[Aller plus loin] Les expressions régulières 2

Caractère générique, classe de caractères et ensemble de caractères

Chaque caractère de base correspond à un seul caractère dans la séquence terminale, mais il est parfois nécessaire de pouvoir exprimer dans un motif que l'on souhaite n'importe quel caractère issu d'un ensemble. Par exemple, pour créer un motif permettant de valider une date, il faut pouvoir écrire que l'on souhaite avoir n'importe quelle chiffre. Peu importe le chiffre, du moment que c'est un chiffre. Les caractères génériques, classes de caractères et ensembles de caractères permettent d'écrire cela.

Le caractère générique correspond au point dans une expression régulière. Il signifie “n'importe quel caractère”, mais uniquement un seul caractère. Par exemple, le motif “ab.de” peut correspondre aux chaînes `“abcde”` ou `“ab$de”`, mais pas à la chaîne `“ab$$de”`.

Ceci explique pourquoi, dans le code précédent, le motif `”.”` était retrouvé dans n'importe quelle chaîne, puisque cela signifie simple “trouve un caractère quelconque dans la chaîne”. Donc seule la chaîne vide `””` retourne faux avec ce motif.

main.cpp

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

int main()
{
```

```
    std::regex pattern { "." };
    std::cout << "'.' match with '': " << std::boolalpha <<
        std::regex_match("", pattern) << std::endl;

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

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

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

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

    pattern = "a.c";
    std::cout << "'a.c' match with 'abc': " << std::
boolalpha <<
        std::regex_match("abc", pattern) << std::endl;

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

affiche :

```
'.' match with '': false
'.' match with 'a': true
'.' match with 'ab': false
'abc' match with 'abc': true
'abc' match with 'a$c': false
'a.c' match with 'abc': true
'a.c' match with 'a$c': true
```

On voit donc bien que la chaîne correspond au motif uniquement si elle

ne contient un et un seul caractère, pas plus, pas moins.

Les ensembles de caractères (*character set*) permettent de représenter une liste de caractères entre crochets droits `[]`. Chaque ensemble permet de remplacer un et un seul caractère. Par exemple, le motif "[abc]" représente un seul caractère, qui peut être a, b ou c.

main.cpp

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

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

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

    std::cout << "'[abc]' match with 'b': " << std::
boolalpha <<
        std::regex_match("b", pattern) << std::endl;

    std::cout << "'[abc]' match with 'z': " << std::
boolalpha <<
        std::regex_match("z", pattern) << std::endl;

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

affiche :

```
'[abc]' match with '': false
'[abc]' match with 'a': true
'[abc]' match with 'b': true
```

```
'[abc]' match with 'z': false
'[abc]' match with 'ab': false
```

Si l'on souhaite écrire un motif qui permet de valider n'importe quelle lettre minuscule ou n'importe quel chiffre, il est fastidieux d'écrire tous les caractères correspondant : ”[abcdefghijklmnopqrstuvwxyz]” et ”[0123456789]”. Pour éviter cela, il est possible de spécifier une plage de valeur en indiquant le premier et le dernier caractère de la plage, séparés par un tiret. Par exemple : ”[a-z]” ou ”[0-9]”.

main.cpp

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

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

    std::cout << "'[a-e]' match with 'a': " << std::
boolalpha <<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'[a-e]' match with 'e': " << std::
boolalpha <<
        std::regex_match("e", pattern) << std::endl;

    std::cout << "'[a-e]' match with 'z': " << std::
boolalpha <<
        std::regex_match("z", pattern) << std::endl;

    std::cout << "'[a-e]' match with 'ab': " << std::
boolalpha <<
        std::regex_match("ab", pattern) << std::endl;

    pattern = "ab[c-f]";
    std::cout << "'ab[c-f]' match with '': " << std::
boolalpha <<
```

```
        std::regex_match("", pattern) << std::endl;

    std::cout << "'ab[c-f]' match with 'abc': " << std::
boolalpha <<
        std::regex_match("abc", pattern) << std::endl;

    std::cout << "'ab[c-f]' match with 'abz': " << std::
boolalpha <<
        std::regex_match("abz", pattern) << std::endl;
}
```

affiche :

```
'[a-e]' match with '': false
'[a-e]' match with 'a': true
'[a-e]' match with 'e': true
'[a-e]' match with 'z': false
'[a-e]' match with 'ab': false
'ab[c-f]' match with '': false
'ab[c-f]' match with 'abc': true
'ab[c-f]' match with 'abz': false
```

Il est possible de spécifier plusieurs plages dans un ensemble, par exemple "[a-er-z]" correspond à un caractère qui peut être a, b, c, d, e, r, s, t, u, v, w, x, y ou z.

Certains ensembles de classes sont souvent utilisées, il existe donc des raccourcis pour éviter de les réécrire à chaque fois. Ce sont les classes de caractères (*character class*). Par exemple, le motif `[[:alpha:]]` correspond à n'importe quel caractère alphabétique, donc est équivalent à `[a-zA-Z]`. Le tableau suivant présente l'ensemble des classes de caractères possible :

Classe de caractères	Description	Ensemble équivalent
`[:alnum:]`	Caractères alphanumériques	`[a-zA-Z0-9]`
`[:alpha:]`	Caractères alphabétiques	`[a-zA-Z]`
`[:ascii:]`	Caractères ASCII	`[\x00-\x7F]`

`[:blank:]`	Espace et tabulation	`[ \t]`
`[:cntrl:]`	Caractères de contrôle	`[\x00-\x1F\x7F]`
`[:digit:]`	Chiffres	`[0-9]`
`[:graph:]`	Caractères visibles	`[\x21-\x7E]`
`[:lower:]`	Caractères minuscules	`[a-z]`
`[:print:]`	Caractères visibles et espace	`[\x20-\x7E]`
`[:punct:]`	Ponctuation et symboles	`[!”#$%&'()*+,\-./:;⇔?@[\\\]^_`{|}~]`
`[:space:]`	Caractères blancs	`[ \t\r\n\v\f]`
`[:upper:]`	Caractères majuscules	`[A-Z]`
`[:word:]`	Lettre, chiffre ou tiret bas	`[A-Za-z0-9_]`
`[:xdigit:]`	Chiffres hexadécimaux	`[A-Fa-f0-9]`

Ces classes de caractères permettent de simplifier l'écriture des motifs.

main.cpp

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

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

    std::cout << "'[[:alpha:]]' match with '1': " << std::
boolalpha <<
```

```
        std::regex_match("1", pattern) << std::endl;

    std::cout << "'[[:alpha:]]' match with '&': " << std::
boolalpha <<
        std::regex_match("&", pattern) << std::endl;

    pattern = "[[:alnum:]]";
    std::cout << "'[[:alnum:]]' match with 'a': " << std::
boolalpha <<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'[[:alnum:]]' match with '1': " << std::
boolalpha <<
        std::regex_match("1", pattern) << std::endl;

    std::cout << "'[[:alnum:]]' match with '&': " << std::
boolalpha <<
        std::regex_match("&", pattern) << std::endl;
}
```

affiche :

```
'[[:alpha:]]' match with 'a': true
'[[:alpha:]]' match with '1': false
'[[:alpha:]]' match with '&': false
'[[:alnum:]]' match with 'a': true
'[[:alnum:]]' match with '1': true
'[[:alnum:]]' match with '&': false
```

Il est possible de prendre la négation d'une classe de caractères, c'est-à-dire de pouvoir accepter tous les caractères sauf ceux de la classe, en utilisant le symbole `^`. Par exemple, `[^[:alpha:]]` signifie “tous les caractères non alphabétiques” :

main.cpp

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

int main()
{
    std::regex const pattern { "[^[:alpha:]]" };
```

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

    std::cout << "'[^[:alpha:]]' match with '1': " << std::
boolalpha <<
        std::regex_match("1", pattern) << std::endl;

    std::cout << "'[^[:alpha:]]' match with '&': " << std::
boolalpha <<
        std::regex_match("&", pattern) << std::endl;
}
```

affiche :

```
'[[:alpha:]]' match with 'a': false
'[[:alpha:]]' match with '1': true
'[[:alpha:]]' match with '&': true
```

Il est possible d'ajouter des caractères à une classe de caractères, en les spécifiant entre les premiers crochets droits. Par exemple, l'ensemble `[123[:alpha:]]` permet de représenter tous les caractères alphabétiques, ainsi que les caractères 1, 2 et 3.

main.cpp

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

int main()
{
    std::regex const pattern { "[123[:alpha:]]" };
    std::cout << "'[123[:alpha:]]' match with 'a': " << std
::boolalpha <<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'[123[:alpha:]]' match with '1': " << std
::boolalpha <<
        std::regex_match("1", pattern) << std::endl;

    std::cout << "'[123[:alpha:]]' match with '4': " << std
::boolalpha <<
```

```
        std::regex_match("4", pattern) << std::endl;

    std::cout << "'[123[:alpha:]]' match with '&': " << std
::boolalpha <<
        std::regex_match("&", pattern) << std::endl;
}
```

affiche :

```
'[123[:alpha:]]' match with 'a': true
'[123[:alpha:]]' match with '1': true
'[123[:alpha:]]' match with '4': false
'[123[:alpha:]]' match with '&': false
```

Pour terminer avec les classes de caractères, il existe une écriture simplifiée pour certaines classes. Ces écritures simplifiés s'écrivent avec la barre oblique inversée `\` suivi d'un caractère. Lorsque le caractère est minuscule, cela correspond à une classe de caractères. Lorsqu'il est en majuscule, cela correspond à l'inverse de la classe de caractères correspondante. Le tableau suivant liste l'ensemble des écritures simplifiées :

Écriture simplifiée	Classe de caractères
`\d`	`[ [:digit:]]`
`\D`	`[^[:digit:]]`
`\s`	`[ [:space:]]`
`\S`	`[^[:space:]]`
`\w`	`[_[:alnum:]]`
`\W`	`[^_[:alnum:]]`

N'oubliez pas qu'il faut ajouter une barre oblique inversée en C++ pour échapper le caractère `\` ou utiliser les littérales chaînes brutes. Par exemple, pour écrire `\d`, il faut écrire en C++ :

```
std::regex const pattern { "\\d" };
// ou
std::regex const pattern { R"(\d)" };
```

main.cpp

```
#include <iostream>
```

```cpp
#include <regex>

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

    std::cout << R"('\w' match with '1': )" << std::
boolalpha <<
        std::regex_match("1", pattern) << std::endl;

    std::cout << R"('\w' match with '&': )" << std::
boolalpha <<
        std::regex_match("&", pattern) << std::endl;
}
```

affiche :

```
'\w' match with 'a': true
'\w' match with '1': true
'\w' match with '&': false
```

Les répétitions

Jusqu'à maintenant, les syntaxes que l'on a vu ne permettent de remplacer qu'un seul caractère. Si l'on veut écrire un motif correspondant à trois lettres, il faudra écrire : `[ [:alpha:]][ [:alpha:]][ [:alpha:]]`. C'est un peu lourd à écrire (surtout si l'on veut 20 lettres par exemple) et cela ne permet pas de spécifier des motifs comme “5 à 10 lettres” ou “tous les caractères jusqu'au premier point”.

Une répétition permet de répéter un caractères ou un groupe de caractères. Il existe plusieurs répétitions :

Symbole	Écriture	Répétition
`*`	`a*`	zéro ou plus

`+`	`a+`	un ou plus
`?`	`a?`	zéro ou un
`{n}`	`a{3}`	n répétitions
`{n,}`	`a{3,}`	n ou plus répétitions
`{n,m}`	`a{3,5}`	entre n et m répétitions

La première répétition `*` permet de répéter zéro ou plusieurs fois un caractère. Par exemple, le motif `a*` correspond à une chaîne vide `""` ou une chaîne contenant un nombre quelconque de caractère `a` (par exemple `"aaa"` ou `"aaaaaa"`), mais aucun autre caractère.

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 'aaaa': " << std::
boolalpha <<
        std::regex_match("aaaa", pattern) << std::endl;

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

affiche :

```
'a*' match with '': true
'a*' match with 'a': true
'a*' match with 'aaaa': true
'a*' match with 'abcd': false
```

La répétition `+` permet de répéter une ou plusieurs fois un caractère. Le motif `a+` est similaire au motif précédent, mais interdit la chaîne vide :

main.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 'aaaa': " << std::
boolalpha <<
        std::regex_match("aaaa", pattern) << std::endl;

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

affiche :

```
'a+' match with '': false
'a+' match with 'a': true
'a+' match with 'aaaa': true
'a+' match with 'abcd': false
```

La répétition `?` permet de répéter zéro ou une fois un caractère. Le motif `“a?”` accepte donc uniquement la chaîne vide et la chaîne `“a”`.

main.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 'aaaa': " << std::
boolalpha <<
        std::regex_match("aaaa", pattern) << std::endl;
}
```

affiche :

```
'a?' match with '': true
'a?' match with 'a': true
'a?' match with 'aaaa': false
```

Les répétitions avec les crochets s'appellent des répétitions bornées (*bounded repeat*). La première permet de spécifier le nombre exact de répétitions du caractère, la seconde le nombre minimal de répétitions et la dernière les nombres minimal et maximal de répétitions.

pourquoi je m'amuse à recopier le std::boolalpha à chaque cout ??

main.cpp

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

int main()
{
    std::regex pattern { "a{3}" };
    std::cout << "'a{3}' match with '': " << std::boolalpha
<<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'a{3}' match with 'aaaa': " << std::
boolalpha <<
        std::regex_match("aaaa", pattern) << std::endl;
```

```
    std::cout << "'a{3}' match with 'aaaaaaaa': " << std::
boolalpha <<
        std::regex_match("aaaaaaaa", pattern) << std::endl
<< std::endl;

    pattern = "a{3,}";
    std::cout << "'a{3,}' match with '': " << std::boolalpha
<<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'a{3,}' match with 'aaaa': " << std::
boolalpha <<
        std::regex_match("aaaa", pattern) << std::endl;

    std::cout << "'a{3,}' match with 'aaaaaaaa': " << std::
boolalpha <<
        std::regex_match("aaaaaaaa", pattern) << std::endl
<< std::endl;

    pattern = "a{3,5}";
    std::cout << "'a{3,5}' match with '': " << std::
boolalpha <<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'a{3,5}' match with 'aaaa': " << std::
boolalpha <<
        std::regex_match("aaaa", pattern) << std::endl;

    std::cout << "'a{3,5}' match with 'aaaaaaaa': " << std::
boolalpha <<
        std::regex_match("aaaaaaaa", pattern) << std::endl;
}
```

affiche :

```
'a{3}' match with '': false
'a{3}' match with 'aaaa': false
'a{3}' match with 'aaaaaaaa': false

'a{3,}' match with '': false
'a{3,}' match with 'aaaa': true
```

```
'a{3,}' match with 'aaaaaaaa': true

'a{3,5}' match with '': false
'a{3,5}' match with 'aaaa': true
'a{3,5}' match with 'aaaaaaaa': false
```

Les ancres

Les ancres (*anchor*) permettent d'attacher un motif au début (avec `^`) ou à la fin (avec `$`) de la séquence cible. Elles servent uniquement (principalement ?) pour les recherches. Ainsi, le motif `^a` recherche le caractère `a` au début d'une chaîne, alors que le motif `a$` le recherche à la fin.

main.cpp

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

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

    std::cout << "'^a' match with 'abcd': " << std::
boolalpha <<
        std::regex_search("abcd", pattern) << std::endl;

    std::cout << "'^a' match with 'dcba': " << std::
boolalpha <<
        std::regex_search("dcba", pattern) << std::endl <<
std::endl;

    pattern = "a$";
    std::cout << "'a$' match with '': " << std::boolalpha <<
        std::regex_search("", pattern) << std::endl;

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

```
    std::cout << "'a$' match with 'dcba': " << std::
boolalpha <<
        std::regex_search("dcba", pattern) << std::endl;
}
```

affiche :

```
'^a' match with '': false
'^a' match with 'abcd': true
'^a' match with 'dcba': false

'a$' match with '': false
'a$' match with 'abcd': false
'a$' match with 'dcba': true
```

Chapitre précédent	Sommaire principal	Chapitre suivant