MOOC Init. Prog. C++
Correction des exercices supplémentaires semaine 6

Les corrigés proposés correspondent à l'ordre des apprentissages : chaque corrigé correspond à la solution à laquelle vous pourriez aboutir au moyen des connaissances acquises jusqu'à la semaine correspondante.

Exercice 16 : segmentation

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

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

bool nextToken(const string& str, size_t& from, size_t& len);
bool issep (char c); // teste si le caractère est un separateur

int main()
{
  string phrase;

  cout << "Entrez une chaîne : ";
  getline(cin,  phrase);

  cout << "Les mots de \"" << phrase << "\" sont :" << endl;
  size_t debut(0);
  size_t longueur(0);
  while (nextToken(phrase, debut, longueur)) {
    cout << "'" << phrase.substr(debut, longueur) << "'" << endl;
    debut += longueur;
  }
  return 0;
}

/* La fonction suivante teste si le caractère est un séparateur.
 *
 * Ecrire une fonction présente l'avantage de pouvoir redéfinir facilement
 * la notion de séparateur (et éventuellement d'en définir plusieurs).
 */
bool issep (char c)
{
  return (c == ' ');
}

/* Il y a de multiples façons d'écrire cette fonction.
```

```
 * Nous trouvons celle-ci est assez élégante.
 */
bool nextToken(const string& str, size_t& from, size_t& len)
{
  const size_t taille(str.size());

  // saute tous les separateurs à partir de from
  while ((from < taille) and issep(str[from])) ++from;

  // avance jusqu'au prochain séparateur ou la fin de str
  len = 0;
  for (size_t i(from); ((i < taille) and not issep(str[i])); ++len, ++i);

  return (len != 0);
}
```

Commentaires

Le corrigé proposé pour `nextToken` est assez compact (cf commentaire donné avant) et nécessite d'avoir bien compris les structures de contrôle (`while`, `for`), ce qui le rend un peu plus difficile à comprendre que d'autres solutions.

Comme indiqué, vous pouvez le faire de plein de façons différentes ; mais examinons justement celle proposée :

```
bool nextToken (string const& str, int& from, int& len)
{
   int const taille(str.size());
```

Bon, jusque là je pense que ça va... ; juste peut être préciser les arguments :

- `string const& str` : la chaîne à traiter. Le passage par « const ref » est, bien sûr, une optimisation ; et l'on pourra dans un premier temps se contenter d'un simple passage par valeur : `string str` ;
- `int& from` : contient au départ la première position à partir de laquelle il faut chercher un nouveau « token », et devra contenir à la sortie de `nextToken` la position du début du token suivant ; il va donc être modifié par `nextToken`, et c'est pour cela qu'il est passé par référence ;
- `int& len` : va également être modifié par `nextToken` pour contenir la longueur du nouveau « token » (éventuellement) trouvé.

Continuons avec :

```
while ((from < taille) && issep(str[from])) ++from;
```

`from` est ici incrémenté(/augmenté) du nombre de séparateurs rencontrés. En effet, on ne peut pas commencer le nouveau « token » par un séparateur.

Comment fait-on pour sauter tous ces séparateurs ?
-> tant que le caractère courant est un séparateur, on avance. Ça, c'est le « `while (issep(str[from]))` ».

Mais il faut penser aussi à ne pas déborder de la chaîne (imaginez le cas où la chaîne ne contient que des séparateurs). Ça, c'est le « `(from < taille)` » dans le test du `while`.

Passons au bloc suivant. Son but est de trouver la fin du « token » (puisqu'on vient de trouver le début avec la boucle `while` précédente).

Cette fin de « token » est en fait indiquée par la longueur, `len`, du « token ». Au départ le « token » est vide (on a très bien pu sortir du `while` précédent par la condition « `from ≥ taille` ». Repensez encore une fois au cas d'une chaîne ne contenant aucun « token », mais uniquement des séparateurs). On a donc :

```
len = 0;
```

Puis on cherche la fin du « token » ; c'est-à-dire le prochain séparateur. C'est-à-dire que tant que l'on n'a pas de séparateur (« `!issep(str[i])` »), on avance.

Ca veut dire quoi « on avance » ?
-> on passe au caractère suivant, ça c'est le « `++i` », *et* on met à jour la taille du « token » en la faisant augmenter de 1, ça c'est le « `++len` ».

D'où part-on ?
-> du caractère « `from` » précédemment trouvé.

Il ne reste plus qu'à ne pas oublier de ne pas déborder de la chaîne (« `i < taille` »), et le tour est joué :

```
for (int i(from); ((i < taille) && !issep(str[i])); ++len, ++i);
```

Pour essayer d'être encore plus clair, ceci peut aussi s'écrire de la façon moins compacte suivante (rappel : « `from` » représente le début de « token ») :

```
bool continuer(true);
int position_courante(from);

do {
// si on déborde de la chaine il faut s'arrêter
if (position_courante >= taille) {
 continuer = false ;
}

// si on rencontre un séparateur, il faut s'arrêter
       // (c'est la fin du token)
else if (issep(str[position_courante])) {
 continuer = false ;
}

// sinon, tout va bien, ...
else {
 // ...on augmente la longueur du token de 1...
 len = len + 1;

 // ...et on passe au caractère suivant.
 ++position_courante;
}
```

```
 } while (continuer);
```

Voilà pour cette partie.

Et pour finir, on renvoit (« `true` » si on a trouvé un « token », c.-à-d. si `len` n'est pas nulle, et « `false` » sinon. Soit :

```
return (len != 0);
```

qui revient exactement à

```
if (len != 0) {
 return true;
} else {
 return false;
}
```

Exercice 17 : Fractions

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

// ============================================================================
// la structure de données
struct Fraction
{
  int numerateur;
  int denominateur;
};

// ============================================================================
// Calcul du plus grand diviseur commun
unsigned int pgcd(unsigned int a, unsigned int b)
{
  unsigned int m(b);
  if (a < b) {
    m = a;
  }

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

  return m;
}

// ============================================================================
// Initialisation d'une fraction, sous sa forme irréductible
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 };
}

// ============================================================================
// Affichage d'une fraction
void afficher_frac(Fraction f)
{
  cout << f.numerateur << "/" << f.denominateur;
}

// ============================================================================
// Addition de 2 fractions
Fraction add_frac(Fraction f1, Fraction f2)
{
  // Note : la fonction init_frac rend la fraction irreductible
  return init_frac(f1.numerateur * f2.denominateur + f2.numerateur * f1.denomin
                   f1.denominateur * f2.denominateur);
```

```
}

// ========================================================================
// Multiplication de 2 fractions
Fraction mult_frac(Fraction f1, Fraction f2)
{
  return init_frac(f1.numerateur   * f2.numerateur,
                   f1.denominateur * f2.denominateur);
}

// ========================================================================
// Multiplication d'une fractions par un nombre
Fraction mult_scal_frac(int scalaire, Fraction f)
{
  return init_frac(f.numerateur * scalaire, f.denominateur);
}

// ========================================================================
int main()
{
  Fraction f1 = init_frac(5, 2);  // 5/2
  Fraction f2 = init_frac(3, 12); // 3/12 --> 1/4

  cout << "f1 = ";
  afficher_frac(f1);
  cout << " et f2 = ";
  afficher_frac(f2);
  cout << endl;

  cout << "f1 + f2 = ";
  afficher_frac(add_frac(f1, f2));
  cout << endl;

  cout << "f1 * f2 = ";
  afficher_frac(mult_frac(f1, f2));
  cout << endl;

  cout << "2 * f2 = ";
  afficher_frac(mult_scal_frac(2, f2));
  cout << endl;

  return 0;
```