Mise en forme de texte

Il s'agit de mettre en forme un texte en anglais (pas de lettres accentuées) structuré en paragraphes, chacun d'eux identifié par une balise particulière insérée dans ce texte. Chaque paragraphe commence par une balise et se termine par la balise de début du paragraphe suivant ou la fin du texte.

Chaque balise est de la forme :
#$C_1C_2C_3C_4C_5$ où
C_1 est la lettre *C* (Centrer), *D* (cadrer à Droite), *G* (cadrer à Gauche) ou *J* (Justifier)
C_2C_3 définissent la *largeur du paragraphe* (nombre maximum de caractères par ligne) sur 2 positions,
C_4C_5 définissent un *retrait gauche du paragraphe* (nombre d'espaces à insérer en tête de la première ligne du paragraphe) sur deux positions. Ce retrait doit être inférieur à la largeur du texte et doit être nul pour les balises *C* et *D*.

Par exemple :
#J8005 définit une balise permettant de justifier un paragraphe dont les lignes font 80 caractères avec un retrait de 5 espaces sur la première ligne.
#C8000 définit une balise permettant de centrer un paragraphe dont les lignes font au maximum 80 caractères.

Exemple de texte à mettre en forme :

```
#C8000 Abstract #J8005
We present a design, including its movitation, for introducing concurrency into
C++.
The design work is based on a set of requirements and elementary execution
properties that generate a corresponding
set of programming language constructs needed to express concurrency. #G6002 The
new constructs continue to
support
object-oriented
facilities such as inheritance and code reuse. Features that allow flexibility in
accepting and subsequently
postponing
servicing of
requests are provided. #D4000 Currently, a major portion of the design is
implemented, supporting concurrent programs on
shared-memory
uniprocessor and multiprocessor computers.
```

Exemple de texte mise en forme après analyse des balises :

```
                                Abstract
     We  present a design, including its movitation, for introducing concurrency
into  C++.  The  design  work  is  based on a set of requirements and elementary
execution  properties  that generate a corresponding set of programming language
constructs needed to express concurrency.
  The new constructs continue to support object-oriented
facilities such as inheritance and code reuse. Features that
allow flexibility in accepting and subsequently postponing
servicing of requests are provided.
Currently, a major portion of the design
   is implemented, supporting concurrent
  programs on shared-memory uniprocessor
           and multiprocessor computers.
```

1ère Partie : Mise en forme d'un texte supposé correctement balisé

1) Créer le répertoire MISE_EN_FORME_1. Dézipper le fichier *miseEnForme-1.zip* dans le répertoire créé. Ce fichier contient les entêtes des sous-programmes utilisés dans l'application (*miseEnforme.h)*, le fichier objet associé (*miseEnForme.o)*, le makefile ainsi qu'un exemple de fichier à mettre en forme (*fichierAMettreEnForme.txt)*.

2) Ecrire l'application *client.C* qui permet de prendre en entrée un fichier texte balisé (on pourra prendre par exemple *fichierAMettreEnForme.txt*) et qui produit un fichier mis en forme après analyse des balises en s'appuyant sur la bibliothèque de sous-programmes *miseEnForme* fournie et sur l'algorithme général ci-dessous. L'application lit et écrit sur les fichiers standards d'entrée-sortie. On utilisera donc les redirections.

```
-- mettre en forme un texte
    lire le premier mot (correspond forcément à une balise) ;
    tantque il reste un paragraphe faire
        traiter le paragraphe ;
    fin tantque ;
```

3) Compléter les entêtes des sous-programmes contenues dans le fichier *miseEnforme.h* pour préciser le rôle joué par chaque paramètre.

4) Compléter le corps du sous-programme *traiterParagraphe2()* et l'intégrer dans le programme principal. Remplacer l'appel à *traiterParagraphe()* du programme principal par *traiterParagraphe2()* pour pouvoir le tester.

```
// traite un paragraphe
void traiterParagraphe2(Mot & motEnAttente)
{
    Balise balise;
    // construire la balise de debut de paragraphe a partir du mot en attente
    balise = …;
    switch (balise.cadrage)
    {
        case 'J' :
            justifierParagraphe(motEnAttente, balise.largeur, balise.retrait);
            break;
        case 'D' :
            …
        case 'G' :
            …
        case 'C' :
            …
    }
}
```

5) En vous inspirant du sous-programme *cadrerGaucheParagraphe()* défini ci-dessous, écrire le corps des sous-programmes *cadrerDroiteParagraphe2()*, *centrerParagraphe2()* et *justifierParagraphe2()* qui doivent donner le même résultat que les sous-programmes de cadrage de paragraphes fournis dans *miseEnforme.h*.

```
// cadre a gauche un paragraphe
void cadrerGaucheParagraphe(Mot &motEnAttente, const int largeur,
const int retrait)
{
    ListeMots l;
    Ligne ligne;
    // lire le mot suivant qui correspond au 1er mot du paragraphe
    lireMot(motEnAttente);
    // lire les mots de la premiere ligne
    remplirListeDeMots(l, motEnAttente, largeur-retrait);
    // construire et ecrire la premiere ligne
    cadrerGaucheLigne(ligne, l, largeur, retrait);
    ecrireLigne(ligne);
    // tantque il reste une ligne dans le paragraphe faire
    while (!cin.eof() && !estUneBalise(motEnAttente))
    {
        // lire les mots de la ligne suivante
        remplirListeDeMots(l, motEnAttente, largeur);
        // construire et ecrire la ligne
        cadrerGaucheLigne(ligne, l, largeur, 0);
        ecrireLigne(ligne);
    }
}
```

6) En vous inspirant du sous-programme *cadrerGaucheLigne()* défini ci-dessous, écrire le corps des sous-programmes *cadrerDroiteLigne2()* et *centrerLigne2()*qui doivent donner le même résultat que les sous-programmes de cadrage de lignes fournis dans *miseEnforme.h*.

```
// cadre une ligne a gauche
void cadrerGaucheLigne(Ligne &ligne, const ListeMots &l, const int largeur,
const int retrait)
{
    ligne.lg = 0;
    // recopier les blancs dans la ligne
    recopierBlancs(ligne, retrait);
    // recopier les mots de la liste dans la ligne
    recopierMots(ligne, l, 1);
}
```

7) Ecrire le corps du sous-programme *justifierLigne2()* remplaçant *justifierLigne()* à partir de l'algorithme ci-dessous et en supposant :

nbCaracteresMots : nombre de caractères des mots de la ligne,
nbIntervalles : nombre d'intervalles entre les mots de la ligne,
nbEspacesParIntervalle : nombre minimal d'espaces entre deux mots,
nbEspacesRestant : nombre d'espaces supplémentaires à répartir,
retrait : nombre d'espaces correspondant au retrait de début de ligne,
largeur : nombre de caractères de la ligne,

tels que :

```
largeur = retrait + nbCaracteresMots + (nbIntervalles*nbEspacesParIntervalle) + nbEspacesRestant
```

```
-- justifier une ligne
    inserer les blancs correspondant au retrait de la ligne (forcement nul pour
    les lignes du paragraphe autres que la premiere);
    si il n'y a qu'un mot dans le paragraphe alors
        recopier ce mot;
    sinon
        calculer nbEspacesTotal;
        calculer nbIntervalles;
        calculer nbEspacesParIntervalle;
        calculer nbEspacesRestant;
        recopier les nbEspacesRestant premiers mots suivis chacun de
        nbEspacesParIntervalle+1 espaces;
        recopier les mots restants séparés par nbEspacesParIntervalle espaces;
    fin si;
```

8) Ecrire le corps des sous-programmes *valeur2()*, *constructionBalise2()* et *remplirListeDeMots2()* remplaçant respectivement *valeur()*, *constructionBalise()* et *remplirListeDeMots()*.

2ème Partie : Mise en forme d'un texte pouvant contenir des erreurs de balises

1) Créer le répertoire MISE_EN_FORME_2. Dézipper le fichier *miseEnForme-2.zip* dans le répertoire créé. Ce fichier contient les entêtes des sous-programmes utilisés dans l'application (*miseEnforme.h)* et le fichier objet associé (*miseEnForme.o)*.

2) Compléter le corps de *constructionBalise2()* pour que ce sous-programme déclenche une exception différente (de type *Exception*) pour chacun des cas d'anomalie suivants :

- 1 : la longueur de la balise est différente de 6,
- 2 : C_1 est différent de 'C', 'D', 'G' ou 'J',
- 3 : C_2 et C_3 ne sont pas des chiffres,
- 4 : C_4 et C_5 ne sont pas des chiffres,
- 5 : la largeur du paragraphe est supérieure à 80,
- 6 : le retrait gauche est différent de 0 pour un paragraphe centré ou cadré à droite,
- 7 : le retrait gauche du paragraphe est supérieur ou égal à la largeur du paragraphe.

3) Modifier le programme source pour que chaque déclenchement d'exception produise un message dans le flot standard d'erreur et ensuite effectue le traitement suivant
   - dans le cas 1, on abandonne le programme en propageant l'exception jusqu'au programme principal,
   - dans les cas 2 à 4, on ignore le paragraphe,
   - dans le cas 5 à 7, on remplace la balise par la balise #G8000.

4) Créer un fichier de test permettant de tester tous les cas d'anomalie.

ANNEXES

LES REDIRECTIONS (RAPPEL DE COURS)

Pour exécuter le programme et prendre le texte fourni dans le fichier, il faut utiliser le mécanisme de redirection vu en Shell.

Par exemple :

```
cmd <fic1 >fic2 2>fic3
```

permet d'exécuter la commande cmd en utilisant :

- comme entrée standard (à la place du clavier) le fichier *fic1*,
- comme sortie standard (à la place de l'écran) le fichier *fic2*,
- comme sortie d'erreur standard (à la place de l'écran) le fichier *fic3*.

Dans le contexte de ce TP, le programme client correspond à la commande cmd dans l'exemple ci dessus.

LES ENTREES-SORTIES (RAPPELS DE COURS)

Pour faire des entrées sorties standards, C++ utilise *cin* associé au flot standard d'entrée, *cout* associé au flot standard de sortie et *cerr* associés au flot standard d'erreur.

- ***cin >>*** lit sur l'entrée standard *stdin* les différentes valeurs et les affecte aux identificateurs de variables précisés dans l'appel.

cin >>identificateur1 >>identificateur2 >>...>>identificateurN

- ***cout*** << affiche sur la sortie standard stdout les valeurs des différentes expressions, suivant une présentation adaptée à leur type.

cout <<expression1 <<expression2 <<... <<expressionN

- ***cerr*** << affiche sur la sortie standard stdout les valeurs des différentes expressions, suivant une présentation adaptée à leur type.

cerr <<expression1 <<expression2 <<... <<expressionN

Pour pouvoir utiliser *cin*, *cout* et *cerr*, il faut inclure le fichier d'en-tête *<iostream>* dans le programme source comme dans l'exemple ci-dessous.

cin>>, *cout*<< et *cerr*<< adaptent le format de saisie ou d'affichage en fonction du type des arguments.

Exemple :

```
#include <iostream>
using namespace std;

int main()
{
    int a, b;
    cout <<"Entrer deux entiers ";
```

```
    cin >>a >>b;
    cout <<"le produit de " <<a <<" par " <<b <<"\nest : " <<a*b;
}
```

Remarques :

- Le manipulateur ***flush*** permet de forcer le vidage de la mémoire tampon associée au fichier de sortie.
- Le manipulateur ***endl*** insère d'abord le caractère '\n' avant de faire flush.

Exemple :

```
#include <iostream>
using namespace std;

int main()
{
    int a, b;
    cout <<"Entrer deux entiers " <<flush;
    cin >>a >>b;
    cout <<"le produit de " <<a <<" par " <<b <<"est : " <<a*b <<endl;
}
```

- ***bool cin.eof()*** : teste si la fin du fichier standard d'entrée est atteinte (tous les caractères ont été lus).

- ***char cin.get()*** : lit un caractère sur l'entrée standard.