Rapport de projet C
Mise en forme d'un texte

Mehdi MAKHLOUF
Mohamed ELAFRIT
Rémi BUISSON
Sanaa KAIDI

LaTeX

Table des matières

Chapitre 1

Introduction

Dans ce projet on va s'intéresser à créer une bibliothèque de mise en forme d'un texte, une chaîne de caractères constituée de mots, d'espacements et de signes de ponctuation, et écrire un programme permettant de le manipuler.
Ce texte doit être lu sur l'entrée standard et affiché sur la sortie standard.
L'utilisation de stdin et stdout permet leur redirection.
On peut ainsi traiter un texte contenu dans un fichier, par une simple redirection de stdout vers stdin.
De la même manière on peut sauvegarder le texte traité, par une simple redirection de stdout vers un fichier.

Maintenant on se pose la question : qu'est-ce qu'on entend par traiter un texte ?
Pour la plupart des utilisateurs traiter un texte signifie obtenir des statistiques dessus, et vérifier la typographie et l'orthographe et avoir la possibilité de justifier le texte.

Dans un premier chapitre nous verrons comment représenter le texte à traiter en mémoire. Ensuite nous nous attarderons sur l'organisation du programme en termes de fichiers sources et fichiers headers.

Chapitre 2

La représentation du texte en mémoire

Dans ce chapitre nous présenterons l'utilité d'une représentation mémoire adaptée du texte. Ensuite nous envisagerons quelques solutions pour cette représentation et enfin, la solution retenue sera explicitée.

2.1 Pourquoi utiliser des structures

Le but de l'utilisation des structures est de représenter un objet, abstraction du monde réel, en mémoire. C'est ce concept qui fait du C un langage orienté objet.
En effet, au lieu de représenter, par exemple, une voiture par plusieurs variables indépendantes, on définit un objet de type structure voiture qui contient un entier pour son kilométrage, une chaîne pour sa marque ...
Ainsi, dans notre projet, il est nécessaire d'utiliser ce concept dans le but de ne pas banaliser un texte : ce n'est pas une vulgaire chaîne de caractères mais bien un ensemble de mots, caractères d'espacement et signes de ponctuation.

Voyons maintenant les différentes solutions de repésentation qui s'offrent à nous.

2.2 Quelques représentations possibles

Il y a différentes manières de représenter un texte.
On peut facilement imaginer qu'un texte est une succession de mots, de caractères d'espacements et de signes de ponctuation.

Nous pouvons alors représenter le texte comme une suite d'éléments définis par leur type (mot, espacement, ponctuation).
Les termes "suite d'éléments", sous-entendu ordonnée, nous amène à réfléchir sur l'utilisation de tableaux ou listes chaînées.

Demandons nous comment peut-on définir un texte ?
Un texte est un ensemble de paragraphes, eux-même composés de phrases et elles-mêmes composées d'éléments de phrase qui sont des mots, des caractères d'espacement et des signes de ponctuation.

Tableaux d'éléments ou liste ? Voyons quelle solution nous allons choisir.

2.3 La solution retenue : une liste chaînée double

Les fonctions que nous allons concevoir vont être difficilement implémentables si nous ne montons pas plus haut dans le niveau d'abstraction.
En effet, un tableau contenant tous les éléments du texte peut être lourd à manipuler quand nous allons, par exemple, justifier le texte paragraphe par paragraphe.

C'est pourquoi le découpage précédent peut nous aider.

Cette représentation peut être faite via une liste doublement chaînée de manière à pouvoir naviguer entre les différents éléments du texte sans retourner à son début. C'est la solution que nous retiendrons (cf. 9.1).

Cette représentation est implémentée dans le fichier `structures.h` (cf. 4).

2.3.1 Le type texte

Le type texte est un moyen plus simple de nommer un pointeur sur une liste double chaînée de paragraphes (cf. 2.3.2). En effet il est beaucoup plus parlant pour nous, par exemple, d'appeler la fonction `texte__afficher()` avec un paramètre de type "texte" qu'avec un "pointeur sur une liste double chaînée de paragraphes". Par conséquent nous avons défini le type texte comme un pointeur sur une liste double chaînée de paragraphes.

2.3.2 La structure paragraphe

La stucture paragraphe est une liste double chaînée de paragraphes. En effet un texte est une succession de paragraphes. Nous pensons que cette représentation mémoire est la mieux adaptée car c'est la plus logique, la plus compréhensible pour l'esprit humain.
Cette structure contient plusieurs éléments :

- `nb_mots` : entier contenant le nombre de mots du paragraphe,
- `nb_car` : entier contenant le nombre de caractères du paragraphes,
- `phrase` : pointeur sur le premier élément d'une liste double chaînée de phrase (celle du paragraphe concerné cf. 2.3.3),
- `suivant` : pointeur sur l'élément suivant (dans la liste) de type paragraphe ; ce pointeur est à NULL pour le dernier élément de la liste,
- `précédent` : pointeur sur l'élément précédent (dans la liste) de type paragraphe ; ce pointeur est à NULL pour le premier élément de la liste.

2.3.3 La structure phrase

La stucture phrase est une liste double chaînée de phrases. En effet un paragraphe est une succession de phrases.
Cette structure contient plusieurs éléments :

- `elem` : pointeur sur le premier élément d'une liste double chaînée d'éléments de phrase (ceux de la phrase concernée cf. 2.3.4),
- `suivant` : pointeur sur l'élément suivant (dans la liste) de type phrase ; ce pointeur est à NULL pour le dernier élément de la liste,
- `précédent` : pointeur sur l'élément précédent (dans la liste) de type phrase ; ce pointeur est à NULL pour le premier élément de la liste.

2.3.4 La structure element_phrase

La stucture element_phrase est une liste double chaînée d'éléments de phrase (mots, espacements, ponctuations). En effet une phrase est une succession de ces éléments .
Cette structure contient plusieurs éléments :

- `type` : c'est un caractère qui indique le type de l'element_phrase. Il prend comme valeur : M pour un mot, E pour un espace, P pour une ponctuation et N pour un saut de ligne ('\n'),
- `taille` : un entier qui indique la taille de la chaîne de caractères pointée par `chaine`,
- `justification` : un caractère qui nous sert de tableau de 8 booléens (cf. FIG. 2.1),
- `nb_espav` : un entier qui donne le nombre d'espaces à mettre avant l'élément (dans le cadre des fonctions de justification),
- `pos_cesure` : un entier qui donne la position de la cesure dans la chaîne (dans le cadre des fonctions de justification),
- `nb_espap_cesure` : un entier qui donne le nombre d'espaces à mettre après la cesure (la partie de l'élément sur la nouvelle ligne) lors de l'appel à la justification droite avec césure,
- `chaine` : c'est un pointeur sur une chaîne de caractères,
- `suivant` : pointeur sur l'élément suivant (dans la liste) de type element_phrase ; ce pointeur est à NULL pour le dernier élément de la liste,
- `precedent` : pointeur sur l'élément précédent (dans la liste) de type element_phrase ; ce pointeur est à NULL pour le premier élément de la liste.

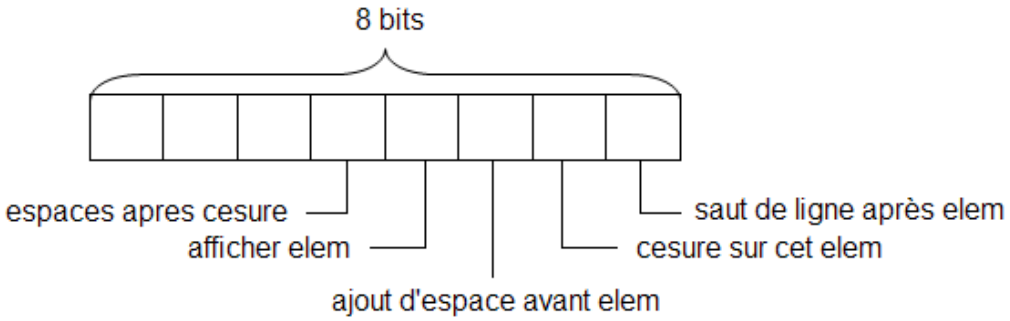

FIG. 2.1 – Utilisation du char comme tableau de booléen

Chapitre 3

L'implantation du dictionnaire

3.1 La problématique

Dans ce projet, nous devons implanter une fonction qui vérifie l'orthographe dans le texte.
Ainsi, il nous faut donc élaborer un dictionnaire contenant tous les mots que nous allons considérer comme corrects.

3.2 Les solutions envisageables

Trois idées nous viennent à l'esprit.
La première consisterait à implanter un arbre "n-aire" où chaque père aurait comme fils l'une des vingt six lettres de l'alphabet.
Une autre solution serait d'interfacer le programme avec un automate ; on n'aurait qu'à définir l'ensemble des états accepteur.
La troisième et dernière solution serait de faire un tableau multi-dimentionnel (cf. 3.1). L'indice de la case correspondrait au numéro de la première lettre du mot et ensuite, pour chaque case, plusieurs cases stockeraient les mots débutant par cette lettre.

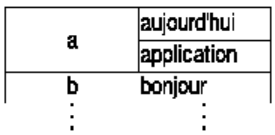

FIG. 3.1 – Représentation du dictionnaire de mots

3.3 La solution retenue : le tableau multi-dimentionnel

Etant donné le nombre de mots restreint que notre dictionnaire contient, la première solution n'est pas envisageable.
La seconde solution est compliquée pour un projet de petite taille comme celui-ci : elle n'est pas retenue.
La troisième est donc celle qui sera utilisée dans le projet. Ainsi, par exemple, pour le mot "application", on prend sa première lettre.
On obtient le caractère 'a'.

Avec la fonction `toupper()`, on obtient 'A'.
La valeur du code ASCII de 'A' est 65.
En prenant la case 'A' $-65 = 0$ on accède bien à la première case du tableau.
On a donc accès à la case en temps constant et on se limite à la recherche dans le sous tableau `tab[0]`.

Chapitre 4

L'organisation des fichiers

4.1 Le problème posé par un projet de développement

L'implémentation d'un projet ne se résume pas à celle de deux ou trois fonctions.
Il va donc y avoir nécessairement un moment où la taille du fichier source, en terme de nombre de lignes de code, va influer sur la rapidité à programmer.
Il faut donc trouver un moyen pour découper le code en plusieurs fichiers distincts.

4.2 Les découpages possibles

Il n'y a aucun intérêt à découper le programme en plusieurs fichiers sources en terme de gain de rapidité à l'exécution ou à la compilation.
Par contre, c'est pour l'humain autant utile que d'indenter le code. C'est un moyen d'améliorer la lisibilité de celui-ci.

On peut le découper en utilisant différents fichiers sources et headers.
En effet, si on veut, par exemple, modifier le code d'une fonction statistique on ouvre simplement le fichier fonctions_statistiques.c (contenant l'implémentation de toutes les fonctions statistiques). De la même manière, l'utilisation de fichiers "header" permet de voir, par exemple, directement le code la fonction main(), en évitant de lire tous les prototypages de fonctions écris au préalable.

4.3 Le découpage effectué

Cette section explique la manière dont est organisé le projet. Chaque flèche signifie "a besoin de".

4.3.1 Le fichier d'entête structures.h

Ce fichier contient les définitions de toutes les structures (cf. 2.3) ainsi que les directives pré-processeur (`#define CONSTANTE VALEUR`).

4.3.2 Les fichiers fonctions.*

Le fichier d'entête fonctions.h

Ce fichier contient les prototypes des fonctions de lecture sur l'entrée standard, l'affichage sur la sortie standard et les fonctions de gestion de la mémoire allouée au cours de l'exécution. Les principales fonctions contenues dans ce fichier sont :

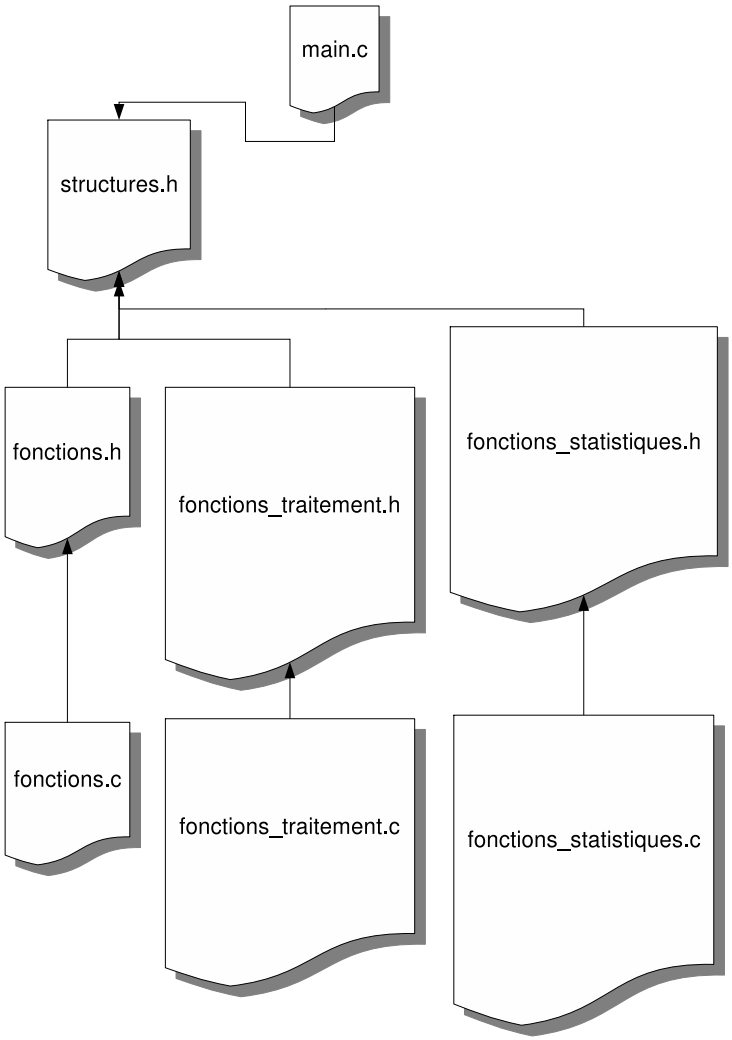

FIG. 4.1 – Organisation des fichiers au sein du projet

- `init__para` : initialise le tableau de structures `struct paragraphe`,
- `init__phrase` : initialise le tableau de structures `struct phrase`,
- `init__element` : initialise le tableau de structures `struct element_phrase`,
- `texte__lire` : lit le texte sur l'entrée standard,
- `texte__afficher` : affiche le texte sur la sortie standard,
- `reinit__justification` : ré-initialise le `char` justification de chaque `struct element_phrase` en le mettant à 0.

Le fichier fonctions.c

Ce fichier contient les implémentations des fonctions dont le prototype est definie dans le fichier fonctions.h (cf. 4.3.2).

4.3.3 Les fichiers fonctions_statistiques.*

Le fichier d'entête fonctions_statistiques.h

Ce fichier contient les prototypes des fonctions statistiques. Les fonctions principales contenues dans ce fichier sont :

- `texte__nb_car` : renvoie le nombre de caractères,
- `texte__nb_mots` : renvoie le nombre de mots du texte,
- `texte__nb_par` : renvoie le nombre de paragraphe,
- `texte__nb_par` : renvoie le nombre de lignes du texte.

Le fichier fonctions_statistiques.c

Ce fichier contient les implémentations des fonctions dont le prototype est définie dans le fichier fonctions_statistiques.h (cf. 4.3.3).

4.3.4 Les fichiers fonctions_traitement.*

Le fichier d'entête fonctions_traitement.h

Ce fichier contient les prototypes des fonctions de traitement sur le texte. Les fonctions principales contenues dans ce fichier sont :

- `cesure__mot` : renvoie un booléen à vrai si une cesure est possible sur le mot (double même consonne),
- `faire__cesure_ligne` : le nom de cette fonction n'est pas vraiment explicite car son vrai usage est de répartir les espaces avant les mots lors de la justification totale,
- `texte__justifier` : justifie le texte (remplie les champs d'informations de justifications des structures `struct element_phrase` (cf. 2.3.4) interprétées lors de l'affichage) ; si les modifications n'ont pas pu être effectuées, la fonction renvoie un booléen à faux,
- `texte__verif_orthographe` : renvoie le numéro du premier mot mal orthographié (0 correspond à un texte bien orthographié),
- `texte__typographie` : vérifie la typographie du texte et y apporte les modifications nécessaires ; si les modifications n'ont pas pu être effectuées, la fonction renvoie un booléen à faux.

Le fichier fonctions_traitement.c

Ce fichier contient les implémentations des fonctions dont le prototype est définie dans le fichier fonctions_traitement.h (cf. 4.3.4).

4.3.5 Le fichier makefile

Ce fichier contient les règles de construction du projet (fichiers sources et dépendances). Il utilise la commande `$>make`. Ainsi on utilise :

- `$>make` ou `$>make prog` pour compiler le projet,
- `$>make clean` pour supprimer les fichiers intermédiaires (`$>*.o *` ...),
- `$>make mrproper` pour supprimer les fichiers intermédiaires, tout comme `make clean` ainsi que le fichier compilé ; cette commande peut être utile lorsqu'on envoit le projet dans une archive, pour inciter l'utilisateur à compiler et ne pas exécuter le fichier existant.

Chapitre 5

Idée générale du fonctionnement des fonctions principales

5.1 La fonction `texte__lire`

Cette fonction retourne un pointeur sur la liste de paragraphes.
Elle lit un à un les caractères sur l'entrée standard et en détermine le type. Le caractère courant est comparé à celui du précédent :

- Si les deux caractères sont de même type , on continue la lecture jusqu'à trouver un type différent ; à ce moment là l'élément précédent est créé.
- Si l'élément est un point on crée une phrase.
- Si l'élément est un saut de ligne on crée un nouveau paragraphe.

Ainsi on remplit les tableaux de structures passés en paramètres : (cf. 9.1).

5.2 La fonction `texte__afficher()`

Pour chaque paragraphe cette fonction parcourt les phrases et pour chacune d'entre elles elle affiche les éléments ainsi que leurs attributs de justification.

5.3 La fonction `texte__typographie()`

Elle vérifie et corrige les erreurs de typographie :

- le premier élément de la phrase est un mot dont la première lettre est une majuscule.
- s'il y a une ponctuation double elle vérifie qu'il y a un espace avant et un autre après sinon elle les ajoute.
- s'il n'y a pas un point à la fin de la phrase, elle le met.
- s'il y a un espace entre un mot et un point ou une virgule, elle l'enlève.

– elle vérifie s'il y a au moins un mot entre deux ponctuations sinon elle renvoit faux.

5.4 La fonction `texte__verif_orthographe()`

Pour chaque paragraphe, pour chacune de ses phrases, pour chacun de ses éléments, si l'élément est un mot qui appartient au dictionnaire (la liste des mots acceptés, cf. 3.1) on continue le parcours jusqu'à trouver un mot qui n'est pas dedans ; dans ce cas la fonction retourne le numéro de ce mot, sinon elle retourne 0 (le 0^emot est mauvais, donc pas de faute).

5.5 La fonction `texte__justifier()`

On parcourt toutes les phrases de tous les paragraphes et on se focalise sur les éléments.
On met en place un compteur auquel on ajoute, pour chaque élément, la taille de celui-ci.
Lorsque le compteur dépasse la limite fixée par l'utilisateur, celà veut dire que l'élément courant est sur la ligne suivante.
On applique donc, suivant le mode de justification, différentes options de justifications :

- si on est en mode de justification gauche, on lui ajoute un '\n' après,
- si on est en mode de justification gauche avec césure, on lui ajoute un '\n' après s'il n'y a pas de césure et sinon un "-\n",
- si on est en mode de justification droite, on lui ajoute un '\n' après et on met le nombre d'espaces qu'il faut pour arriver aux nombres de caractères total devant le premier élément de la ligne précédente,
- si on est en mode de justification droite, on lui ajoute un '\n' après s'il n'y a pas de césure et sinon un "-\n" ; on met le nombre d'espaces qu'il faut pour arriver aux nombres de caractères total devant le premier élément de la ligne précédente,
- si on est en mode de justification totale, on fait quasiment la même chose que pour la justification droite, sauf que l'on répartit les espaces (qu'on aurait mis devant le premier élément) entre tous les élements de la ligne hormis le premier.
- si on est en mode de justification totale avec césure, on fait quasiment la même chose que pour la justification droite avec césure, sauf que l'on répartit les espaces (qu'on aurait mis devant le premier élément) entre tous les éléments de la ligne hormis le premier.

5.6 Les fonctions statistiques

Pour ces fonctions, nous mettons simplement à profit les champs des structures prévues à cet effet.
Par exemple, pour la fonction `texte__nb_car()`, on parcourt tous les paragraphes et on additionne simplement tous les `nb_car` de chaque élément de la liste chaînée de paragraphes.

Chapitre 6

Jeux d'essais

Il est important de réaliser des jeux d'essais lorsqu'on a terminé un programme.
C'est grâce à cela qu'il est possible d'observer le comportement pratique du programme qui est parfois différent de ce qu'on attend.
En effet, il est essentiel de savoir ce que fait réellement une fonction et non ce qu'elle est censée faire.
Tout au long de ces jeux d'essais, nous utiliserons le texte suivant :

```
ceci␣est␣un␣exemple␣de␣texte␣a␣traiter␣dans␣ce␣projet␣.nous␣sommes␣toujours
dans␣le␣meme␣paragraphe.
ici␣on␣,␣change␣de␣paragraphe.␣si␣ceci␣fonctionne␣je␣dis␣bravo␣!
bravo␣!
avec␣cesure:␣consonne␣voyelle.bonjour␣ca␣va␣?
;␣?␣Test.
```

6.1 Ce qui fonctionne

Tout d'abord vérifions que certaines erreurs, comme des lettres ou des nombres négatifs passés au programme comme un nombre de colonnes, ou même, une option qui n'existe pas, sont gérés :

```
$> cat exemple.txt | ./prog -argumentquinexistepas
Saisir votre texte :
```

Le programme ne fait rien, ce qui paraît logique puisque l'option n'existe pas et donc il ne faut appeler aucune fonction.

```
$>cat exemple.txt | ./prog -jg a -jd 0 -jt -9
Saisir votre texte :
Justification gauche: le nombre de colonnes doit etre strictement positif !


Justification droite: le nombre de colonnes doit etre strictement positif !


Justification totale: le nombre de colonnes doit etre strictement positif !
```

Le programme affiche des messages d'erreur car les nombres de colonnes passés aux options `-jg -jd -jt` (pour les options et leur signification, cf. 7.1) pour la justification ne sont pas valides (c'est un caractère).

Le texte entré n'étant pas très beau, nous allons vérifier, voire modifier sa typographie et utiliser ce texte modifié dans la suite des jeux d'essais.
Nous cacherons volontairement l'affichage de la vérification typographique. En effet, nous n'aurons pas besoin de cet affichage pour voir le fonctionnement des autres fonctions.

```
$>cat exemple.txt | ./prog -t
Saisir votre texte :

      ----------- Verification typographique -----------

La verification typographique a ete realisee avec succes !

Ceci␣est␣un␣exemple␣de␣texte␣a␣traiter␣dans␣ce␣projet.␣Nous␣sommes␣toujours
dans␣le␣meme␣paragraphe.
Ici␣on,␣change␣de␣paragraphe.␣Si␣ceci␣fonctionne␣je␣dis␣bravo␣!
Bravo␣!
Avec␣cesure␣:␣consonne␣voyelle.␣Bonjour␣ca␣va␣?
Test.
```

Le programme affiche un message indiquant à l'utilisateur que la typographie s'est bien passée.
En effet, les phrases commencent par des majuscules et se finissent par un point.
Il y a des espaces insécables entre un mot et une ponctuation et un espace après celle-ci.
Un espace sépare deux phrases.
Les ponctuations sont enlevées en début de phrase jusqu'à atteindre le premier mot.

Testons maintenant le programme pour la justification gauche sur 40 colonnes :

```
$>cat exemple.txt | ./prog -t -jg 40
Saisir votre texte :

      ----------- Justification Gauche -----------

Ceci␣est␣un␣exemple␣de␣texte␣a␣traiter
dans␣ce␣projet.␣Nous␣sommes␣toujours
dans␣le␣meme␣paragraphe.
Ici␣on,␣change␣de␣paragraphe.␣Si␣ceci
fonctionne␣je␣dis␣bravo␣!
Bravo␣!
Avec␣cesure␣:␣consonne␣voyelle.␣Bonjour
ca␣va␣?
Test.
```

Nous voyons, d'une part que la justification gauche fonctionne et d'autre part, ce qui est vraiment du ressort d'une justification, les espaces en fin de ligne sont enlevés.

Testons maintenant le programme pour la justification gauche avec césure sur 21 colonnes :

```
$>cat exemple.txt | ./prog -t -jgc 21
Saisir votre texte :

      ----------- Justification Gauche avec cesure -----------
```

```
Ceci␣est␣un␣exemple
de␣texte␣a␣traiter
dans␣ce␣projet.␣Nous
sommes␣toujours␣dans
le␣meme␣paragraphe.
Ici␣on,␣change␣de
paragraphe.␣Si␣ceci
fonctionne␣je␣dis
bravo␣!
Bravo␣!
Avec␣cesure␣:␣conson-
ne␣voyelle.␣Bonjour
ca␣va␣?
Test.
```

Nous constatons le bon fonctionnement de cette fonctionnalité.
Pour les autres tests, nous allons exécuter les fonctions de justification les unes à la suite des autres. Testons

maintenant le programme pour la justification gauche avec césure sur 21 colonnes :

```
$>cat exemple.txt | ./prog -t -jd 60 -jdc 30 -jt 15 -jtc 30
Saisir votre texte :

      ----------- Justification Droite -----------


Ceci␣est␣un␣exemple␣de␣texte␣a␣traiter␣dans␣ce␣projet.␣Nous
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sommes␣toujours␣dans␣le␣meme␣paragraphe.
␣␣␣␣␣Ici␣on,␣change␣de␣paragraphe.␣Si␣ceci␣fonctionne␣je␣dis
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣bravo␣!
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Bravo␣!
␣␣␣␣␣␣␣␣␣␣␣␣␣Avec␣cesure␣:␣consonne␣voyelle.␣Bonjour␣ca␣va␣?
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Test.




      ----------- Justification Droite avec cesure -----------


Ceci␣est␣un␣exemple␣de␣texte␣a␣traiter␣dans␣ce␣projet.␣Nous
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣sommes␣toujours␣dans␣le␣meme␣paragraphe.
␣␣␣␣␣Ici␣on,␣change␣de␣paragraphe.␣Si␣ceci␣fonctionne␣je␣dis
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣bravo␣!
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Bravo␣!
␣␣␣␣␣␣␣␣␣␣␣␣␣Avec␣cesure␣:␣consonne␣voyelle.␣Bonjour␣ca␣va␣?
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Test.



      ----------- Justification Totale -----------
```

```
Ceci␣␣␣est␣␣␣un
exemple␣␣␣␣␣␣de
texte␣a␣traiter
dans␣ce␣projet.
Nous␣␣␣␣␣␣sommes
toujours␣␣␣dans
le␣␣␣␣␣␣␣␣␣␣meme
paragraphe.
Ici␣␣␣on,␣change
de␣␣␣paragraphe.
Si␣␣␣␣␣␣␣␣␣␣ceci
fonctionne␣␣␣je
dis␣␣␣␣␣␣bravo␣!
Bravo␣!
Avec␣␣␣␣cesure␣:
consonne
voyelle.
Bonjour␣ca␣va␣?
Test.
```

```
      ----------- Justification Totale avec cesure -----------
```

```
Ceci␣est␣un␣exemple␣de␣texte␣a
traiter␣␣␣dans␣␣␣ce␣projet.␣Nous
sommes␣␣␣toujours␣␣␣dans␣le␣meme
paragraphe.
Ici␣␣␣on,␣change␣de␣paragraphe.
Si␣␣␣␣ceci␣␣␣fonctionne␣␣␣je␣␣␣dis
bravo␣!
Bravo␣!
Avec␣␣␣cesure␣:␣consonne␣voyel-
le.␣␣␣␣␣␣Bonjour␣␣␣␣␣␣ca␣␣␣␣␣va␣?
Test.
```

Dans tous ces tests, on peut constater qu'avec des tailles diverses, les fonctions fonctionnent, en respectant les règles de typographie de la langue française (les espaces insécables ne sont pas, comme leur nom l'indique, séparés du mot précédent ...).

Testons maintenant la vérification orthographique :

```
$>./prog -o
Saisir votre texte :
Ceci est un exmple de texte dans ce projets.
^D
      ----------- Verfication orthographique -----------

4 eme mot invalide !
```

```
$>./prog -o
Saisir votre texte :
Ceci est un exemple de texte dans ce projet.
^D
      ----------- Verfication orthographique -----------

Texte correctement orthographie.
```

Ici, tous ces mots ont été entrés dans le dictionnaire. Ainsi le mot "exemple" existe mais pas "exmple". Le programme indique donc la position du mot incorrect.

```
$>cat exemple.txt | ./prog -t -jg 80 -s
Saisir votre texte :

      ----------- Justification Gauche -----------

Ceci est un exemple de texte a traiter dans ce projet. Nous sommes toujours dans
le meme paragraphe.
Ici on, change de paragraphe. Si ceci fonctionne je dis bravo !
Bravo !
Avec cesure : consonne voyelle. Bonjour ca va ?
Test.



      ----------- Statistiques -----------

Nb caracteres : 187
Nb mots: 38
Nb paragraphes: 5
Nb lignes: 6
```

Ici, on voit que la fonctions de statistiques fonctionne bien.

6.2 Les "bugs"

Lors de l'élaboration d'un programme ,il n'est pas évident de considérer l'ensemble des environnements d'exécution dans lequel il va se trouver.
C'est d'ailleurs pourquoi, les entreprises de développement mettent à disposition de leurs clients un ensemble de patchs correctifs.
Ceux-ci permettent en général de corriger un problème qu'un utilisateur a rencontré sur un logiciel en lui donnant certains paramètres.
Ce sont ces paramètres que l'utilisateur va reporter à l'entreprise via, généralement, des "issue tracker" (Bug-Zilla ...).
Ainsi, dans notre projet, nous avons trouvé quelques "bugs" après la remise des sources, lors des jeux d'essais.

La première concerne la césure. En effet, si on n'entre qu'un seul mot une seule césure est possible, elle n'est pas faite. Ce bug vient du fait que dans la fonction `texte__justifier()` on regarde quand le nombre de caractères dépasse la taille et on sauve des informations de justifications dans les champs de justifications de l'élement précédant (puisque c'est la taille de l'élément courant qui fait dépasser la taille de la ligne).

Ainsi, pour que cette fonctionnalité ait un comportement correcte, il faut avoir un texte au moins de deux éléments.
D'un autre côté, ce n'est pas vraiment un bug au sens où un mot seul n'est pas un texte donc le programme n'est pas censé le traiter.

```
$>./prog -jgc 7
Saisir votre texte :
consonne
^D
      ----------- Justification Gauche avec cesure -----------

consonne
```

Le deuxième bug concerne la fonction `texte__nb_ligne()`. En effet, lors de la justification le nombres d'espaces, de sauts de lignes ... sont modifiés mais pas le nombre de caractères total. Ainsi le nombre de lignes est faussé.

```
$>cat exemple.txt | ./prog -t -jg 20 -s
Saisir votre texte :

      ----------- Justification Gauche -----------

Ceci est un exemple
de texte a traiter
dans ce projet. Nous
sommes toujours dans
le meme paragraphe.
Ici on, change de
paragraphe. Si ceci
fonctionne je dis
bravo !
Bravo !
Avec cesure :
consonne voyelle.
Bonjour ca va ?
Test.



      ----------- Statistiques -----------

Nb caracteres : 187
Nb mots: 38
Nb paragraphes: 5
Nb lignes: 12
```

Le troisième et dernier bug concerne la justification. En effet, en ajoutant le fait de programmer une fonction de justification qui suit les règles de typographie française a entrainé certains problémes sur un texte mal typographié, comme dans l'exemple ci-dessous.

```
$>cat exemple.txt | ./prog -jg 20
```

```
Saisir votre texte :

      ----------- Justification Gauche -----------

ceci est un exemple
de texte a traiter
dans ce projet .nous
sommes toujours dans
le meme paragraphe.
ici on , change de
paragraphe. si ceci
fonctionne je dis
bravo !
bravo !
avec cesure:
consonne voyelle.bonjour
ca va ?
; ? Test.
```

Chapitre 7

Remarques générales sur le projet

7.1 Comment utiliser le programme ?

Dans le shell, on écrit la commande :
`$>./prog [-a] [-o] [-t] [-s] [-jg nb_colonnes] [-jgc nb_colonnes] [-jd nb_colonnes] [-jdc nb_colonne`
`[-jt nb_colonnes] [-jtc nb_colonnes]`
Les options, traitées séquentiellement, sont les suivantes :

- `-a` : affiche le texte tel qu'il est au moment de l'appel (par exemple si c'est la première option le texte est affiché comme il a été saisi),
- `-t` : vérifie la typographie du texte,
- `-o` : vérifie l'orthographe du texte,
- `-jg` : justifie le texte à gauche,
- `-jd` : justifie le texte à droite,
- `-jt` : justifie le texte totalement,
- `-jgc` : justifie à gauche avec cesure,
- `-jdc` :justifie à droite avec cesure,
- `-jtc` :justifie totalement avec cesure.

Les affichages sont fait sur la sortie standard (stdout). Ceci permet de rediriger celle-ci vers un fichier de manière à enregistrer le texte traité.
Cette action peut se faire avec la commande suivante :
`$>cat entree.txt | ./prog [-option] > sortie.txt`.

Le programme lit le texte sur l'entrée standard (stdin). Il est donc possible de l'exécuter via la commande : `$>./prog` puis saisie du texte, ou via la commande `$>cat fichier.txt | ./prog`.

7.2 Le travail réalisé

Le travail réalisé est un programme qui prend en compte toutes les spécifications attendues par le sujet. Le programme a été testé pour un texte de moyenne (cf. 6) et de grande taille.
Nous n'avons trouvé que trois bugs (cf. 6.2) qui, pour deux d'entre eux, n'en sont pas en eux-même.

Chapitre 8

Conclusion

Le projet qui nous a été donné nous a permis de découvrir le travail en équipe et les difficultés qui en découlent.
En effet une des premières difficultés est la communication : faire comprendre ses idées aux autres et les convaincre que c'est la meilleure.
Cette difficulté n'est pas observable lorsqu'on mène un projet tout seul, car il est difficile d'être en désaccord avec soi-même.
Un second obstacle est l'échange des projets.
En effet, même si le rapport intermédiaire est bien fait, se plonger dans le code du projet et se mettre dans son ambiance, est un exercice qui n'est pas évident.
Un autre obstacle est le respect du facteur temps.
En effet, respecter les délais de préparation ou achever une mission pendant une durée plus ou moins courte est une tâche à ne pas négliger par une entreprise pour ne pas perdre ses clients.
Ainsi, pour conclure, nous pouvons dire que ce qui a été avant tout mis en avant dans le projet est la démarche de l'ingénieur dans une entreprise : convaincre, communiquer et motiver le personnel pour avancer.

Table des figures

Chapitre 9

Annexe 1 : Schémas

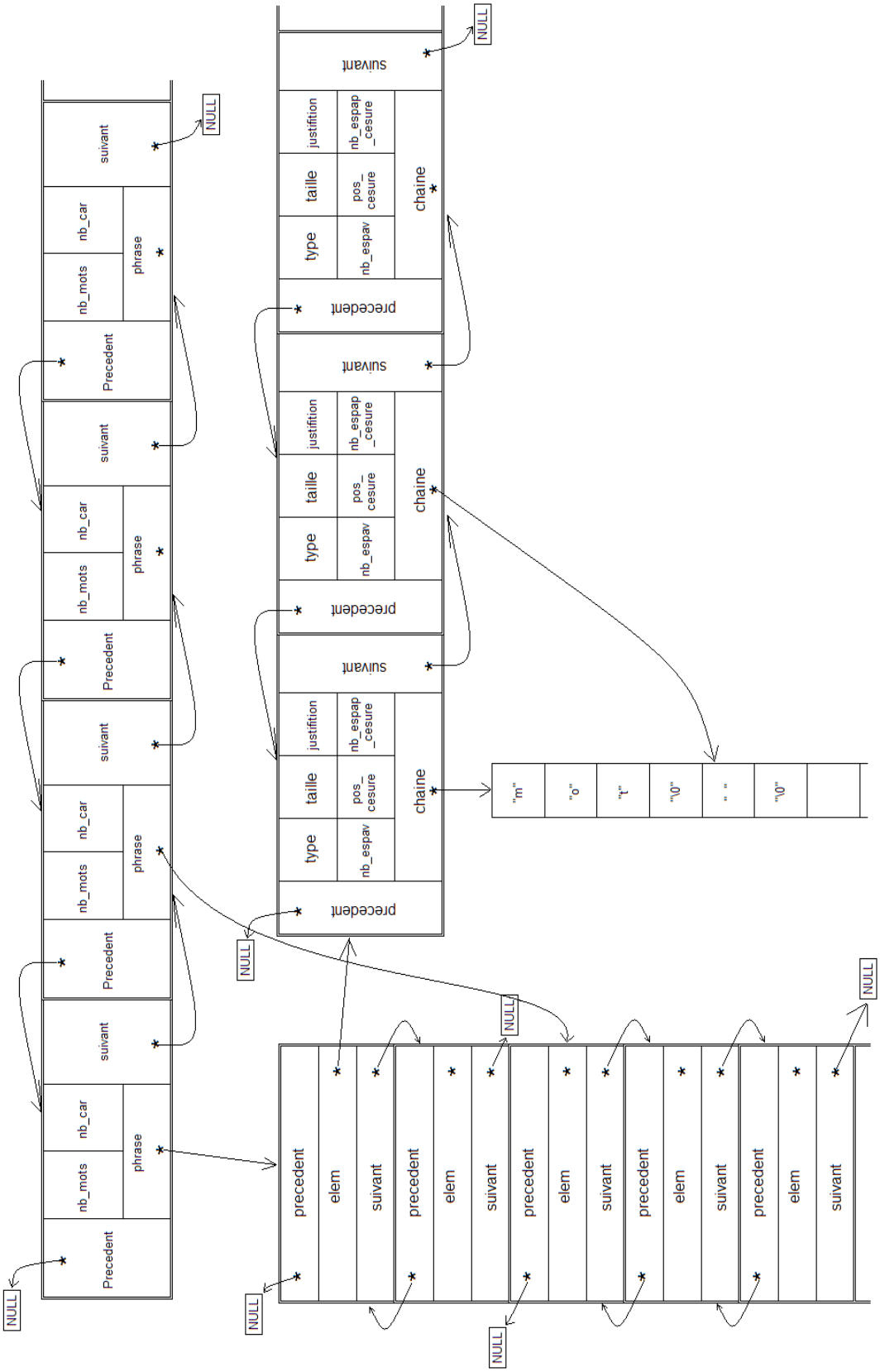

FIG. 9.1 – Utilisation du char comme tableau de booléen

Chapitre 10

Annexe 2 : Code source

```c
#include"fonctions.h"
#include"fonctions_traitement.h"
#include"fonctions_statistiques.h"
#include<fcntl.h>
#include<unistd.h>

int main(int argc, char * argv[])
 {
   texte t;
   char * dico[26][MAX_DICO];
   char chaine[TAILLE_CHAINE]; /* tableau de chaînes de caractères */
   struct element_phrase element[TAILLE_ELEMENT]; /* tableau de structures element_phrase */
   struct phrase phrase[TAILLE_PHRASE]; /* tableau de structures phrase */
   struct paragraphe paragraphe[TAILLE_PARA]; /* tableau de structures paragraphe */
   int col; /* le nombre de colonnes de la console */
   int indexelement;
   int indexphrase;
   int indexparagraphe;
   int ortho;
   int i, j;

   /* initialisation du dictionnaire de mots */
   for(i = 0; i < 26; i ++)
     {
       for(j = 0; j < MAX_DICO; j ++)
         {
           dico[i][j] = " ";
         }
     }

   dico['A' - 65][0] = "a";
   dico['A' - 65][1] = "avec";
   dico['B' - 65][0] = "bravo";
   dico['C' - 65][0] = "ce";
   dico['C' - 65][1] = "ceci";
   dico['C' - 65][2] = "change";
   dico['C' - 65][3] = "cesure";
   dico['C' - 65][4] = "consonne";
   dico['D' - 65][0] = "dans";
   dico['D' - 65][1] = "de";
   dico['D' - 65][2] = "dis";
   dico['E' - 65][0] = "est";
   dico['E' - 65][1] = "exemple";
   dico['F' - 65][0] = "fonctionne";
   dico['I' - 65][0] = "ici";
   dico['J' - 65][0] = "je";
   dico['L' - 65][0] = "le";
   dico['M' - 65][0] = "meme";
   dico['N' - 65][0] = "nous";
   dico['O' - 65][0] = "on";
   dico['P' - 65][0] = "projet";
   dico['P' - 65][1] = "paragraphe";
   dico['S' - 65][0] = "sommes";
   dico['S' - 65][1] = "si";
   dico['T' - 65][0] = "traiter";
   dico['T' - 65][1] = "texte";
   dico['T' - 65][2] = "toujours";
   dico['U' - 65][0] = "un";
   dico['V' - 65][0] = "voyelle";

   col = 72;
   ortho = -1;
   indexelement = 0;
   indexphrase = 0;
   indexparagraphe = 0;
   i = 1;

   /* Initialisations */
```



```c
   init__para(paragraphe, TAILLE_PARA);
   init__phrase(phrase, TAILLE_PHRASE);
   init__element(element, TAILLE_ELEMENT);

   /* lecture du texte sur l'entrée standard */
   t = texte__lire(chaine, element, phrase, paragraphe, TAILLE_PARA, TAILLE_PHRASE, TAILLE_ELEMENT, TAILLE_CHAINE, &indexelement, &indexphrase, &indexparagraphe);

   /* Tant qu'il y a des arguments */
   while(i < argc)
     {
       /*justification gauche */
       if(strncmp("-jg", argv[i], 3) == 0)
         {
           /* on verifie qu'il y est bien le nombre de colonnes passe en argument */
           if(i + 1 >= argc)
             {
               fprintf(stderr, "Usage: %s -jg nombre_de_colonnes\n", argv[0]);
             }
           else
             {
               i ++;
               col = atoi(argv[i]);

               /* on verifie que le nombre de colonnes ne soit ni une lettre ni inferieur ou egal a 0 */
               if(col > 0)
                 {
                   /* reinitialisation des informations de justifications */
                   reinit__justification(t);
                   /* avec cesure */
                   if(strcmp("-jgc", argv[i - 1]) == 0)
                     {
                       printf("\n ———————— Justification Gauche avec cesure ————————\n\n");
                       if(!texte__justifier(t, col, 'G', 1))
                         {
                           printf("La justification n'a pu etre effectuee correctement !\n");
                         }
                     }
                   else /* sans cesure */
                     {
                       printf("\n ———————— Justification Gauche ————————\n\n");
                       if(!texte__justifier(t, col, 'G', 0))
                         {
                           printf("La justification gauche n'a pu etre effectuee correctement !\n");
                         }
                     }

                   /* affichage du texte et reinitialisation des informations de justifications */
                   texte__afficher(t);
                 }
               else
                 {
                   fprintf(stderr, "Justification gauche: le nombre de colonnes doit etre strictement positif !\n");
                 }
             }
         }

       /*justification droite */
       if(strncmp("-jd", argv[i], 3) == 0)
         {
           /* on verifie qu'il y est bien le nombre de colonnes passe en argument */
           if(i + 1 >= argc)
             {
               fprintf(stderr, "Usage: %s -jg nombre_de_colonnes\n", argv[0]);
             }
           else
```

```
            {
              i ++;
              col = atoi(argv[i]);

              /* on verifie que le nombre de colonnes ne soit ni une lettre ni inferieu
r ou egal a 0 */
              if(col > 0)
                {
                  /* reinitialisation des informations de justifications */
                  reinit__justification(t);
                  /* avec cesure */
                  if(strcmp("-jdc", argv[i - 1]) == 0)
                    {
                      printf("\n    ---------- Justification Droite avec cesure ----------\n\n");
                      if(!texte__justifier(t, col, 'D', 1))
                        {
                          printf("La justification n'a pu etre effectuee correctement !\n");
                        }
                    }
                  else /* sans cesure */
                    {
                      printf("\n    ---------- Justification Droite ----------\n\n");
                      if(!texte__justifier(t, col, 'D', 0))
                        {
                          printf("La justification droite n'a pu etre effectuee correctement !\n");
                        }
                    }

                  /* affichage du texte et reinitialisation des informations de justifi
cations */
                  texte__afficher(t);
                }
              else
                {
                  fprintf(stderr, "Justification droite: le nombre de colonnes doit etre strictement positif !\n")
;
                }
            }
        }

      /*justification totale */
      if(strncmp("-jt", argv[i], 3) == 0)
        {
          /* on verifie qu'il y est bien le nombre de colonnes passe en argument */
          if(i + 1 >= argc)
            {
              fprintf(stderr, "Usage: %s -jg nombre_de_colonnes\n", argv[0]);
            }
          else
            {
              i ++;
              col = atoi(argv[i]);

              /* on verifie que le nombre de colonnes ne soit ni une lettre ni inferieu
r ou egal a 0 */
              if(col > 0)
                {
                  /* reinitialisation des informations de justifications */
                  reinit__justification(t);
                  /* avec cesure */
                  if(strcmp("-jtc", argv[i - 1]) == 0)
                    {
                      printf("\n    ---------- Justification Totale avec cesure ----------\n\n");
                      if(!texte__justifier(t, col, 'T', 1))
                        {
                          printf("La justification n'a pu etre effectuee correctement !\n");
                        }
                    }
                  else /* sans cesure */
```



```
                    {
                      printf("\n    ---------- Justification Totale ----------\n\n");
                      if(!texte__justifier(t, col, 'T', 0))
                        {
                          printf("La justification totale n'a pu etre effectuee correctement !\n");
                        }
                    }

                  /* affichage du texte */
                  texte__afficher(t);
                  reinit__justification(t);
                }
              else
                {
                  fprintf(stderr, "Justification totale: le nombre de colonnes doit etre strictement positif !\n");
                }
            }
        }

      /* statistiques */
      if(strcmp("-s", argv[i]) == 0)
        {
          printf("\n    ---------- Statistiques ----------\n\nNb caracteres : %d\nNb mots: %d\nNb paragraph
es: %d\nNb lignes: %d\n", texte__nb_car(t), texte__nb_mots(t), texte__nb_par(t), texte__nb_lig
nes(t, col));
        }

      /* verification orthographique */
      if(strcmp("-o", argv[i]) == 0)
        {
          printf("\n    ---------- Verfication orthographique ----------\n\n");
          ortho = texte__verif_orthographe(t, dico, MAX_DICO);
          if(ortho != -1)
            {
              printf("%d eme mot invalide !\n", ortho);
            }
          else
            {
              printf("Texte correctement orthographie.\n");
            }
        }

      /* verification typographique */
      if(strcmp("-t", argv[i]) == 0)
        {
          printf("\n    ---------- Verification typographique ----------\n\n");
          if(!texte__typographie(&t,chaine, element, phrase, paragraphe, TAILLE_PARA, T
AILLE_PHRASE, TAILLE_ELEMENT, TAILLE_CHAINE, &indexelement, &indexphrase, &indexparagrap
he))
            {
              printf("Le texte n'est pas correctement typographie !\n\n");
            }
          else
            {
              printf("La verification typographique a ete realisee avec succes !\n\n");
            }
          texte__afficher(t);
        }

      /* simple affichage du texte */
      if(strcmp("-a", argv[i]) == 0)
        {
          printf("\n    ---------- Texte entre ----------\n\n");
          texte__afficher(t);
        }
      printf("\n\n");
      i ++;
    }
```

```
  return 1;
}
```

```
#ifndef STRUCTURES_H
#define STRUCTURES_H

#define TAILLE_CHAINE 5000
#define TAILLE_PARA 200
#define TAILLE_PHRASE 500
#define TAILLE_ELEMENT 3000
#define MAX_DICO 10

/* Définition des structures */
struct element_phrase                /* l'élément de phrase est caractérisé par son type
 (E,M,P), les caractÃ¨res de l'élément, sa taille */
{
  char type;                         /* type de l'élement: E,M,P,N */
  int taille;                        /* la taille de l'élément */
  char * chaine;                     /* l'élément Ã  stocker */
  char justification;                /* les informations de justification */
  int nb_espav;                      /* le nombre d'espaces avant */
  int pos_cesure;                    /* la position de la cesure */
  int nb_espap_cesure;               /* le nombre d'espaces apres la cesure */
  struct element_phrase * suivant;   /* pointeur sur l'element suivant */
  struct element_phrase * precedent; /* pointeur sur l'élement précédent */
};

struct phrase                        /* la phrase est une liste double chaînée d'élément
s de phrases */
{
  struct element_phrase * elem;      /* élement de phrase */
  struct phrase * suivant;           /* pointeur sur le paragraphe suivant */
  struct phrase * precedent;         /* pointeur sur le paragraphe précédent */
};

struct paragraphe                    /* le paragraphe est une liste double chaînée de ph
rases */
{
  int nb_mots;                       /* le nombre de mots dans le paragraphe */
  int nb_car;                        /* le nombre de caractères dans le paragraphe */
  struct phrase * phrase;            /* phrase */
  struct paragraphe * suivant;       /* pointeur sur la phrase suivante */
  struct paragraphe * precedent;     /* pointeur sur la phrase précédente */
};


typedef struct paragraphe * texte;   /* le texte est une liste double chaînée de paragra
phes */

#endif
```

```c
#include "fonctions.h"

void /* on ne retourne rien*/
init__para(
           struct paragraphe * paragraphe, /* le tableau qui contient les structures par
agraphe  */
           int taille /* la taille du tableau */
           )
{
  int i;

  for(i = 0; i < taille; i ++)
    {
      paragraphe[i].suivant = NULL;
      paragraphe[i].precedent = NULL;
      paragraphe[i].phrase = NULL;
      paragraphe[i].nb_mots = 0;
      paragraphe[i].nb_car = 0;
    }
}

void /* on ne retourne rien*/
init__phrase(
           struct phrase * phrase, /* le tableau qui contient les structures phrase */
           int taille /* la taille du tableau */
           )
{
  int i;

  for(i = 0; i < taille; i ++)
    {
      phrase[i].suivant = NULL;
      phrase[i].precedent = NULL;
      phrase[i].elem = NULL;
    }
}

void /* on ne retourne rien*/
init__element(
           struct element_phrase * element, /* le tableau qui contient les structures el
ement_phrase */
           int taille /* la taille du tableau */
           )
{
  int i;

  for(i = 0; i < taille; i ++)
    {
      element[i].suivant = NULL;
      element[i].precedent = NULL;
      element[i].chaine = "Chaine vide.";
      element[i].type = '\0';
      element[i].taille = 0;
      element[i].justification = 0;
      element[i].nb_espav = 0;
      element[i].pos_cesure = 0;
      element[i].nb_espap_cesure = 0;
    }
}

void /* on ne retourne rien*/
reinit__justification(
                      texte t /* pointeur sur la structure texte */)
{
 /* Déclaration */
  struct paragraphe * pa; /* paragaphe */
  struct phrase * ph; /* la phrase */
  struct element_phrase * e; /* l'élement de paragraphe */
```



```c
  /* Initialisation  */
  pa = t;
  ph = pa->phrase;
  e = ph->elem;

  /* Tant qu'on a des paragraphes */
  while(pa != NULL)
    {
      ph = pa->phrase;
      /* Tant qu'on a des phrases */
      while(ph != NULL)
        {
          e = ph->elem;
          /* Tant qu'on a des mots, espaces ou ponctuation, on les affiche */
          while(e != NULL)
            {
              e->nb_espav = 0;
              e->pos_cesure = 0;
              e->justification = 0;
              e = e->suivant;
            }
          ph = ph->suivant;
        }
      pa = pa->suivant;
    }
}

texte /* on retourne un pointeur sur la liste de paragraphes */
texte__lire(
           char * chaine, /* le tableau qui contient les chaînes de caractères  */
           struct element_phrase * element, /* le tableau qui contient les structures e
lement_phrase */
           struct phrase * phrase, /* le tableau qui contient les structures phrase */
           struct paragraphe * paragraphe, /* le tableau qui contient les structures pa
ragraphe  */
           int taille_para, /* la taille du tableau de structures paragraphe */
           int taille_phrase, /* la taille du tableau de structures phrase */
           int taille_element, /* la taille du tableau de structures element_phrase */
           int taille_chaine, /* la taille du tableau des chaînes de caractères */
           int * indexelement, /* l'index de l'élément */
           int * indexphrase, /* l'index de la phrase */
           int * indexparagraphe /* l'index du paragraphe */
)
{
  /* Déclaration */
  int creer_element, creer_phrase, creer_para; /* à vrai si on doit créer l'élément (pas
 au sens element_pharse)  */
  int taillechaine; /* la taille de la chaine */
  int index; /* l'index du début de la chaîne */
  int car; /* caractère lu par fgetc */
  char last; /* type du dernier caractère lu */
  char current; /* type du carcactère courant */

  /* Initialisation */
  taillechaine = 0;
  index = 0;
  *indexelement = 0;
  *indexphrase = 0;
  *indexparagraphe = 0;
  creer_element = 0;
  creer_phrase = 0;
  creer_para = 0;

  last = current = ' ';
  paragraphe[*indexparagraphe].phrase = & phrase[*indexphrase];
  phrase[*indexphrase].elem = & element[*indexelement];

  printf("Saisir votre texte : \n");
```

```
  /* On attend tous les caractères sur l'entrée standard */
  do
    {

      /* création des différents éléments */
      if(creer_element == 1)
        {
          element[*indexelement].suivant = & element[*indexelement + 1];
          (*indexelement) ++;
          element[*indexelement].precedent = & element[*indexelement -1];

          creer_element = 0;
        }
       if(creer_phrase == 1)
        {
          phrase[*indexphrase].suivant = & phrase[*indexphrase + 1];
          (*indexphrase) ++;
          phrase[*indexphrase].precedent = & phrase[*indexphrase - 1];

          (*indexelement) ++;
          phrase[*indexphrase].elem = & element[*indexelement];

          creer_phrase = 0;
        }
      if(creer_para == 1)
        {
          paragraphe[*indexparagraphe].suivant = & paragraphe[*indexparagraphe + 1];
          (*indexparagraphe) ++;
          paragraphe[*indexparagraphe].precedent = & paragraphe[*indexparagraphe - 1];

          (*indexphrase) ++;
          paragraphe[*indexparagraphe].phrase = & phrase[*indexphrase];

          (*indexelement) ++;
          phrase[*indexphrase].elem = & element[*indexelement];

          creer_para = 0;
        }

      /* on lit le caractère sur l'entrée standard */
      car = fgetc(stdin);

      /* si c'est un espace ou une tabulation: type E */
      if(car == ' ' || car == '\t')
        {
          current = 'E';
        }
      else /* sinon type M ou P */
        {
          /* si c'est une ponctuation */
          if((car >= '!' && car <= '/') || (car >= ':' && car <= '?')  || (car >= '[' && c
ar <= '`')  || (car >= '{' && car <= '~') )  /*tout les poncuations possibles*/
            {

              current = 'P';
            }
          else /* sinon c'est un mot */
            {
              current = 'M';
            }
        }

      /* s'il y a une nouvelle ligne */
      if(car == '\n')
        {
          current = 'N';
        }

      /* si le type du caractère courant est différent de celui du caractère précédent o
```



```
u que c'est une ponctuation alors c'est un nouveau type */
      if((current != last && last != ' ') || last == 'P')
        {
          chaine[index + taillechaine] = '\0'; /* fin de la chaîne */
          if (last == 'E') /* si on a un plusieurs espaces on ne garde qu'un */
            {
              taillechaine = 1;
              chaine[index] = ' ';
              chaine[index + taillechaine] = '\0';
            }

          element[*indexelement].chaine = & chaine[index]; /* le début de la chaîne est
l'adresse de la index-ieme case  */
          element[*indexelement].type = last; /* du du dernier élément créé, pas le cour
ant  */
          element[*indexelement].taille = taillechaine; /* on met la taille de la chaîne
 dans la structure */
          index += taillechaine + 1; /* index du premier caractère de la chaîne suivante
 */

          /* on incrémente le nombre de mots dans le paragraphe si c'est un mot qu'on aj
oute */
          if(last == 'M')
            {
              paragraphe[*indexparagraphe].nb_mots ++;
            }

          /* on ajoute le nombre de caractère créés à celui déjà présent dans la structu
re */
          paragraphe[*indexparagraphe].nb_car += taillechaine;

          /* si c'était un espacement ou un saut de ligne on ne le compte pas */
          if(last == 'E' || last == 'N')
            {
              paragraphe[*indexparagraphe].nb_car -= taillechaine;
            }

          /* on crée un nouvel élément */
          creer_element = 1;

          taillechaine = 0;
        }

      /* si l'élément précédent était un point et qu'il existe un élément précédent, on
ajoute une phrase */
      if(element[*indexelement].precedent != NULL && strcmp(element[*indexelement].chain
e, ".") == 0)
        {
          /* on crée une nouvelle phrase et donc pas un élément, ni un paragraphe */
          creer_phrase = 1;
          creer_element = 0;
          creer_para = 0;
        }

      /* si l'élément précédent est un \n un on crée un nouveau paragraphe */
      if(last == 'N')
        {
          /* on crée un nouveau paragraphe et donc pas de nouvelle phrase, ni d'élément
*/
          creer_para = 1;
          creer_phrase = 0;
          creer_element = 0;
        }

      /* la chaine temporaire reçoit le caractère */
      chaine[index + taillechaine] = car;

      /* on incrémente la taille de la chaine */
      taillechaine ++;
```

```
      /* le dernier prend la valeur du courant */
      last = current;
    }
  while(car != EOF && (*indexparagraphe < taille_para) && (*indexphrase < taille_phrase)
 && (*indexelement < taille_element) && (index + taillechaine + 1 < taille_chaine)); /*
tant qu'on n'a pas atteint la fin du texte et que les limites en taille des tableaux ne
sont pas dépassées */

  if(index + taillechaine + 1 >= taille_chaine)
    {
      chaine[index + taillechaine] = '\0';
      element[*indexelement].chaine = & chaine[index];
    }

  return & paragraphe[0];
}

void /* on ne retourne rien; on affiche seulement le texte */
texte__afficher(
            texte t /* pointeur sur la structure texte */
)
{
  /* Déclaration */
  struct paragraphe * pa; /* paragaphe */
  struct phrase * ph; /* la phrase */
  struct element_phrase * e; /* l'élement de paragraphe */
  int i, j;
  int esp_av;
  int cesure;
  int sautligne;
  int pas_elem;
  int esp_ap_cesure;

  /* Initialisation  */
  i = j = 0;
  esp_av = 0;
  cesure = 0;
  sautligne = 0;
  pas_elem = 0;
  esp_ap_cesure = 0;

  pa = t;
  ph = pa->phrase;
  e = ph->elem;

  /* Tant qu'on a des paragraphes */
  while(pa != NULL)
    {
      ph = pa->phrase;
      /* Tant qu'on a des phrases */
      while(ph != NULL)
        {
          e = ph->elem;
          /* Tant qu'on a des mots, espaces ou ponctuation, on les affiche */
          while(e != NULL)
            {
              sautligne = e->justification & 0x01;
              cesure = (e->justification & 0x02) >> 1;
              esp_av = (e->justification & 0x04) >> 2;
              pas_elem = (e->justification & 0x08) >> 3;
              esp_ap_cesure = (e->justification & 0x10) >> 4;

              /* s'il y a des espaces avant l'element on en affiche autant que besoin */
              if(esp_av && !cesure)
                {
                  for(i = 0; i < e->nb_espav; i ++)
                    {
                      printf(" ");
```



```
                    }
                }

              /* s'il y a une cesure a faire */
              if(cesure)
                {
                  for(i = 0; i < e->taille; i ++)
                    {
                      printf("%c", e->chaine[i]);

                      /* on affiche un tiret et on revient a la ligne si le compteur vau
t la position de la cesure */
                      if(i == e->pos_cesure)
                        {
                          printf("-\n");

                          /* s'il y a des espaces apres la cesure on en affiche autant q
ue besoin */
                          if(esp_ap_cesure)
                            {
                              for(j = 0; j < e->nb_espap_cesure; j ++)
                                {
                                  printf(" ");
                                }
                            }
                        }
                    }
                }
              else /* affichage normal de la chaine */
                {
                  /* si ce n'est pas un espace a ne pas afficher, on affiche la chaine *
/
                  if(!pas_elem)
                    {
                      printf("%s", e->chaine);
                    }
                }

              /* il y a un saut de ligne apres l'element */
              if(sautligne)
                {
                  printf("\n");
                }

              e = e->suivant;
            }
          ph = ph->suivant;
        }
      pa = pa->suivant;
    }
}
```

```
#ifndef FONCTIONS_H
#define FONCTIONS_H

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include "structures.h"

/* DÃ©finition des prototypes de fonctions */
void init__para(struct paragraphe * paragraphe, int taille);
void init__phrase(struct phrase * phrase, int taille);
void init__element(struct element_phrase * element_phrase, int taille);
texte texte__lire(char * chaine, struct element_phrase * element, struct phrase * phrase
, struct paragraphe * paragraphe,  int taille_para, int taille_phrase, int taille_elemen
t, int taille_chaine, int * indexelement, int * indexphrase, int * indexparagraphe);
void texte__afficher(texte t);
void reinit__justification(texte t);

#endif
```

```c
#include "fonctions_statistiques.h"

int /* on renvoit le nombre de caractÃ"res de type entier  */
texte__nb_car(
              texte t /* la structure texte, remplie lors de la saisie */
)
{
  /* DÃ©claration */
  int nbcar; /* le nombre de caracteres */
  struct paragraphe * pa; /* paragaphe */

  /* Initialisation  */
  pa = t;
  nbcar = 0;

  /* Tant qu'on a des paragraphes */
  while(pa != NULL)
    {
      nbcar += pa->nb_car; /* nbcar <- nbcar + taille de l'Ã©lÃ©ment  */
      pa = pa->suivant;
    }

  return nbcar;
}

int /* on renvoit le nombre de mots de type entier  */
texte__nb_mots(
              texte t /* la structure texte, remplie lors de la saisie */
)
{
  /* DÃ©claration */
  int nbmots; /* le nombre de mots */
  struct paragraphe * pa; /* paragaphe */

  /* Initialisation  */
  pa = t;
  nbmots = 0;

  /* Tant qu'on a des paragraphes */
  while(pa != NULL)
    {
      nbmots += pa->nb_mots; /* nbmots = nbmots + nombre de mots dans le paragraphe cour
ant */
      pa = pa->suivant;
    }

  return nbmots;
}

int /* on renvoit le nombre de mots de type entier  */
texte__nb_par(
              texte t /* la structure texte, remplie lors de la saisie */
)
{
  /* DÃ©claration */
  int nbpar; /* le nombre de paragraphes */
  struct paragraphe * pa; /* paragaphe */

  /* Initialisation  */
  pa = t;
  nbpar = 0;

  /* Tant qu'on a des paragraphes */
  while(pa != NULL)
    {
      nbpar ++;
      pa = pa->suivant;
    }
```



```c
  return nbpar;
}

int /* on renvoit le nombre de lignes de type entier  */
texte__nb_lignes(
              texte t, /* la structure texte, remplie lors de la saisie */
              int nb_colonnes_console /* le nombre de colonnes de la console */
)
{
  /* DÃ©claration */
  struct paragraphe * pa; /* paragaphe */
  struct phrase * ph; /* la phrase */
  struct element_phrase * e; /* l'élement de paragraphe */
  int nblignes; /* le nombre de lignes complÃ"tes */
  int reste; /* pour le reste de la division euclidienne */
  int quotient; /* pour le quotient de la division euclidienne */
  int esp; /* le nombre d'espaces avant */

  /* Initialisation */
  esp = 0;
  nblignes = 0;
  pa = t;
  ph = pa->phrase;
  e = ph->elem;

  /* Tant qu'on a des paragraphes */
  while(pa != NULL)
    {
      esp = 0;
      ph = pa->phrase;
      /* Tant qu'on a des phrases */
      while(ph != NULL)
        {
          e = ph->elem;
          /* Tant qu'on a des mots, espaces ou ponctuation, on les affiche */
          while(e != NULL)
            {
              /* si le nombre d'espace est positif on l'ajoute au resultat precedent */
              if(e->nb_espav > 0)
                {
                  esp += e->nb_espav;
                }
              e = e->suivant;
            }
          ph = ph->suivant;
        }
      reste = (pa->nb_car + esp) % nb_colonnes_console; /* reste de la division euclidie
nne de nb_colonnes_console par nb_car */
      quotient = (pa->nb_car + esp - ((pa->nb_car + esp) % nb_colonnes_console))/nb_colo
nnes_console; /* le quotient: le nombre de lignes */

      nblignes += quotient; /* nblignes = nblignes + quotient  */

      /* s'il y a un rÃ©sidu sur une autre ligne, on incrÃ©mente le nombre de lignes */
      if (reste != 0)
        {
          nblignes ++;
        }
      pa = pa->suivant;
    }

  return nblignes;
}
```

```c
#include "fonctions_statistiques.h"

int /* on renvoit le nombre de caractÃ"res de type entier  */
texte__nb_car(
              texte t /* la structure texte, remplie lors de la saisie */
)
{
  /* DÃ©claration */
  int nbcar; /* le nombre de caracteres */
  struct paragraphe * pa; /* paragaphe */

  /* Initialisation  */
  pa = t;
  nbcar = 0;

  /* Tant qu'on a des paragraphes */
  while(pa != NULL)
    {
      nbcar += pa->nb_car; /* nbcar <- nbcar + taille de l'Ã©lÃ©ment  */
      pa = pa->suivant;
    }

  return nbcar;
}

int /* on renvoit le nombre de mots de type entier  */
texte__nb_mots(
              texte t /* la structure texte, remplie lors de la saisie */
)
{
  /* DÃ©claration */
  int nbmots; /* le nombre de mots */
  struct paragraphe * pa; /* paragaphe */

  /* Initialisation  */
  pa = t;
  nbmots = 0;

  /* Tant qu'on a des paragraphes */
  while(pa != NULL)
    {
      nbmots += pa->nb_mots; /* nbmots = nbmots + nombre de mots dans le paragraphe cour
ant */
      pa = pa->suivant;
    }

  return nbmots;
}

int /* on renvoit le nombre de mots de type entier  */
texte__nb_par(
              texte t /* la structure texte, remplie lors de la saisie */
)
{
  /* DÃ©claration */
  int nbpar; /* le nombre de paragraphes */
  struct paragraphe * pa; /* paragaphe */

  /* Initialisation  */
  pa = t;
  nbpar = 0;

  /* Tant qu'on a des paragraphes */
  while(pa != NULL)
    {
      nbpar ++;
      pa = pa->suivant;
    }
```



```c
  return nbpar;
}

int /* on renvoit le nombre de lignes de type entier  */
texte__nb_lignes(
              texte t, /* la structure texte, remplie lors de la saisie */
              int nb_colonnes_console /* le nombre de colonnes de la console */
)
{
  /* DÃ©claration */
  struct paragraphe * pa; /* paragaphe */
  struct phrase * ph; /* la phrase */
  struct element_phrase * e; /* l'élement de paragraphe */
  int nblignes; /* le nombre de lignes complÃ"tes */
  int reste; /* pour le reste de la division euclidienne */
  int quotient; /* pour le quotient de la division euclidienne */
  int esp; /* le nombre d'espaces avant */

  /* Initialisation */
  esp = 0;
  nblignes = 0;
  pa = t;
  ph = pa->phrase;
  e = ph->elem;

  /* Tant qu'on a des paragraphes */
  while(pa != NULL)
    {
      esp = 0;
      ph = pa->phrase;
      /* Tant qu'on a des phrases */
      while(ph != NULL)
        {
          e = ph->elem;
          /* Tant qu'on a des mots, espaces ou ponctuation, on les affiche */
          while(e != NULL)
            {
              /* si le nombre d'espace est positif on l'ajoute au resultat precedent */
              if(e->nb_espav > 0)
                {
                  esp += e->nb_espav;
                }
              e = e->suivant;
            }
          ph = ph->suivant;
        }
      reste = (pa->nb_car + esp) % nb_colonnes_console; /* reste de la division euclidie
nne de nb_colonnes_console par nb_car */
      quotient = (pa->nb_car + esp - ((pa->nb_car + esp) % nb_colonnes_console))/nb_colo
nnes_console; /* le quotient: le nombre de lignes */

      nblignes += quotient; /* nblignes = nblignes + quotient  */

      /* s'il y a un rÃ©sidu sur une autre ligne, on incrÃ©mente le nombre de lignes */
      if (reste != 0)
        {
          nblignes ++;
        }
      pa = pa->suivant;
    }

  return nblignes;
}
```

```c
#include "fonctions_traitement.h"

int texte__justifier(texte t,int l, char mode, char mode_cesure)
{
  /* DÃ©claration */
  struct paragraphe * pa; /* paragaphe */
  struct phrase * ph; /* la phrase */
  struct phrase * ph_tmp; /* la phrase */
  struct element_phrase * e; /* l'Ã©lement de paragraphe */
  struct element_phrase * elem_deb_ligne; /* l'Ã©lement de paragraphe */
  struct element_phrase * elem_tmp; /* l'Ã©lement de paragraphe */
  struct element_phrase * e_tmp; /* l'Ã©lement de paragraphe */
  int cpt_ligne;
  int retour;
  int cesure;
  char c;
  int add;
  int i;
  int insec;
  int insec2;
  int nb_esp;
  int nb_esp_elem;
  int nb_esp_elem_reste;
  int nb_element_ligne;

  /* Initialisation  */
  pa = t;
  ph = pa->phrase;
  e = ph->elem;
  elem_deb_ligne = e;
  cpt_ligne = 0;
  cesure = 0;
  retour = 1;
  add = 0;
  nb_esp = 0;
  nb_esp_elem = 0;
  nb_esp_elem_reste = 0;
  nb_element_ligne = 0;
  i = 0;
  insec = 0;
  insec2 = 0;
  c = ' ';

  /* Tant qu'on a des paragraphes */
  while(pa != NULL)
    {
      cpt_ligne = 0;
      ph = pa->phrase;

      /* Tant qu'on a des phrases */
      while(ph != NULL)
        {
          ph_tmp = ph;
          cesure = 0;
          e = ph->elem;
          /* Tant qu'on a des mots, espaces ou ponctuation, on les affiche */
          while(e != NULL)
            {
              e_tmp = e;
              insec = 0;
              insec2 = 0;
              nb_esp = 0;
              cpt_ligne += e->taille;
              c = ' ';
              i = 1;
              add = 0;
              cesure = 0;

              /* s'il y a un espace insecable apres l'element */
```



```c
              if(e->suivant != NULL && e->suivant->type == 'E' && e->suivant->suivant != NULL && e->suivant->suivant->type == 'P' && (strcmp(e->suivant->suivant->chaine, ":") == 0 || strcmp(e->suivant->suivant->chaine, "!") == 0 || strcmp(e->suivant->suivant->chaine, ";") == 0 || strcmp(e->suivant->suivant->chaine, "?") == 0))
                {
                  insec = 1;
                  add = e->suivant->taille + e->suivant->suivant->taille;
                }

              /* s'il ne doit pas y avoir d'espace apres l'element car il y a une virgule ou un point */
              if(e->suivant != NULL && e->suivant->type == 'P' && (strcmp(e->suivant->chaine, ",") == 0 || strcmp(e->suivant->chaine, ".") == 0))
                {
                  insec2 = 1;
                  add = e->suivant->taille;
                }

              cpt_ligne += add;

              /* si le compteur depasse la taille de la ligne */
              if(cpt_ligne > l)
                {
                  if(mode_cesure == 1)
                    {
                      while(c != '\0' && !cesure)
                        {
                          c = e->chaine[i];

                          /* si le caractere est une cesure (pas une voyelle) */
                          if(((c != 'a' && c != 'e' && c != 'i' && c != 'o' && c != 'u' && c != 'y' && c != 'A' && c != 'E' && c != 'I' && c != 'O' && c != 'U' && c != 'Y')) && ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')))
                            {
                              /* si le prochain caractere existe et que c'est une consonne (et le meme caractere) alors il y a cesure */
                              if(i < e->taille && e->chaine[i] == e->chaine[i + 1] && ((c != 'a' && c != 'e' && c != 'i' && c != 'o' && c != 'u' && c != 'y' &&  c != 'A' && c != 'E' && c != 'I' && c != 'O' && c != 'U' && c != 'Y')) && ((c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z')))
                                {
                                  cesure = 1;
                                }
                            }
                          i ++;
                        }
                    }
                  /* si l'element precedent n'est pas nul */
                  if(e->precedent != NULL)
                    {
                      /* il y a une cesure a faire si cesure vaut vrai et si on a la place sur la ligne */
                      if(cesure && (cpt_ligne - e->taille + i + 1) <= l)
                        {
                          printf("%s %d - %d + 1 <= %d\n", e->chaine,cpt_ligne,i, l);
                          /* s'il y a une cesure on reaffiche l'espace qu'on n'a pas affiche */
                          if(e->precedent != NULL && e->precedent->type == 'E')
                            {
                              e->precedent->justification = 0;
                            }
                          e->justification = 0x02;

                          /* si on est en mode droit, on met le champ espaces apres cesure de justification a vrai */
                          if(mode == 'D' || mode == 'T')
                            {
                              e->justification = 0x12;
                            }
```

```
                        /* position de la cesure = position du debut de la double cons
onne - 1 */
                        e->pos_cesure = i - 1;

                        /* le compteur de ligne vaut desormais le nombres de lettres s
ur la ligne suivante */
                        cpt_ligne = e->taille + add - i;
                      }
                    else /* sinon on traite l'element */
                      {
                        e->precedent->justification = 0x01;
                        cpt_ligne = 0;
                        cpt_ligne = e->taille + add;

                        /* si on tombe sur un espace, enfin de ligne, on ne l'affiche
pas et on affiche un \n */
                        if(e->precedent != NULL && e->precedent->type == 'E')
                          {
                            elem_deb_ligne->nb_espap_cesure ++;
                            elem_deb_ligne->nb_espav ++;
                            e->precedent->justification = 0x09;
                          }
                      }
                  }

                if((!cesure && (e->taille - ((e->taille + add) % l)) / l > 0) || (cesu
re && (i >= l || (e->taille - i - 1 - ((e->taille + add) % l)) / l > 0))) /* l'element e
st trop grand pour la ligne et pas de cesure ou si sa taille moins la position de la ces
ure ne loge pas sur deux lignes on a pas pu justifier correctement */
                  {
                    retour = 0;
                  }
              }

            /* si le compteur est a 1 et que c'est un espace, on ne l'affiche pas et o
n met le compteur a 0 */
            if(cpt_ligne == 1  && e->type == 'E')
              {
                e->justification = 0x08;
                cpt_ligne = 0;
              }

            /* s'il y a un element de type saut de ligne, on augmente le nombre d'espa
ce car il prend un caractere */
            if(e->type == 'N')
              {
                /* s'il y a un \n avant ce \n, on ne l'affiche pas */
                if(e->precedent != NULL && (e->precedent->justification & 0x01) == 0x0
1)
                  {
                    e->precedent->justification &= 0xFE;
                    e->justification = 0x09;
                  }
                elem_deb_ligne->nb_espav ++;
                elem_deb_ligne->nb_espap_cesure ++;
              }

            /* si on justifie a droite ou totale */
            if(mode == 'D' || mode == 'T')
              {
                /* si l'element precedent existe et qu'on saute une ligne pour la just
ification ou que c'est l'element de debut de paragraphe  */
                if((e->precedent != NULL && (e->precedent->justification & 0x01) == 1)
 || e == pa->phrase->elem || (e->justification & 0x12) == 0x12)
                  {
                   /* si c'est une cesure on ajoute a la ligne courante la taille du
mot jusqu'a la cesure  */
                    if((e->justification & 0x12) == 0x12)
```



```
                      {
                        elem_deb_ligne->nb_espav -= i + 1;
                      }

                    /* si l'element de debut de ligne est cachÃ© on rajoute un espace
*/
                    if((elem_deb_ligne->justification & 0x08) == 0x08)
                      {
                        elem_deb_ligne->nb_espav ++;
                      }

                    /* si le mode de justification est total */
                    if(mode == 'T' && nb_element_ligne > 0)
                      {
                        if(nb_element_ligne <= 1)
                          {
                            /* on n'affiche pas les espaces apres la cesure */
                            elem_deb_ligne->justification &= 0xEF;

                            /* s'il n'y a pas de cesure sur l'element */
                            if((elem_deb_ligne->justification & 0x02) != 0x02)
                              {
                                /* on n'affiche pas les espaces apres la cesure */
                                elem_deb_ligne->justification &= 0xEB;
                              }
                            /* on n'affiche pas les espaces apres la cesure ni ceux av
ant l'element */
                            elem_deb_ligne->justification &= 0xEB;
                          }
                        else
                          {
                            nb_element_ligne --;
                            /* on recupere le nombre d'espace et on calcule le nombre
d'espace avant chaque element */
                            if(mode_cesure && (elem_deb_ligne->justification & 0x10) =
= 0x10)
                              {
                                nb_esp = elem_deb_ligne->nb_espap_cesure; /* si on est
 en mode cesure on recupere le nombre d'espace apres la cesure */
                              }
                            else
                              {
                                nb_esp = elem_deb_ligne->nb_espav;
                              }
                            nb_esp_elem = (nb_esp - (nb_esp % nb_element_ligne)) / nb_
element_ligne;
                            nb_esp_elem_reste = nb_esp % nb_element_ligne;
                            elem_tmp = elem_deb_ligne;

                            /* tant qu'on a des elements sur la ligne, qu'on a n'a pas
 atteint la fin et qu'on a encore des espaces a mettre */
                            while(elem_deb_ligne != NULL && elem_deb_ligne != e_tmp &&
 nb_esp >= 0)
                              {
                                /* si on est au premier element */
                                if(elem_deb_ligne == elem_tmp)
                                  {
                                    /* on n'affiche pas les espaces apres la cesure */
                                    elem_deb_ligne->justification &= 0xEF;

                                    /* s'il n'y a pas de cesure sur l'element */
                                    if((elem_deb_ligne->justification & 0x02) != 0x02)
                                      {
                                        /* on n'affiche pas les espaces apres la cesur
e */

                                        elem_deb_ligne->justification &= 0xEB;
                                      }
                                  }
```

```
                                /* si on n'est pqs au premier element */
                                if(elem_deb_ligne != elem_tmp && elem_deb_ligne->type
== 'M')
                                  {
                                    /* on affiche nb_esp_elem espaces avant puis on de
cremente le nombre d'espaces total */
                                    elem_deb_ligne->justification |= 0x04;
                                    elem_deb_ligne->nb_espav = nb_esp_elem;
                                    nb_esp -= nb_esp_elem;


                                    /* si le reste est superieur a 0 (la division du n
ombre d'espace par le nombre d'elements ne tombe pas juste), on en insere un et on le de
cremente */
                                    if(nb_esp_elem_reste > 0)
                                      {
                                        elem_deb_ligne->nb_espav ++;
                                        nb_esp_elem_reste --;
                                      }
                                  }

                                /* on passe a l'element suivant */
                                elem_deb_ligne = elem_deb_ligne->suivant;

                                /* si l'element est nul, fin de la phrase, on passe a
la suivante */
                                if(elem_deb_ligne == NULL)
                                  {
                                    elem_deb_ligne = ph_tmp->elem;
                                  }
                              }
                          }
                      }
                    nb_element_ligne = 0;

                    /* s'il reste un espace a afficher, on le met au dernier element *
/
                    if(elem_deb_ligne != NULL && elem_deb_ligne->precedent != NULL &&
(nb_esp_elem_reste > 0 || nb_esp_elem > 0))
                      {
                        elem_deb_ligne->precedent->nb_espav += nb_esp_elem + nb_esp_el
em_reste;
                      }

                    elem_deb_ligne = e; /* l'element courant devient le premier elemen
t de la ligne suivante */
                    elem_deb_ligne->nb_espav = l; /* on initialise le nombre d'espaces
 a la taille de la ligne */

                    /* si l'element de debut de ligne est un espace */
                    if(e->type == 'E' && e->suivant != NULL)
                      {
                        /* on le cache et on prend le suivant*/
                        e->justification = 0x08;
                        elem_deb_ligne = e->suivant;

                        elem_deb_ligne->nb_espav = l + 1; /* on augmente le nombre d'e
space avant de 1 */
                      }

                    elem_deb_ligne->nb_espap_cesure = l + i; /* on initialise le nombr
e d'espaces a la taille de la ligne + le nombre de lettres apres la cesure */
                    elem_deb_ligne->justification |= 0x04; /* la justification est ini
tialisee avec des espaces avant */
                  }
                elem_deb_ligne->nb_espav -= e->taille + add; /* en enleve au nombre de
'espace de l'element de debut de ligne la taille de l'element courant */
                elem_deb_ligne->nb_espap_cesure -= e->taille + add; /* pareil pour les
 espaces apres cesure: ainsi le nombre d'espace est le meme mais depend du booleen de ju
```



```
stification */
                  }

                /* si l'element n'est pas cache on incremente le nombre d'elements sur la
ligne */
                if(e->type == 'M' && (e->justification & 0x08) != 0x08)
                  {
                    nb_element_ligne ++;
                  }

                /* s'il y a un espace insecable apres l'element */
                if(insec)
                  {
                    e = e->suivant->suivant;
                  }

                /* s'il ne doit pas y avoir d'espace apres l'element car il y a une virgul
e ou un point */
                if(insec2)
                  {
                    e = e->suivant;
                  }

                /* si le pointeur sur l'element n'est pas nul */
                if(e != NULL)
                  {
                    e = e->suivant;
                  }
              }
            ph = ph->suivant;
          }
        pa = pa->suivant;
      }

    /* si le mode de justification est total */
    if(mode == 'T' && nb_element_ligne > 0)
      {
        if(nb_element_ligne <= 1)
          {
            /* on n'affiche pas les espaces apres la cesure */
            elem_deb_ligne->justification &= 0xEF;

            /* s'il n'y a pas de cesure sur l'element */
            if((elem_deb_ligne->justification & 0x02) != 0x02)
              {
                /* on n'affiche pas les espaces apres la cesure */
                elem_deb_ligne->justification &= 0xEB;
              }
            /* on n'affiche pas les espaces apres la cesure ni ceux avant l'element */
            elem_deb_ligne->justification &= 0xEB;
          }
        else
          {
            nb_element_ligne --;
            /* on recupere le nombre d'espace et on calcule le nombre d'espace avant chaqu
e element */
            if(mode_cesure && (elem_deb_ligne->justification & 0x10) == 0x10)
              {
                nb_esp = elem_deb_ligne->nb_espap_cesure; /* si on est en mode cesure on r
ecupere le nombre d'espace apres la cesure */
              }
            else
              {
                nb_esp = elem_deb_ligne->nb_espav;
              }
            nb_esp_elem = (nb_esp - (nb_esp % nb_element_ligne)) / nb_element_ligne;
            nb_esp_elem_reste = nb_esp % nb_element_ligne;
            elem_tmp = elem_deb_ligne;
```

```
          /* tant qu'on a des elements sur la ligne, qu'on a n'a pas atteint la fin et q
u'on a encore des espaces a mettre */
          while(elem_deb_ligne != NULL && elem_deb_ligne != e_tmp && nb_esp >= 0)
            {
              /* si on est au premier element */
              if(elem_deb_ligne == elem_tmp)
                {
                  /* on n'affiche pas les espaces apres la cesure */
                  elem_deb_ligne->justification &= 0xEF;

                  /* s'il n'y a pas de cesure sur l'element */
                  if((elem_deb_ligne->justification & 0x02) != 0x02)
                    {
                      /* on n'affiche pas les espaces apres la cesure */
                      elem_deb_ligne->justification &= 0xEB;
                    }
                }

              /* si on n'est pqs au premier element */
              if(elem_deb_ligne != elem_tmp && elem_deb_ligne->type == 'M')
                {
                  /* on affiche nb_esp_elem espaces avant puis on decremente le nombre d
'espaces total */
                  elem_deb_ligne->justification |= 0x04;
                  elem_deb_ligne->nb_espav = nb_esp_elem;
                  nb_esp -= nb_esp_elem;

                  /* si le reste est superieur a 0 (la division du nombre d'espace par l
e nombre d'elements ne tombe pas juste), on en insere un et on le decremente */
                  if(nb_esp_elem_reste > 0)
                    {
                      elem_deb_ligne->nb_espav ++;
                      nb_esp_elem_reste --;
                    }
                }

              /* on passe a l'element suivant */
              elem_deb_ligne = elem_deb_ligne->suivant;

              /* si l'element est nul, fin de la phrase, on passe a la suivante */
              if(elem_deb_ligne == NULL)
                {
                  elem_deb_ligne = ph_tmp->elem;
                }
            }
        }
    }

  return retour;
}

int texte__verif_orthographe(texte t, char * dico[26][MAX_DICO], int nb_mots)
{
  /* DÃ©claration */
  struct paragraphe * pa; /* paragaphe */
  struct phrase * ph; /* la phrase */
  struct element_phrase * e; /* l'Ã©lement de paragraphe */
  int i, j;
  int trouve;
  int retour;

  /* Initialisation  */
  retour = -1;
  i = 0;
  j = 0;
  trouve = 0;

  pa = t;
  ph = pa->phrase;
```



```
  e = ph->elem;

  /* Tant qu'on a des paragraphes */
  while(pa != NULL && retour == -1)
    {
      ph = pa->phrase;
      /* Tant qu'on a des phrases */
      while(ph != NULL && retour == -1)
        {
          e = ph->elem;
          /* Tant qu'on a des mots, espaces ou ponctuation, on les affiche */
          while(e != NULL && retour == -1)
            {
              /* si l'element est un mot */
              if(e->type == 'M')
                {
                  j ++;
                  trouve = 0;
                  i = 0;

                  /* tant qu'il y a des mots commencant par cette lettre et qu'on n'a pa
s trouve une correspondance  */
                  while(i < nb_mots && !trouve)
                    {
                      /* si les chaines sont egales */
                      if(strcasecmp(dico[toupper(e->chaine[0]) - 65][i], e->chaine) == 0
)
                        {
                          /* on a trouve le mot */
                          trouve = 1;
                        }
                      i ++;
                    }

                  /* si on a pas trouve le mot on retourne l'indice du mot mal orthograp
hie */
                  if(!trouve)
                    {
                      retour = j;
                    }
                }
              e = e->suivant;
            }
          ph = ph->suivant;
        }
      pa = pa->suivant;
    }
  return retour;
}

int /* on un entier qui indique l'etat du traitement */
texte__typographie(
                   texte * t, /* le texte */
                   char * chaine, /* le tableau qui contient les chaÃ®nes de caractÃ¨res
  */
                   struct element_phrase * element, /* le tableau qui contient les struc
tures element_phrase */
                   struct phrase * phrase, /* le tableau qui contient les structures phr
ase */
                   struct paragraphe * paragraphe, /* le tableau qui contient les struct
ures paragraphe  */
                   int taille_para, /* la taille du tableau de structures paragraphe */
                   int taille_phrase, /* la taille du tableau de structures phrase */
                   int taille_element, /* la taille du tableau de structures element_phr
ase */
                   int taille_chaine, /* la taille du tableau des chaÃ®nes de caractÃ¨re
s */
                   int * indexelement, /* l'index de l'Ã©lÃ©ment */
                   int * indexphrase, /* l'index de la phrase */
```

```
                    int * indexparagraphe /* l'index du paragraphe */
)
{

  /* DÃ©claration */
  struct paragraphe * pa; /* paragaphe */
  struct phrase * ph; /* la phrase */
  struct element_phrase * e; /* l'Ã©lement de paragraphe */
  int retour;
  char last;
  char current;

  /* Initialisation  */
  retour = 1;
  last = ' ';
  current = ' ';

  pa = *t;
  ph = pa->phrase;
  e = ph->elem;

  /* Tant qu'on a des paragraphes */
  while(pa != NULL)
    {
      ph = pa->phrase;
      /* Tant qu'on a des phrases */
      while(ph != NULL)
        {
          last = ' ';
          e = ph->elem;

          /* Tant qu'on a des mots, espaces ou ponctuation, on les affiche */
          while(e != NULL)
            {
              /* type courant = type de l'element courant */
              current = e->type;

              /* si on est au debut de la toute premiere phrase et que ce n'est pas un m
ot */
              if(ph->precedent == NULL && last == ' ' && current != 'M')
                {
                  /* on cherche le premier mot */
                  while(e != NULL && e->type != 'M')
                    {
                      e = e->suivant;
                    }

                  /* si on le trouve on fait pointer le debut phrase sur lui et on met l
e premiere lettre en majuscule */
                  if(e != NULL)
                    {
                      e->precedent = NULL;
                      e->chaine[0] = toupper(e->chaine[0]);
                      ph->elem = e;
                    }
                  else /* sinon typographie impossible */
                    {
                      retour = 0;
                    }
                }

              /* si l'element courant est de type M ou si c'est une phrase commencant pa
r un espace puis un mot et que c'est le premier de la premiere phrase du paragraphe, on
met sa premiere lettre en majuscule */
              if((current ==  'M' || (e != NULL && e->suivant != NULL && ph->precedent
!= NULL && e->precedent == NULL && e->type == 'E')) && last == ' ')
                {
                  /* c'est un mot on met sa premiere lettre en majuscule */
                  if(current == 'M')
```



```
                    {
                      e->chaine[0] = toupper(e->chaine[0]);
                    }
                  else /* c'est un espace, on met la premiere lettre du mot suivant en m
ajuscule */
                    {
                      e->suivant->chaine[0] = toupper(e->suivant->chaine[0]);
                    }
                }

              /* il n'y a pas d'espace avant le premier mot de la phrase et que ce n'est
 pas la premiere du paragraphe, on en met un */
              if(e != NULL && last == ' ' && e->type != 'N' && e->type != 'E' && ph->pre
cedent != NULL)
                {
                  /* on rajoute s'il y a de la place dans le tableau */
                  if(*indexelement + 1 < taille_element)
                    {
                      *indexelement = *indexelement + 1;
                      element[*indexelement].suivant = e;
                      element[*indexelement].precedent = NULL;
                      element[*indexelement].chaine = " ";
                      element[*indexelement].type = 'E';
                      element[*indexelement].taille = 1;
                      e->precedent = & element[*indexelement];
                      ph->elem = & element[*indexelement];
                    }
                  else
                    {
                      retour = 0;
                    }
                }

              /* si l'element courant est une ponctuation double, on vÃ©rifie qu'il y ai
t un espace insÃ©cable avant et un espace apres */
              if(e != NULL && e->type == 'P' && (strcmp(e->chaine, ":") == 0 || strcmp(e
->chaine, "!") == 0 || strcmp(e->chaine, ";") == 0 || strcmp(e->chaine, "?") == 0))
                {
                  /* il n'y a pas d'espace avant */
                  if(e->precedent != NULL && e->precedent->type != 'E')
                  {
                    /* on rajoute s'il y a de la place dans le tableau */
                    if(*indexelement + 1 < taille_element)
                    {
                      *indexelement = *indexelement + 1;
                      element[*indexelement].suivant = e;
                      element[*indexelement].precedent = e->precedent;
                      element[*indexelement].chaine = " ";
                      element[*indexelement].type = 'E';
                      element[*indexelement].taille = 1;
                      e->precedent->suivant = & element[*indexelement];
                    }
                    else
                    {
                      retour = 0;
                    }
                  }

                  /* il n'y a pas d'espace apres */
                  if(e->suivant != NULL && e->suivant->type != 'E' && e->suivant->typ
e != 'N')
                  {
                    /* on rajoute s'il y a de la place dans le tableau */
                    if(*indexelement + 1 < taille_element)
                    {
                      *indexelement = *indexelement + 1;
                      element[*indexelement].suivant = e->suivant;
                      element[*indexelement].precedent = e;
                      element[*indexelement].chaine = " ";
```

```
                        element[*indexelement].type = 'E';
                        element[*indexelement].taille = 1;
                        e->suivant = & element[*indexelement];
                      }
                      else
                      {
                        retour = 0;
                      }
                    }
                  }

              /* s'il n'y a pas de point a la fin de la phrase on en met un */
              if(e != NULL && e->suivant != NULL && e->suivant->type == 'N' && e->type
!= 'P')
                {
                  /* on rajoute s'il y a de la place dans le tableau */
                  if(*indexelement + 1 < taille_element)
                    {
                      *indexelement = *indexelement + 1;
                      element[*indexelement].suivant = e->suivant;
                      element[*indexelement].precedent = e;
                      element[*indexelement].chaine = ".";
                      element[*indexelement].type = 'P';
                      element[*indexelement].taille = 1;
                      e->suivant = & element[*indexelement];
                      e = NULL; /* fin de la boucle car on est a la fin de la phrase */
                    }
                  else
                    {
                      retour = 0;
                    }
                }

              /* s'il y a un espace entre un mot et un virgule ou un point, on l'enlÃ¨ve
 */
              if(e != NULL && e->type == 'P' && (strcmp(e->chaine, ".") == 0 || strcmp(e
->chaine, ",") == 0) && e->precedent != NULL && e->precedent->type == 'E' && e->preceden
t->precedent != NULL && e->precedent->precedent->type == 'M')
                {
                  e->precedent = e->precedent->precedent;
                  e->precedent->suivant = e;
                }

              /* si on est sur une ponctuation, on regarde qu'il y ait au moins un mot e
ntre elle et la prochaine */
              if(e != NULL && e->type == 'P' && e->suivant != NULL && (e->suivant->type
== 'P' || (e->suivant->type == 'E' && e->suivant->suivant != NULL && e->suivant->suivant
->type == 'P')))
                {
                  retour = 0;
                }

              last = current;

              /* si l'element n'est pas nul on passe au suivant */
              if(e != NULL)
                {
                  e = e->suivant;
                }
            }
          ph = ph->suivant;
        }
      pa = pa->suivant;
    }

  return retour;
}
```

```
#ifndef FONCTIONS_TRAITEMENT_H
#define FONCTIONS_TRAITEMENT_H

#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<strings.h>
#include <ctype.h>
#include "structures.h"

/* DÃ©finition des prototypes de fonctions */
int texte__justifier(texte t, int l, char mode, char mode_cesure);
int texte__verif_orthographe(texte t, char * dico[26][MAX_DICO], int nb_mots);
int texte__typographie(texte * t, char * chaine, struct element_phrase * element, struct
 phrase * phrase, struct paragraphe * paragraphe, int taille_para, int taille_phrase, in
t taille_element, int taille_chaine, int * indexelement, int * indexphrase, int * indexp
aragraphe);

#endif
```