MOOC Init. Prog. C++
Exercices supplémentaires facultatifs semaine 6

Segmentation en mots (niveau 3)

> Cet exercice correspond à l'exercice n°19 (pages 56 et 219) de l'ouvrage *C++ par la pratique* (3e édition, PPUR).

Cadre

On s'intéresse ici au problème de la segmentation d'un texte en mots. Le but est de trouver, l'un après l'autre, les différents «mots» d'un texte ; un «mot» étant défini comme une séquence de caractères ne contenant pas de «séparateur». Pour simplifier, on considérera ici que le seul caractère séparateur est l'*espace* (i.e. `' '`).

Par exemple, le texte « heuu bonjour, voici ma chaîne ! » aura comme «mots» : «heuu», «bonjour,» (y compris la virgule ici), «voici», «ma», «chaîne» et «!».

Fonction `nextToken`

Dans le fichier `token.cc`, prototypez puis définissez la fonction :

```
bool nextToken(string const& str, int& from, int& len)
```

dont le rôle sera d'identifier (par la position de départ et la longueur) le prochain mot à partir de la position `from` dans la chaîne `str`. Elle indiquera de plus, par sa valeur de retour, si elle a trouvé un mot ou non.

Cette fonction modifiera donc les arguments `from` et `len` de sorte qu'ils déterminent la position du premier caractère du mot trouvé et sa longueur, pour autant qu'un tel mot existe (et dans ce cas la fonction devra retourner `true`).

Dans le cas où il n'existe plus de mot à partir de `from` dans `str`, le résultat retourné par cette fonction `nextToken` sera `false` (et les valeurs modifiées de `from` et `len` ne seront plus significatives).

Par exemple, si l'on appelle `nextToken` avec `" heuu bonjour, voici ma chaîne ! "` dans `str` et `0` dans `from`, la fonction retournera `true` (oui, elle a trouvé un mot : «heuu») et aura modifié `from` à `1` (ce mot trouvé commence à la position 1) et `len` à `4` (le mot trouvé a une longueur de 4 caractères).

Si on l'appelle avec la même chaîne, mais `6` dans `from`, la fonction retournera `true` (oui, elle a trouvé un mot : «bonjour,») et aura laissé `from` à `6` (ce mot trouvé commence à la position 6) et `len` à `8`.

Si par contre, on appelle `nextToken`, toujours avec la même chaîne, mais `32` dans `from`, elle retournera alors `false` (non, elle n'a pas trouvé de mot à partir de la position 32).

Application

Depuis le `main` du programme, vous demanderez à l'utilisateur d'entrer une chaîne de caractères au clavier, et afficherez (en faisant des appels successifs à la fonction `nextToken`) l'ensemble des mots de la chaîne entrée, à raison de un mot par ligne, placés entre apostrophes.

Notez que pour lire une phrase entière (avec les espaces) depuis `cin`, il faut faire :

```
  getline(cin, phrase);
```

parce que la lecture usuelle (p.ex. « `cin >> mot;` ») ne lit pas les espaces et s'arrête à la première espace rencontrée (oui, l'espace du typographe est féminine).

Utilisez l'exemple de fonctionnement ci-après pour vérifier la validité de votre programme ; faites en particulier attention à ce que les apostrophes entourent les mots **sans qu'il y ait d'espace entre les deux** .

Vérifiez également que le programme se comporte correctement même lorsque la chaîne entrée se termine par une suite d'espaces.

Exemple de fonctionnement :

```
Entrez une chaîne :  heuu bonjour, voici ma chaîne !
Les mots de " heuu bonjour, voici ma chaîne !  " sont:
'heuu'
'bonjour,'
'voici'
'ma'
'chaîne'
'!'
```

Fractions (niveau 2)

Nous allons définir une structure `Fraction`, qui permettra de représenter des fractions:

```
  struct Fraction
  {
    int numerateur;
    int denominateur;
  };
```

qui correspondra à la fraction

$$\frac{\texttt{numerateur}}{\texttt{denominateur}}.$$

Nous voulons que les fractions soient toujours irréductibles, même après un calcul. Par exemple, le produit des fractions $^{4}\!/_{25}$ et $^{15}\!/_{2}$ devra donner la fraction $^{6}\!/_{5}$, et non pas la fraction $^{60}\!/_{50}$. Pour cela, on pourra utiliser la fonction `pgcd`:

```
unsigned int pgcd(unsigned int a, unsigned int b)
{
  unsigned int m;

  if (a < b) {
    m = a;
  } else {
    m = b;
  }

  while ((a % m != 0) or (b % m != 0)) {
    --m;
  }

  return m;
}
```

Ainsi, la fonction `init_frac` s'écrit:

```
Fraction init_frac(int num, int den)
{
  const unsigned int div( pgcd( abs(num), abs(den) ) );

  if (num < 0 and den < 0) {
    num = -num;
    den = -den;
  }

  return { num / div , den / div };
}
```

Ecrivez les fonctions `afficher_frac`, `add_frac`, `mult_frac`, `mult_scal_frac`, dont le but est, respectivement, d'afficher une fraction, d'additioner 2 fractions, de multiplier 2 fractions et de multiplier une fraction par un scalaire.

En utilisant la fonction `init_frac`, les fonctions `add_frac`, `mult_frac`, `mult_scal_frac` peuvent s'écrire très simplement, sur une seule ligne.
