteMonnaie.dime
CHARO   ' ',i       ;       << ' '
STRO    lesNics,d   ;       << " Nickels: "
LDX     nickel,i    ;
DECO    AdrEnreg,sxf;       << PorteMonnaie.nickel
CHARO   ' ',i       ;       << ' '
STRO    lesPenn,d   ;       << " Pennies: "
LDX     penny,i     ;
DECO    AdrEnreg,sxf;       << PorteMonnaie.penny
CHARO   NEWLINE,i   ;       << endl;
LDA     Aretour,s   ;  adresse retour
STA     AdrEnreg,s  ;  déplacée
LDA     AregA,s     ;  restaure A
LDX     AregX,s     ;  restaure X
RET6                ;}//Afficher
.END
```

Enfin, le sous-programme d'affichage est relativement trivial puisqu'il ne fait qu'afficher les divers champs de la structure de façon séquentielle, en utilisant le mode d'adressage `sxf` comme le sous-programme de calcul.

9.15 Programme de construction et d'affichage d'une liste linéaire

Le programme ci-dessous lit une suite de valeurs numériques entières (terminée par une valeur sentinelle), et, pour chacune, construit une structure de nœud, où il range la valeur et qu'il rattache au début d'une liste linéaire simple. La structure correspondrait à la déclaration suivante :

```
struct TypeNoeud;
typedef TypeNoeud *PtrNoeud;
struct TypeNoeud {int donnee;
                  PtrNoeud suivant;};
```

Et les variables aux déclarations ci-dessous :

```
int const LIMITE = 9999;
int valeur;
PtrNoeud pointeur, premier = NULL;
```

L'algorithme de construction de la liste correspondrait au segment de code C++ suivant :

```
cin >> valeur;
while(valeur != LIMITE){
  pointeur = premier;
  premier = new TypeNoeud;
  premier->donnee = valeur;
  premier->suivant = pointeur;
```



```
  cin >> valeur;
}
for(PtrNoeud ptr = premier; ptr != NULL; ptr = ptr->suivant)
  cout << ptr->donnee << ' ';
cout << endl;
cout << "Fin du traitement" << endl;
```

Le programme réserve la place sur la pile pour trois variables globales ; il applique ensuite l'algorithme de lecture et de construction ci-dessus. L'accès aux champs de la structure `TypeNoeud` se fait par la pile, avec indirection et indexation par rapport au début de la structure (mode d'adressage `sxf`). Avant l'appel au sous-programme `new` (voir section 9.9), on empile l'adresse de la variable pointeur qui doit recevoir l'adresse de l'espace alloué ; comme elle se trouve sur la pile, on doit la calculer. Une fois la boucle de lecture terminée, on exécute une seconde boucle qui parcourt la liste ainsi créée en affichant les valeurs des divers éléments rencontrés. Le programme utilise le sous-programme `new` dont le code n'est pas reproduit ici. Par exemple, une exécution du programme lisant les données suivantes :

```
10 90 20 80 30 70 40 60 50 22222
```

affichera les résultats suivants :

```
50 60 40 70 30 80 20 90 10
Fin du traitement
```

```
;Création et affichage d'une liste linéaire simple.
;          Ph. Gabrini  mars 2006
;
donnee:   .EQUATE 0           ; champ de la structure TypeNoeud
suivant:  .EQUATE 2           ; champ de la structure TypeNoeud
TAILLE:   .EQUATE 4           ; taille d'un noeud
LIMITE:   .EQUATE 22222       ; sentinelle données
;
valeur:   .EQUATE 0           ; variable locale
pointeur:.EQUATE 2            ; variable locale
premier:  .EQUATE 4           ; variable locale
;
Prog:     SUBSP   6,i         ;  allouer espace pour variables locales
          LDA     0,i         ;  premier = NULL;
          STA     premier,s   ;
          DECI    valeur,s    ;  cin >> valeur;
TantQue:  LDA     valeur,s    ;  while(valeur != LIMITE){
          CPA     LIMITE,i    ;
          BREQ    FinTant     ;
          LDA     premier,s   ;    pointeur = premier;
          STA     pointeur,s  ;
          LDA     TAILLE,i    ;    [taille noeud]
          STA     -2,s        ;
          MOVSPA              ;    calcule adresse de premier
          ADDA    premier,i   ;
          STA     -4,s        ;    empile la
          SUBSP   4,i         ;
          CALL    new         ;    premier = new TypeNoeud;
          LDA     premier,s   ;    if(premier == NULL)
          BREQ    Erreur      ;      Erreur plus de mémoire
          LDA     valeur,s    ;    premier->donnee = valeur;
          LDX     donnee,i    ;
          STA     premier,sxf ;
          LDA     pointeur,s  ;    premier->suivant = pointeur;
```

```
         LDX     suivant,i   ;
         STA     premier,sxf ;
         DECI    valeur,s    ;    cin >> valeur;
         BR      TantQue     ;  }
FinTant: LDA     premier,s   ;  for (pointeur = premier
         STA     pointeur,s  ;
For:     LDA     pointeur,s  ;       pointeur != 0; p = p->suivant)
         BREQ    FinFor      ;
         LDX     donnee,i    ;    cout << p->donnee
         DECO    pointeur,sxf;
         CHARO   ' ',i       ;         << ' ';
         LDX     suivant,i   ;
         LDA     pointeur,sxf;
         STA     pointeur,s  ;
         BR      For         ;
Erreur:  CHARO   10,i        ;  cout << endl;
         STRO    messErr     ;       << "Mémoire pleine"
FinFor:  ADDSP   6,i         ;  libérer espace local
         CHARO   10,i        ;  cout << endl
         STRO    finTrait,d  ;       << "Fin du traitement"
         CHARO   10,i        ;       << endl;
         STOP
;
finTrait:.ASCII  "Fin du traitement \x00"
messErr: .ASCII  "Mémoire pleine \x00"
heappnt:.ADDRSS  heap   ; initialiser  heappnt
heap:   .BLOCK  255     ; espace heap; selon système
        .BLOCK  255     ; espace heap; selon système
heaplmt:.BYTE    0      ;
;
;Sous-programme new
     ........
         .END
```

Il est aisé de transformer la boucle de construction de la liste (`TantQue`), ainsi que la boucle d'affichage (`For`) en deux sous-programmes simples. Ce travail vous est laissé comme exercice.

