Chaînes de caractères

Le type chaîne de caractères

Les chaînes de caractères (string) est un type abstrait qui est très utilisé en l'Informatique. Chaque chaîne est une suite de caractères est les fonctions de base sont :

- création d'une chaîne;
- insertion des caractères;
- suppression des caractères;
- copie;
- concaténation de deux chaînes;
- longueur de la chaîne;
- opérations d'entrée/sortie.

Certains langages (SNOBOL, Awk, Perl) était créés exclusivement pour manipulations de strings. Autres langages (Basic, Turbo Pascal) disposent d'un véritable type `string. Les variables de ce type ont comme des valeurs de suites de caractères qui peuvent évoluer, à la fois en contenu et en longueur, au fil du déroulement du programme. En Pascal standard, Fortran et C il n'existe un type spécial et on doit stocker les chaînes en tableaux et manipules leur contenue caractère par caractère.

Présentation des chaînes en C

En C la chaîne est présentée par un tableau unidimensionnel de caractères mais il y a une convention qui permet la représentation et le traitement des strings – la fin de la chaîne actuelle est dénotée par '\0' (le caractère avec code zéro) (Fig.8.1).

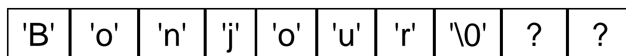


Fig. 8.1

Déclaration

La convention permet de définir des chaînes constantes :

```
#define salutation  "Bonjour"
```

On déclare une variable comme un tableau de char, dont la taille est égale à la longueur maximale de la chaîne plus un. C'est pour que on puisse stocker le caractère '\0' dénotant la fin.

```
char chaine[10]; // la longueur maximale = 9
```

Initialisation

La chaîne peut être initialisée un tableau ordinaire (ex.1 et ex.2), mais dans ce cas on ne doit pas oublier le caractère '\0'. Elle peut être initialisée par une constante littérale et dans ce cas le compilateur va stocker le zéro s'il y a place. On peut initialiser un pointeur avec un littérale mais dans ce cas le pointeur va designer la constante et on ne peut plus changer les caractères (fig. 8.2).

```
char machaine1[]= {'B','o','n','j','o','u','r','\0'}; // ex.1
char machaine2[10]= {'B','o','n','j','o','u','r','\0'}; // ex.2
char machaine3[]= "Bonjour"; // ex.3
char machaine4[8]= "Bonjour"; // ex.4
char *p = "Bonjour";//  Attention c'est constante
```

```
char a[]= "Bonjour";
char *p= "Bonjour";
```

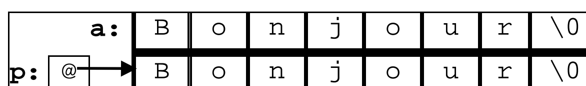


Fig. 8.2

On doit faire attention d'avoir assez de place pour tous les caractères y compris le '\0' final. Dans le premier exemple ci-dessous le compilateur va stocker la chaîne mais sans le zéro final et quand on utilise cette variable au cours de déroulement du programme il y aura une erreur d'exécution car on ne peut pas trouver la fin du string. Dans le deuxième exemple la place n'est pas suffisante et il y aura une erreur de compilation.

```
char ch1[7]= "Bonjour";// erreur d'exécution
char ch2[6]= "Bonjour"; //erreur de compilation
```

Accès et affectation

Accès aux caractères individuels

La chaîne est un tableau dont les éléments sont des caractères, donc l'accès aux caractères individuels est comme aux éléments du tableau (fig. 8.3)

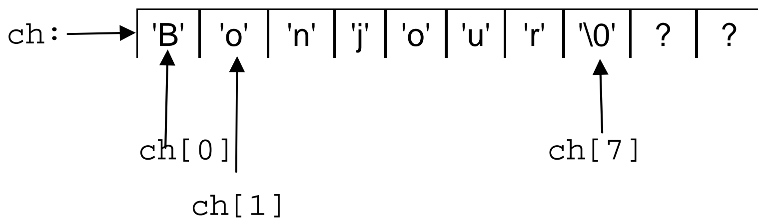


Fig. 8.3

Affectation et initialisation

Quand on initialise une chaîne déclarée comme un tableau les caractères sont stockés dans le tableau n'import quel est le moyen – des caractères individuels ou une chaîne constante.

```
char machaine[10]= {'B','o','n','j','o','u','r','\0'};
char machaine[10]= "Bonjour";
```

Quand un pointeur est déclaré et initialisé avec une chaîne constante (littéral), qui est le seul moyen, sa valeur devient l'adresse du premier caractère de la chaîne. Dans ce cas les caractères ne sont pas copiés.

```
char *p = "Bonjour";
```

On ne peut pas affecter directement une chaîne à une autre chaîne parce que le nom de la chaîne est le nom d'un pointeur constant et l'affectation essaie de changer sa valeur (la valeur du premier caractère) avec l'adresse du premier caractère de première chaîne. Les caractères ne sont pas copiés

```
char machaine[10];
machaine = "Bonjour";// erreur - machaine est un pointeur constant
```

L'affectation à un pointeur est valide est dans ce cas le pointeur va designer la nouvelle chaîne. Mais seulement l'adresse est copiée, et non les caractères.

```
char p;
p = "Bonjour";//c'est bon
```

Pour copier les caractères c'est-à-dire pour créer une copie d'une chaîne on doit copier caractère par caractère (fig. 8.4) dans une boucle. Ci-dessous son montrés deux exemples d'une fonction pour la copie des chaînes.

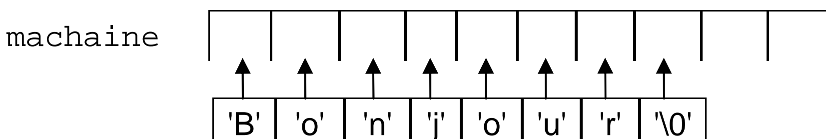


Fig. 8.4

```
void chainecop(char *a, const char *de){
   int i;
   for (i=0; de[i] !='\0';i++) a[i]=de[i];
   a[i]='\0';
}
```

```
void chainecop(char *a, const char *de){
   while (*de  !='\0') *a++ = *de++;
   *a ='\0';
}
```

Entrée-sortie des chaînes

On peut utiliser les fonctions *scanf* et *printf* avec La spécification "%s" mais il y a quelques fonctions spéciales pour les strings.

La fonction gets(chaîne)

Lit des caractères tapés sur clavier jusqu'à la fin de la ligne la fin de fichier soit atteint et ajoute un '\0' et les stock dans la chaîne Elle a un grand désavantage – il n`y a pas de contrôle de la longueur de la chaîne. Si la les caractères lus sont plus nombreux que le nombre d'éléments de la chaîne de réception, la dernière va être débordée. Elle élimine le caractère '\n' – fin de la ligne. .

La fonction fgets(chaîne,nombre_max_de_char, fichier)

Cette fonction lit les caractères depuis in fichier. Mais le nombre de caractères est limité par le deuxième paramètre *nombre_max_de_char*, c'est-à-dire qu'elle va lire jusqu'à la fin de la ligne ou quand le nombre de caractères lus devient égal à *nombre_max_de_char*, lequel vient le premier. Si on lit depuis le clavier on le paramètre réel pour fichier doit être *stdin*

La fonction puts(chaîne)

Affiche des caractères de la chaîne (les caractères jusqu'au '\0') sur stdout. Elle ajoute un caractère nouvelle ligne ('\n').

La fonction scanf

On utilise la spécification %s. On peut limiter le nombre des caractères lus. La fonction ajoute '\0'.

La fonction printf

On l'utilise de façon normale avec spécification %s.

Exemple

Dans l'exemple ci-dessous on peut voir l'effet de l'utilisation de toutes le fonctions énumérées au dessus, Les résultats sont affichés au dessous du programme. Avec caractères noirs sur fond clair est montrée la saisie et l'affichage est imprimé par caractères blancs sur fond noir.

```
#include <stdio.h>
void main (void) {
  char ch1[50];
  char ch2[10];
  int i;
  scanf("%s", ch1); puts(ch1);
  printf("%s\n",ch1);
  gets(ch1); puts(ch1);
  printf("%s\n",ch1);
  fgets(ch1,50,stdin); puts(ch1);
  scanf("%5s%d",ch2,&i);
  printf("|%s|  |%d|\n",ch2,i);
  printf("|%5.2s|  |%3d|\n",ch2,i);
}
```

```
C'est une chaine de caracteres
C'est
C'est
 une chaine de caracteres
 une chaine de caracteres
Une autre chaine de caracteres
Une autre chaine de caracteres

abcd123
|abcd1|  |23|
|   ab|  | 23|
```

Les fonctions standard

Les fonctions standard qui manipulent les chaînes de caractères sont nombreuses. Ici nous allons donner quelques d'eux. Elles son montrées avec les nom des fichiers en-tête (header files) dans lesquels se trouvent leurs définitions.

La longueur

int strlen(char *s) – string.h
Rends la longueur de *s*, c'est à dire le nombre de caractères qui se trouvent entre l'adresse qui est la valeur de *s* et le premier caractère '\0'.

Fonctions de conversion

int atoi(char *s) – stdlib.h
Elle convertit *s* (qui doit être la présentation d'un nombre entier) dans un entier

long atol(char*) – stdlib.h
Elle convertit *s* (qui doit être la présentation d'un nombre entier) dans un entier long.

double atof(char*) – stdlib.h
Elle convertit *s* (qui doit être la présentation d'un nombre réel) dans un double.

char *itoa(int value, char *string, int radix) – stdlib.h
Elle représente la valeur entière value dans une chaîne *string* dans le système numérique de base *radix*.

int toupper(int ch) – ctype.h
Elle rends le caractère *ch* converti en majuscule.

int tolower(int ch) – ctype.h
Elle rends le caractère *ch* converti en minuscule.

Concaténation et copie

Ces fonctions ont deux variants:

- fonction simple – qui traite toute la chaîne n'import quelle est sa longueur.
- fonction avec restriction – elle traite au maximum *maxl* caractères, où *maxl* est un paramètre entier. Ces fonctions ont un *n* supplémentaire dans leurs noms.

char *strcat(char *dest, const char *src) – string.h
Elle colle une copie de *src* au *dest*

char *strncat(char *dest, const char *src, size_t maxlen)
Elle colle une copie de *src* au *dest* mais au maximum *maxl* caractères et elle n'ajout pas un '\0'.

char *stpcpy(char *dest, const char *src) – string.h
Elle copie *src* au *dest*.

char *strncpy(char *dest, const char *src, size_t maxlen)
Elle copie *src* au *dest* mais au maximum maxl caractères et elle n'ajout pas un '\0'.

Comparaison

int strcmp(const char *s1, const char *s2) – string.h
Comparer les *s1* et *s2*. Le résultat est : *-1 – s1>s2, 0 – s1=s2, 1 – s1<s2*

int strncmp(const char *s1, const char *s2, size_t maxlen)
Comparer les *s1* et *s2*. Le résultat est : *-1 – s1>s2, 0 – s1=s2, 1 – s1<s2*. Mais elle traite au maximum *maxl* caractères.

int stricmp(...), int strincmp(...) –
Elles comparent les chaînes ignorant la différence entre majuscules et minuscules

Recherche

char *strchr(const char *s, int c) – string.h
Elle cherche la 1-ère occurrence de *c* en *s*. Le résultat est pointeur vers l'occurrence ou NULL si *c* n'existe pas

char *strrchr(const char *s, int c)
Elle cherche la dernière occurrence de *c* en *s*. Le résultat est pointeur vers l'occurrence ou NULL si *c* n'existe pas

char *strstr(const char *s1, const char *s2)
Elle cherche la 1-ère occurrence de *s2* en *s1*. Le résultat est pointeur vers l'occurrence ou NULL si *s2* n'existe pas

Fonctions de classification – ctype.h

int isalpha(int c)
Elle rends 1 si *c* est une lettre et 0 dans l'autre cas.

int isdigit(int c)
Elle rend 1 si *c* est un chiffre et 0 dans l'autre cas.

int isspace(int c)

Elle rend 1 si *c* est une espace, une tabulation ou nouvelle ligne et 0 dans les autres cas.

int isupper(int c)
Elle rend 1 si *c* est une lettre majuscule et 0 dans l'autre cas.

int islower(int c)
Elle rend 1 si *c* est une lettre minuscule et 0 dans l'autre cas.

Insertion et suppression

Rendre le résultat dans une autre chaîne

La fonction *strdela* supprime de la chaîne *src* les n caractères en commençant de celui à place numéro *from* et stocke le résultat dans la chaîne *dest*.

```
char * strdela(char *dest, char *src, int from, int n){
  if (from > strlen(src)) return src;
  strncpy(dest,src,from); // copier les caractères avant les supprimés
  dest[from]='\0'; //marque la fin de la première sous chaîne
  strcat(dest, src+from+n); //coller les caractères après les supprimés
  return dest;
}
```

La fonction *insera* insère la chaîne *ins* dans la chaîne *src* en commençant de caractère numéro *from* et stocke le résultat dans la chaîne *dest*.

```
char* insera(char *dest, char *src,int from, char *ins){
      char *p=src+from; int i;
      if (from > strlen(src)) return dest;
      for(i=0;i<from;i++)dest[i]=src[i]; // copier les caractères avant l'insertion
                          // ou strncpy(dest,src,from)
      dest[from]='\0'; //marque la fin de la première sous chaîne
      strcat(dest,ins);// coller la chaîne insérée
      strcat(dest,p); //coller les caractères après l'insertion
      return dest;
}
```

Dans les deux fonctions il y a un paramètre *dest* pour le résultat. On ne peut pas créer une chaîne dans la fonction et rendre l'adresse de cette chaîne par l'instruction return parce que la chaîne est locale dans la fonction et va être détruit à la sorte de la fonction. De cette façon la valeur (l'adresse) retournée

Rendre le résultat dans la même chaîne

La fonction *strdel* supprime de la chaîne *s* les n caractères en commençant de celui à place numéro *from* et stocke le résultat dans la chaîne *s*.

```
char * strdel(char *s, int from, int n){
  if (from > strlen(s)) return s;
  if (from + n > strlen(s)) // copier les caractères avant les supprimés
     n = strlen(s) - from; //calcule l'adresse du premier caractère après
                           //les supprimés
  strcpy(s+from, s+from +n); //déplace les caractères après les supprimés
  return s;
}
```

La fonction *inser* insère la chaîne *ins* dans la chaîne *dest* en commençant de caractère numéro *from* et stocke le résultat dans la chaîne *dest* .La chaîne *dest* doit être assez large pour être capable de contenir la chaîne résultat

```
char* inser(char *dest, int from, char *src){
      char *p=dest+from;
      char *p1 = p+strlen(src);
      int i;
      if (from > strlen(dest)) return dest;
      for(i=strlen(p);i>=0;i--)p1[i]=p[i]; //déplace les caractères après la place
                     //d'insertion à l'arrière pour libérer place pour l'insertion
      strncpy(p,src,strlen(src)); //copier les caractères insérés à la place libre
      return dest;
}
```

Exemple

Vérifier di une chaîne de caractères est palindrome ou non.

Une chaîne est un palindrome si elle peut être lue de gauche à droite ou de droite à gauche en conservant le même sens. On ignore les signes de ponctuation, les espaces et la différence entre majuscules et minuscules. Exemples:

- radar
- Elu par cette crapule
- Esope reste ici et se repose

L'algorithme général est montre à fig. 8.5. On voit deux étapes essentielles. La première est de préparer la chaîne pour la vérification – supprimer tout signe de ponctuation et tout espace et de convertir toutes les lettres en minuscules. La deuxième étape est la vérification elle-même. Pour cette raison sont développées deux fonctions: *prep* et *pal*.

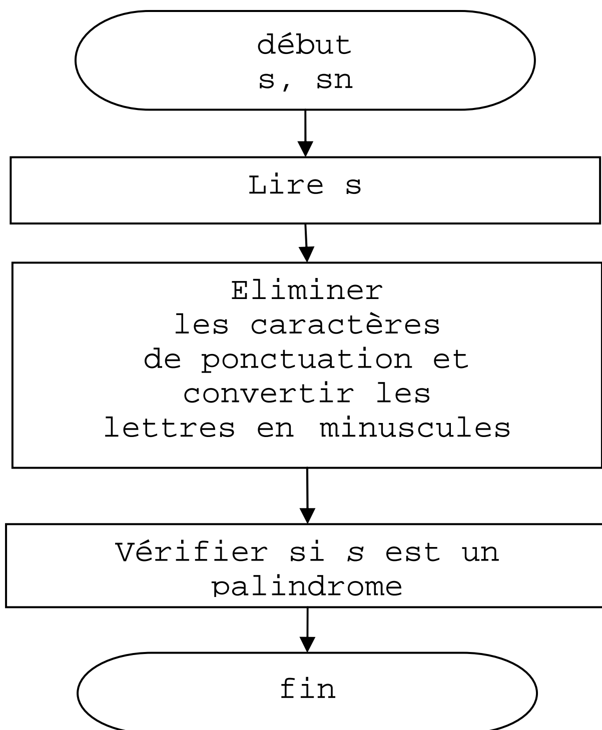
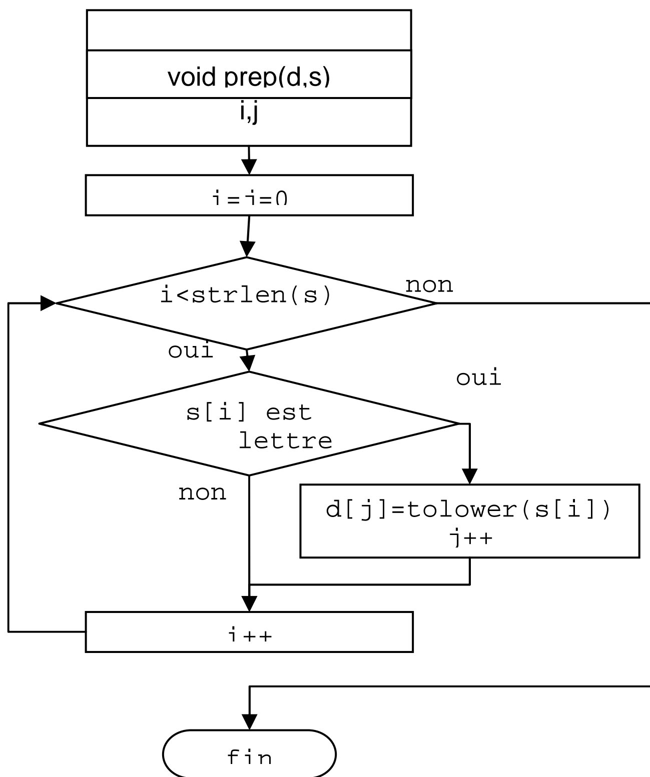


Fig. 8.5

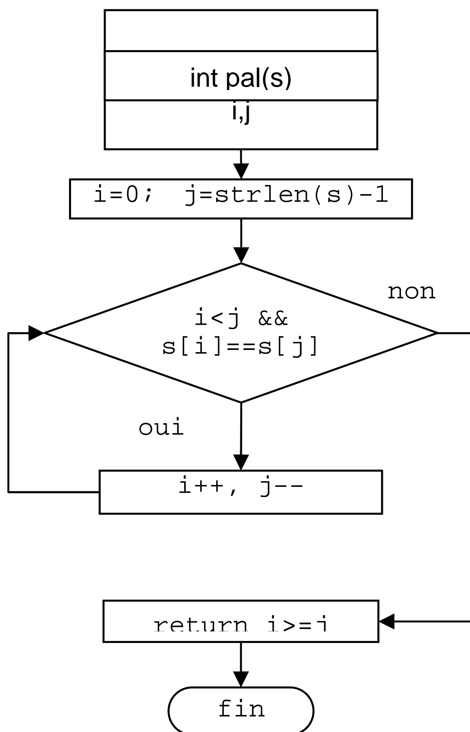
L'algorithme et le programme de *prep* sont montrés à fig.8.6. *prep* copie les lettres converties en minuscules de son deuxième paramètre *s* dans son premier paramètre *d*.

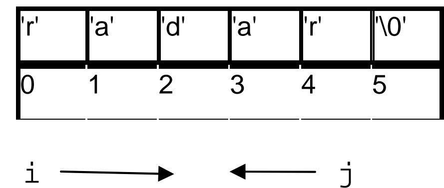
L'algorithme et le programme de *pal* sont montrés à fig.8.7. *pal* vérifie si son paramètre *s* est un palindrome (rand 1) ou n'est pas un palindrome (rend 0). D'abord la variable *i* est initialisée par la valeur 0 (l'indice du premier caractère de la chaîne) et *j* est initialisée par la valeur *length(s)-1* (l'indice du dernier caractère de *s*). A chaque répétition de la boucle on compare le *i*-ème et le *j*-ème caractère et s'ils sont égaux on continue en augmentant *i* par 1 et en diminuant *j* par 1. La boucle termine soit quand *i* et *j* se rencontrent (*i* devient supérieur ou égal à *j*) et dans ce cas la chaîne est un palindrome, soit quand les deux caractères comparés ne sont pas égaux et dans ce cas la chaîne n'est pas un palindrome.




```
void prep(char *d, char*s){
 int i,j;
   for (i=j=0; i <strlen(s);i++){
    if (isalpha(s[i])){
      d[j]=tolower(s[i]);
       j++;
    // d[j++] = tolower(s[i]);
    }
      d[j]='\0';
   }
}
```

Fig. 8.6






```
int pal(char *s){
  int i,j;
  for (i=0, j=strlen(s)-1;
       i<j && s[i]==s[j]; i++,j--);
  return i>=j;
}
```

Fig. 8.7

Tableaux de chaînes de caractères

Présentation de tableaux de chaînes

Chaque chaîne est un tableau unidimensionnel de caractères. Donc le tableau de chaîne sera un tableau bidimensionnel. On peut distinguer quelques cas.

Tableau rectangulaire

La présentation de la déclaration suivante est montrée à la fig. 8.8 les chaînes sont placées une après l'autre dans la mémoire.

```
char tc[3][7]= {"un","deux","trois"};
```

tc:

'u'	'n'	'\0'				
'd'	'e'	'u'	'x'	'\0'		
't'	'r'	'o'	'i'	's'	'\0'	

Fig. 8.8

Dans ce cas on peut modifier les caractères des chaînes mais pas leur ordre.

```
tc[0][1] = 't';
strcat(tc[0],tc[1]);
```

Tableau de pointeurs vers chaînes constantes

La déclaration suivante initialise un tableau de pointeurs avec trois strings constants. Sa présentation est montrée à fig. 8.9, Les chaînes peuvent ne pas être voisines dans la mémoire.

```
char *tc[]= {"un","deux","trois"};
```

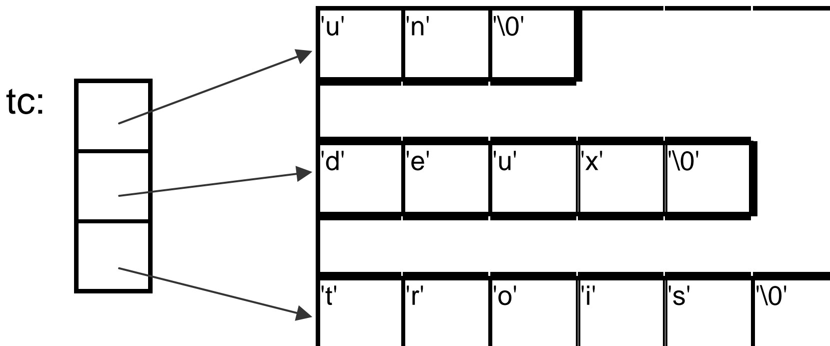


Fig. 8.9

Dans ce cas on peut changer l'ordre (les indices) des chaînes ou d'affecter autres chaînes mais on ne peut pas changer les caractères des chaînes originales

```
tc[2] = "quatre";
tc[0]= tc[1];
```

Présentation mixte

On peut déclarer des chaînes comme des tableaux unidimensionnels ou bidimensionnels et puit affecter leur adresse à un tableau de pointeurs (Fig. 8.10). C'est le schéma le plus flexible (en excluant l'allocation de mémoire dynamique).

```
char tc[3][7]= {"un","deux","trois"};
char *tp[] = {tc[0],tc[1],tc[2]};
char mot1[20],mot2[25];
char *tab[]={mot1,mot2}
```

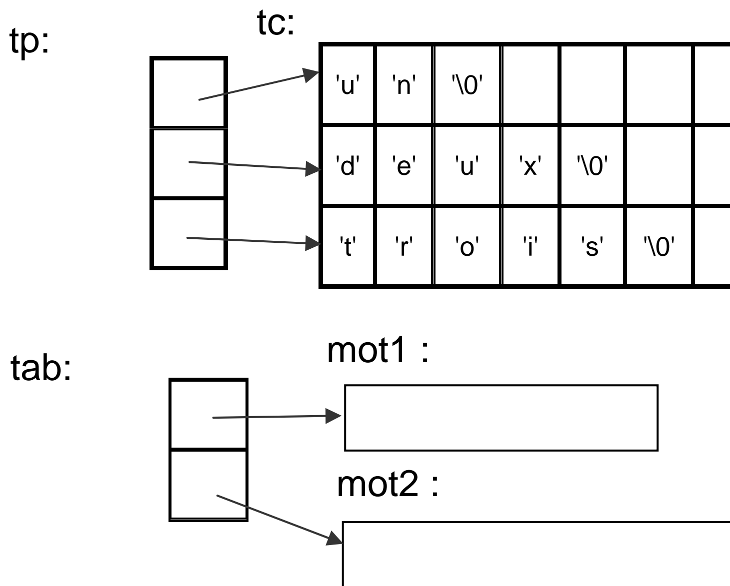
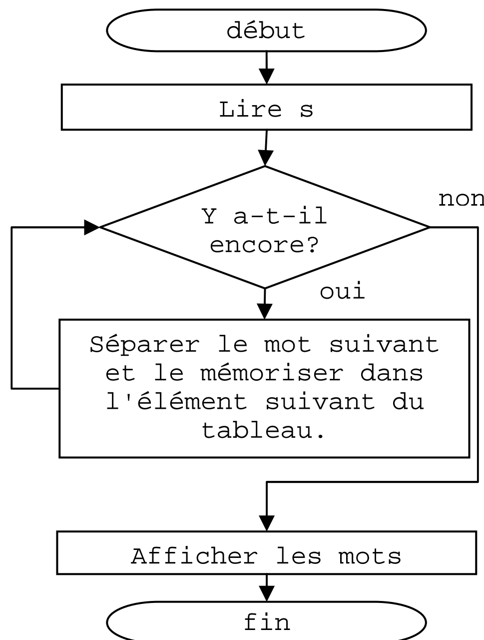


Fig. 8.10

Exemples

Exemple1 – Lire un texte et mettre tous les mots dans un tableau

D'bord on doit spécifier que le texte peut avoir plusieurs lignes et on doit définir le terme mot. Ici on va appeler un mot chaque suite de lettre qui contient éventuellement des tirets. L'algorithme est montré à la fig. 8.11. On va lire le texte dans un tableaux de strings dont chaque élément représente une ligne de teste. Puis dans une boucle on doit trouver les mots un par un et les stocker dans un autre tableau appelé dictionnaire ou d. La dernière étape est d'afficher le dictionnaire. Deux types sont déclarés : *ligne* qui est un string au maximum de 79 caractères et représente une ligne de texte et *mot* qui et un string au maximum de MOTLEN-1 caractères où MOTLEN est une constante définie (20). Dans le programme sont déclarées deux variables: *tex* qui est un tableau de ligne et représente le texte original et *dict* qui est un tableau de mot et représente le dictionnaire.



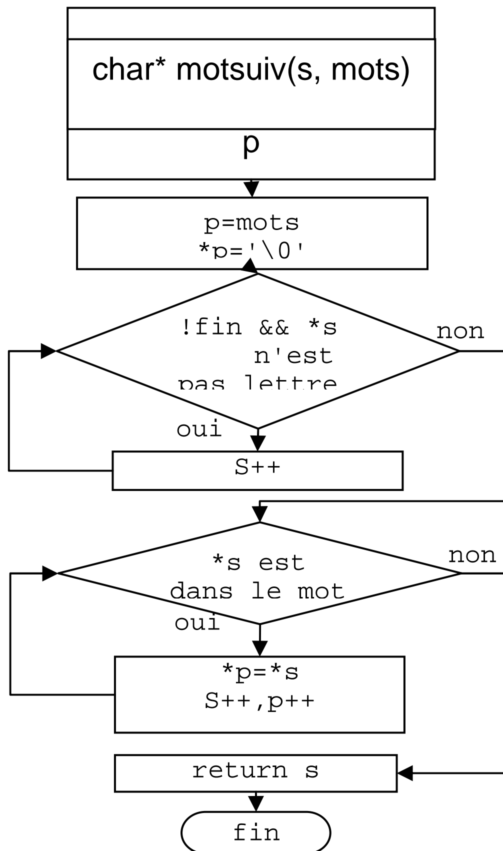

```
typedef char ligne[100];
typedef char mot[MOTLEN];
...
ligne tex[MAXLIGNES];
mot dict[MAXMOTS];
```

Fig. 8.11

Ici sont développées deux fonctions:

La fonction *prends_mot_suivant(char *source, mot mots)* trouve le premier mot de la chaîne dont le début est donné par *source* et stocke se caractères en *mots* (fig. 8.12). Elle rends comme résultat un pointeur qui désigne le

caractère suivant le mot séparé. Elle comporte deux boucle : la première passe tous les caractères jusqu'à la première lettre et la deuxième prends toutes lettres et éventuellement les tirets jusqu'au premier caractère qui ne peut pas se trouver dans le mot. Chaque caractère trouvé dans cette boucle est stocké dans *mots*. Enfin le caractère '\0' est ajouté pour fermer le mot. Au cours de l'exécution des boucles la valeur de *source* est modifiée pour désigner toujours le caractère courant. A la fin il va désigner le caractère qui suit le dernier caractère du mot séparé et cette valeur est rendue comme résultat. Si la fin de la ligne est atteinte avant de trouver un mot NULL est rendu.




```
char* prends_mot_suivant(
   char *source, mot mots){
  char *p = mots;
  *p = '\0';
  while ( *source != '\0' &&
       ! isalpha(*source)) source++;
    if ( *source == '\0') return NULL;
    while (isalpha(*source) ||
           *source =='-')
             *p++ = *source++;
    *p = '\0';
    return source;
}
```

Fig. 8.12

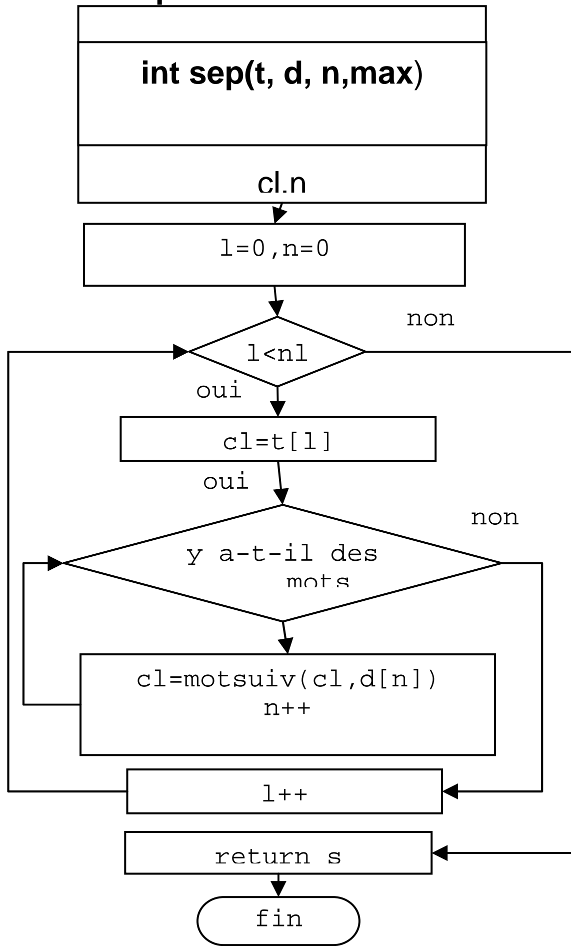
La fonction *separer_mots(ligne texte[], mot dict[], const int nl, const int maxmot)* utilise la fonction précédente pour trouver toutes les mot du texte *ligne* (fig.8.13), qui contient *nl* lignes, et les mémoriser dans le dictionnaire *dict* qui peut contenir au maximum *maxmot* mots. Le résultat est le nombre des mots trouvés. Dans le programme il y a un moment obscur. Comment on test la fin de la ligne. La condition pour la continuation de la boucle

```
while (n<maxmot && curligne !=NULL && *curligne != '\0')
```

contient 3 comparaisons. La première assure que le dictionnaire ne puisse pas être débordé, la deuxième et la troisième vérifient si la fin de ligne n'est pas atteinte soit sans séparer un mot, soit après le dernier caractère du mot. Ici *curligne* est pointeur vers le caractère courant. Dans l'instruction

```
if ((curligne=prends_mot_suivant( curligne,dict[n])) != NULL) n++;
```

la fonction *prends_mot_suivant* est appelée avec premier paramètre *curligne* et le résultat est affecté aussi à *curligne*. De tel façon on fourni à la fonction l'adresse du caractère courant avant le mot suivant et puis on reçoit comme résultat l'adresse du caractère qui suit le mot déjà séparé. On test se résultat et s'il n'est pas NULL on augmente le nombre des mot séparés. Les parenthèses sont essentielles parce que l'affectation à une priorité plus basse que celle de la comparaison.




```
int separer_mots(ligne texte[],mot dict[],
  const int nl,  const int maxmot){
  char *curligne;
  int l, n=0;
  for (l=0; l<nl; l++) {
    curligne = texte[l];
    while (n<maxmot && curligne !=NULL
      && *curligne != '\0'){
     if ((curligne=prends_mot_suivant
       ( curligne,dict[n])) != NULL) n++;
    }//while
  }//for
  return n;
}
```

Fig. 8.13

Exemple2 – trier les mots

Nous allons utiliser le tableau *dict* pour écrire la fonction *trie_mots(char *dict[],* ***const int*** *n)* (Fig.8.14). On utilise le trie à la boule. La comparaison se fait à l'aide de la fonction standard *strcmp* et on change seulement les valeur des pointeur sans déplacer les mots eux-mêmes. L'état initial est montré par flèches pointillées et l'état final est dessiné par flèches solides.

```
void trie_mots(char *dict[],
          const int n){
  // trie les mots a la boule
  char *t;
  int i,exch;
  do{
    exch = 0;
    for (i=1;i<n;i++) {
      if (strcmp(dict[i-1],dict[i])>0){
       exch =1;
       t =dict[i-1];
       dict[i-1]=dict[i];
       dict[i]=t;
      }
    }
  }while (exch);
}
```

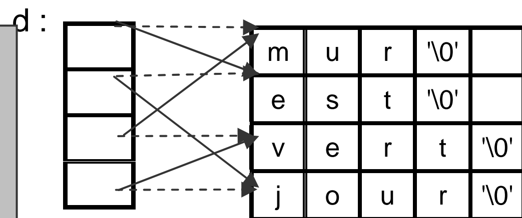


Fig. 8.14