[Aller plus loin] Les expressions régulières 1

Lorsque nous lisons un texte, nous sommes capable de reconnaître la signification (ou sémantique) de certains motifs dans le texte. Par exemple, si on écrit `“25/12/2014”`, beaucoup de personnes reconnaîtront une date, correspondant au 25 décembre 2014. Si on écrit `“18:30”`, on reconnaît une heure : dix-huit heure trente. Ou encore, on reconnait que “`http://www.google.fr`” est une URL internet.

Nous sommes capable de trouver la sémantique d'une chaîne parce que l'on connaît le motif qui caractérise cette chaîne. On a l'habitude d'écrire les dates en indiquant le jour, le mois et l'année (en français). On a l'habitude aussi de voir des URL écrites sous la forme `“http:/ /”` suivi de plusieurs mots séparés par des points ou des barres obliques.

Les expressions régulières sont un moyen efficace d'écrire de tels motifs. Avec ces motifs, il sera ensuite possible de vérifier qu'une chaîne respecte ce motif ou encore d'identifier les sous-chaînes qui respectent ce motif.

Création et initialisation

Origine du terme “expression régulière” ?

Les expressions régulières sont une fonctionnalité que l'on trouve dans beaucoup de langages de programmation modernes. En C++, une expression régulière correspond à la classe `regex` de la bibliothèque standard (dans le fichier d'en-tête `regex`). Il est possible de créer une expression régulière directement à partir d'une littérale chaîne de caractères ou d'une variable chaîne de type `string`.

main.cpp

```cpp
#include <regex>
#include <string>

int main()
{
    std::regex pattern1 { "bla bla bla" }; // création à
partir d'une littérale

    std::string s { "bla bla bla" };
    std::regex pattern2 { s };             // création à
partir d'une string
}
```

Utilisation des raw string

Pour bien comprendre les expressions régulières, il faut donner quelques définitions :

- une **séquence cible** (*target sequence*) est la chaîne de caractères sur laquelle est appliquée l'expression régulière.
- un **motif** (*pattern*) est la séquence de caractères représentant ce que l'on cherche à identifier.
- une **correspondance** (*match*) est une sous-chaîne de la séquence cible qui correspond au motif.

Plus concrètement, si l'on prend la chaîne suivante : “La date du 25/12/2014 est un jeudi” et que l'on demande d'écrire une expression régulière pour trouver la date dans cette chaîne, alors la chaîne “La date du 25/15/2014 est un jeudi” est la séquence cible et la correspondance est “25/12/2014”. Le motif est une chaîne qui signifie “trouver une date au format jour/mois/année”. Bien sûr, il n'est pas possible d'écrire un motif de cette façon, il faut utiliser une syntaxe spécifique, qui sera décrite dans la suite de ce chapitre.

Avec une expression régulière, on va donc pouvoir réaliser principalement trois opérations, chaque opération correspondant à une fonction. Ces différentes fonctions seront détaillées dans les prochains chapitres, la suite de ce chapitre sera consacrée à la syntaxe utilisable pour écrire un motif. Mais pour vous permettre de pratiquer et apprendre

correctement les expressions régulières, nous allons voir rapidement une syntaxe possible de ces fonctions (il est possible d'utiliser ces fonctions de différentes façons, nous n'en verrons qu'une seule pour le moment).

La première fonctionnalité des expressions régulières est la **validation** d'une chaîne, c'est-à-dire vérifier qu'une chaîne respecte un motif. La fonction correspondante est la fonction `regex_match`. Une version simple de cette fonction prend en arguments la séquence cible et l'expression régulière et retourne une valeur booléenne (vrai si la séquence cible correspond au motif, faux sinon).

main.cpp

```
#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::regex pattern { "abc" }; // on recherche le motif
"abc"
    std::string target { "abcdef" };
    bool result = std::regex_match(target, pattern);
    std::cout << std::boolalpha << result << std::endl;

    target = "abc";
    result = std::regex_match(target, pattern);
    std::cout << std::boolalpha << result << std::endl;
}
```

affiche :

```
false
true
```

Le motif “abc” correspond donc à la séquence cible “abc”, mais pas à la séquence “abcdef”. Il permet donc de vérifier que la séquence cible correspond exactement à la chaîne “abc”. Ce n'est pas très utile, le but est surtout pour montrer la syntaxe de la fonction.

La deuxième utilité des expressions régulières est de **rechercher** les

sous-chaînes de la séquence cible correspondant au motif. La fonction correspondante est `regex_search`. Celle-ci prend les mêmes arguments que la fonction `regex_match` et retourne vraie si la fonction trouve au moins une sous-chaîne correspond au motif.

main.cpp

```
#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::regex pattern { "abc" }; // on recherche le motif
"abc"
    std::string target { "abcdef" };
    bool result = std::regex_search(target, pattern);
    std::cout << std::boolalpha << result << std::endl;

    target = "abc";
    result = std::regex_search(target, pattern);
    std::cout << std::boolalpha << result << std::endl;
}
```

affiche :

```
true
true
```

Le code est le même que précédemment, en remplaçant `regex_match` par `regex_search`. Par contre, le résultat est très différent : la séquence cible `“abcdef”` ne correspondait pas au motif lorsque l'on utilise `regex_match`, mais il correspond en utilisant `regex_search` (il existe bien le motif `“abc”` dans la séquence `“abcdef”`).

Pour terminer, la troisième utilisation principales des expressions est le remplacement des sous-chaînes correspondant à un motif par d'autres chaînes. La fonction correspondante est `regex_replace`, qui prend en argument la séquence cible, l'expression régulière (comme pour les précédentes fonctions) et la chaîne de remplacement, puis retourne la chaîne après modifications.

main.cpp

```
#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::regex pattern { "abc" }; // on recherche le motif
"abc"
    std::string target { "abcdefabc" };
    std::string replacement { "123" };
    std::string result = std::regex_replace(target, pattern,
replacement);
    std::cout << result << std::endl;
}
```

affiche :

```
123def123
```

Ce qui correspond bien à la chaîne `“abcdefabc”`, en remplaçant les sous-chaînes `“abc”` par `“123”`.

Les différentes syntaxes possibles

Maintenant que vous avez une idée générale du fonctionnement des expressions régulières, il va falloir apprendre à écrire des motifs. La syntaxe à utiliser n'est pas très compliquée quand on a l'habitude, mais elle est écrite sous forme assez compacte dans une chaîne, ce qui ne facilite pas la lecture. Il ne faut donc pas hésiter à s'attarder un peu sur ce chapitre et pratiquer un maximum d'exercices, le temps de bien assimiler la syntaxe des expressions régulières.

Même si les expressions régulières sont utilisables dans de nombreux langages, cela ne veut pas dire pour autant que la syntaxe pour écrire des motifs soit la même dans tous ces langages. Il existe en pratique plusieurs syntaxes possibles, par

exemple :

- ECMAScript, issu de l'éditeur de texte Emacs ;
- basic posix et extended posix, définis dans la norme POSIX pour Linux ;
- grep et egrep, issus du programme en ligne de commande grep (recherche de chaînes sous Linux) ;
- awk, issu du programme en ligne de commande du même nom.

La classe `std::regex` du C++ prend en charge différentes syntaxes, que l'on peut spécifier comme second argument lors de la création d'une expression régulière. Par exemple :

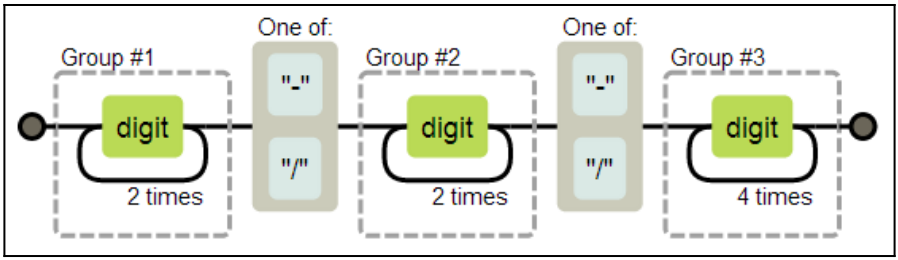
```
std::regex("abc", std::regex::ECMAScript);
```

La liste des syntaxes acceptées par `std::regex` est disponible dans la documentation de cette classe. Par défaut, c'est la syntaxe ECMAScript qui est utilisée, nous utiliserons donc uniquement cette syntaxe dans ce cours. Mais sachez que d'autres syntaxes sont possibles.

Visualiser les expressions régulières

Lorsque l'on débute avec les expressions régulières, il est parfois plus facile de visualiser le motif sous forme graphique. Si vous êtes plus à l'aise pour apprendre de cette façon, il existe des outils qui permettent de générer une représentation graphique à partir d'une expression régulière.

Par exemple, l'expression régulière `”(\d{2})[-/](\d{2})[-/](\d{4})”`, qui permet de valider une date, peut être représentée sous la forme suivante :



Ce graphique se lit de gauche vers la droite, il suffit de suivre les chemins possibles pour lire ce motif. Ce motif est donc constitué de trois groupes séparés par les caractères `-` ou `/`. Chaque groupe est constitué de chiffres (“digit”) répété 2 fois pour les deux premiers groupes et 4 fois pour le dernier groupe.

L'ensemble des graphiques de ce cours pour les expressions régulières sont générés automatiquement par le site http://www.regexper.com/ et sont sous licence Creative Common (CC BY 3.0).

Les caractères de base

La forme la plus simple de motif est constitué de caractères alphanumériques de base (a, b, E, G, 3, 7, etc). Chaque caractère du motif est recherché dans la séquence cible à l'identique. Ainsi, le motif “abc” permettra de rechercher dans une chaîne cette séquence exacte.

main.cpp

```cpp
#include <iostream>
#include <regex>

int main()
{
    std::regex const pattern { "a" };
    std::cout << "'a' match with '': " << std::boolalpha <<
        std::regex_match("", pattern) << std::endl;

    std::cout << "'a' match with 'a': " << std::boolalpha <<
        std::regex_match("a", pattern) << std::endl;
```

```
    std::cout << "'a' match with 'b': " << std::boolalpha <<
        std::regex_match("b", pattern) << std::endl;

    std::cout << "'a' match with 'ab': " << std::boolalpha
<<
        std::regex_match("ab", pattern) << std::endl;
}
```

affiche :

```
'a' match with '': false
'a' match with 'a': true
'a' match with 'b': false
'a' match with 'ab': false
```

Ainsi, le motif “a” ne peut correspondre que si la séquence cible correspond exactement à “a”, les chaînes ””, “b” et “ab” ne correspondent pas.

Attention de bien faire attention à la fonction que l'on utilise. Si on réalise une recherche de sous-chaînes (avec `regex_search`) au lieu d'une validation de chaîne (avec `regex_match`), le résultat obtenu n'est pas identique.

main.cpp

```
#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::regex pattern { "a" };
    std::cout << "'a' match with 'ab': " << std::boolalpha
<<
        std::regex_match("ab", pattern) << std::endl;

    std::cout << "search 'a' in 'ab': " << std::boolalpha <<
        std::regex_search("ab", pattern) << std::endl;
}
```

affiche :

```
'a' match with 'ab': false
search 'a' in 'ab': true
```

Dans le premier cas, la chaîne “ab” ne correspond pas au motif “a” (elles ne sont pas identiques). Dans le second cas, le motif “a” est retrouvé dans le chaîne “ab” (la séquence cible contient le motif).

Il est possible d'utiliser la grande majorité des caractères dans un motif (même des caractères étrangers, ce point sera détaillé dans la partie “Internationalisation”). Cependant, certains caractères ont une signification particulière (que vous verrez par la suite) dans un motif et ne peuvent être utilisé directement.

Imaginons que vous souhaitez par exemple vérifier qu'une chaîne contient un point. Vous pourriez écrire le code suivant par exemple :

main.cpp
```cpp
#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::regex const pattern { "." };
    std::string target { "cette phrase contient un point" };
    std::cout << target << " : " << std::boolalpha <<
        std::regex_search(target, pattern) << std::endl;

    target = "cette phrase contient un point.";
    std::cout << target << " : " << std::boolalpha <<
        std::regex_search(target, pattern) << std::endl;
}
```

On pourrait s'attendre à ce que, la recherche échoue dans le premier cas et réussit dans le second. Cependant, le résultat affiché ne correspond pas à ce que l'on attend.

```
cette phrase contient un point : true
cette phrase contient un point. : true
```

il y a déjà beaucoup à dire sur les regex, parler des caractères d'échappement et des raw string plus tôt

pas clair, à réécrire

Donc la recherche réussit dans les deux, alors que la première séquence cible ne contient pas de point. La raison est que le point est un caractère spécial et ne permet pas de rechercher un point dans la séquence cible. Pour rechercher le caractère point `.`, il faut le faire précéder du caractère barre oblique inversée `\`. Le motif pour rechercher un point dans une chaîne est donc “\.” et non pas simplement ”.”.

Cependant, le caractère barre oblique inversée possède également un signification particulière en C++. On l'appelle le caractère d'échappement. Il permet d'entrer un caractère spéciale, comme les tabulations ou les retours à la ligne (comme vu dans les chapitres précédents). Ce caractère d'échappement s'associe avec le caractère qui le suit pour ne former qu'un seul caractère. Ainsi, '\n' ou '\n' ne sont pas deux caractères (\ puis n) mais bien un seul (si vous essayer d'écrire '/n', vous obtiendrez une erreur, puisqu'il n'est pas possible de mettre deux caractères dans une littérale caractère. Par contre '\n' ne pose pas de problème puisque c'est considéré comme un seul caractère).

Pour écrire le caractère `\`, il faut donc écrire `\\` en C++, ce qui fait que le motif `“\.”` devient `“\\.”` en C++. Cette chaîne doit être lue de la façon suivante : le premier `“\”` correspond au caractère d'échappement, donc `“\\”` correspond au caractère `“\”` dans le motif, et donc le motif `“\.”` permet de rechercher un point.

Une autre solution en C++, pour éviter de devoir utiliser les caractères d'échappement, est d'utiliser les littérales chaînes brutes (*raw string*). Dans ce cas, les caractères spéciaux du C++ (`\` ou `”` par exemple) sont interprété comme des caractères normaux. Pour écrire une littérale chaîne brute, il faut remplacer `”…”` par `R”(…)”`.

Ainsi, au lieu d'écrire `“\\.”`, il est possible d'écrire `R”(\.)”`. le code devient alors :

main.cpp

```cpp
#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::regex const pattern { R"(\.)" };
    std::string target { "cette phrase contient un point" };
    std::cout << target << " : " << std::boolalpha <<
        std::regex_search(target, pattern) << std::endl;

    target = "cette phrase contient un point.";
    std::cout << target << " : " << std::boolalpha <<
        std::regex_search(target, pattern) << std::endl;
}
```

Ce code affiche le résultat attendu :

```
cette phrase contient un point : false
cette phrase contient un point. : true
```

Les caractères spéciaux des expressions régulières, qu'il faut ajouter `\` sont les suivants :

Caractère utilisé dans les motifs	Caractère recherché dans la séquence cible
`\^`	`^`
`\$`	`$`
`\\`	`\`
`\.`	`.`
`\*`	`*`
`\+`	`+`
`\?`	`?`
`\(`	`(`
`\)`	`)`
`\[`	`[`
`\]`	`]`
`\{`	`{`
`\}`	`}`