Comparer si deux chaînes sont identiques

Les chapitres précédents présentaient la syntaxe des expressions régulières. Bien sûr, il faudra pratiquer pour assimiler correctement ce langage. Dans ce chapitre et les suivants, nous allons voir plus en détail l'utilisation des expressions régulières, mais côté C++ cette fois ci. Nous verrons également l'utilisation de plusieurs algorithmes de la bibliothèque standard.

Les opérateurs de comparaison

Dans les chapitres précédents, vous avez vu l'utilisation de l'opérateur de comparaison `==` (*"egal to" operator*). La classe `string` fournit également cet opérateur, vous pouvez donc écrire :

main.cpp

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

int main() {
    std::string const s1 { "salut" };
    std::string const s2 { "salut" };
    std::string const s3 { "hello" };
    std::cout << std::boolalpha << (s1 == s2) << std::endl;
    std::cout << (s1 == s3) << std::endl;
}
```

affiche :

```
true
false
```

Remarque : ne pas oublier les parenthèses autour du test d'égalité pour

`string`. L'opérateur `«` ayant un sens pour cette classe, le compilateur ne pourra pas savoir si vous souhaitez écrire :

```
std::cout << std::boolalpha << (s1 == s2) << std::endl;
std::cout << std::boolalpha << s1 == (s2 << std::endl);
```

Pour vérifier si deux chaînes sont différentes, il est possible d'utiliser l'opérateur booléen de négation `!` : `!(s1 == s2)`. Pour simplement, on peut directement utiliser l'opérateur de comparaison `!=` (*“not equal to” operator*).

main.cpp

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

int main() {
    std::string const s1 { "salut" };
    std::string const s2 { "salut" };
    std::string const s3 { "hello" };
    std::cout << std::boolalpha << (s1 != s2) << std::endl;
    std::cout << (s1 != s3) << std::endl;
}
```

affiche :

```
false
true
```

On voit ici que les opérateurs `==` et `!=` sont étroitement liés. Lorsque l'un de ces deux opérateurs est utilisable, on peut s'attendre à ce que l'autre opérateur le soit également. On dit qu'une classe qui propose l'opérateur d'égalité `==` qu'elle respecte le concept de “comparable par égalité” (EqualityComparable). Ce concept précise que l'opérateur d'égalité doit suivre les propriétés suivantes :

- réflexivité : quelque soit `a`, `a == a` est toujours vrai ;
- commutativité : si `a == b`, alors `b == a` ;
- transitivité : si `a == b` et `b == c`, alors `a == c`.

Ce concept est assez classique, on le retrouve en mathématique dans la

théorie des ensembles. On voit ici un point important : lorsqu'une classe définit un opérateur `==`, on s'attend à ce qu'elle suive un certain nombre de règles. On dit qu'elle suis une sémantique, cela facilite son utilisation. Du point de vue de l'utilisateur de cette classe, on pourra utiliser n'importe quelle classe respectant cette sémantique de la même façon. Du point de vue du concepteur de la classe (ce que vous apprendrez à faire dans la suite de ce cours), il suffit de définir les sémantiques que l'on souhaite donner à notre classe et l'écriture de cette classe sera simplifié.

Au contraire, le non respect d'une sémantique sera très perturbant pour l'utilisateur - et une source d'erreur sans fin. Imaginez que l'opérateur `==` ne réalise pas un test d'égalité, mais permet de faire la concaténation de deux chaînes ?

Bien sûr, ces considérations s'appliquent à l'ensemble des sémantiques usuelles, en particulier celle que l'on connait en mathématique (addition avec `+`, soustraction avec `-`, etc.)

La sémantique de valeur

Un concept complexe peut être décomposé en une série de concepts plus simple (par exemple “est comparable par égalité” est composé des concepts “est réflexif”, “est commutatif” et “est transitif”). De la même façon, il est possible de combiner des concepts pour créer de nouveaux concepts plus complexe. Un concept peut également autorisé ou interdire l'utilisation d'autres concepts (par exemple, le concept “est égal” autorise l'utilisation du concept “est différent”).

C'est la cas du concept “est comparable par égalité”, qui fait parti d'un concept plus général : la sémantique de valeur. Cette sémantique s'applique à tout ce qui représente une valeur : un entier, un nombre réel, un nombre complexe, une chaîne de caractères, un tableau de données, etc. La sémantique de valeur autorise les concepts suivants :

Constructible par défaut (DefaultConstructible). On peut initialiser avec une valeur par défaut (on parle aussi de “zero initialization” puisque la valeur par défaut sera 0 ou équivalent).

```
int const i {}; // construction par défaut
```

Copiable par construction (CopyConstructible) et par affectation (CopyAssignable). Cela signifie que l'on peut créer une valeur en copiant une autre valeur. Par exemple, pour un entier :

```
int const i { 123 };
int j { i }; // construction par copie
j = i;       // affectation par copie
```

Comparable par égalité (EqualityComparable) ou par “plus petit que” (LessThanComparable). Cela signifie que l'on peut utiliser les opérateurs d'égalité `==` et “plut petit que” `<` (ainsi que les opérateurs dérivés : “différent de” `!=`, “plus petit ou égal à” `<=`, “plus grand que” `>` et “plus grand ou égal à” `>=`) :

```
int const i { 123 };
int const j { 456 };
cout << (i == j) << endl; // égalité
cout << (i != j) << endl; // différent
cout << (i < j) << endl;  // plus petit
cout << (i <= j) << endl; // plus petit ou égal
cout << (i > j) << endl;  // plus grand
cout << (i >= j) << endl; // plus grand ou égal
```

Si cela a un sens, la sémantique de valeur permet également de définir des opérateurs arithmétiques classiques : addition `+`, soustraction `-`, multiplication `*` et division `/`. Par exemple, pour les entiers :

```
int const i { 123 };
int const j { 456 };
cout << (i + j) << endl; // addition
cout << (i - j) << endl; // soustraction
cout << (i * j) << endl; // multiplication
cout << (i / j) << endl; // division
```

Le rôle de ces opérateurs peut varier en fonction du type. Par exemple, l'opérateur `+` correspondra à une addition pour les types numériques et à une concaténation pour les chaînes de caractères `string`.

main.cpp

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

int main() {
    int const i1 { 1 };
    int const i2 { 2 };
    std::cout << (i1 + i2) << std::endl; // addition
    std::string const s1 { "1" };
    std::string const s2 { "2" };
    std::cout << (s1 + s2) << std::endl; // concaténation
}
```

affiche :

```
3
12
```

De plus, selon les types, tous les opérateurs n'ont pas forcement un sens. Par exemple, pour les chaînes `string`, seul l'opérateur `+` a un sens, les autres opérateurs ne sont pas définis.

Les deux grands types de classes sont les classes à sémantique de valeur (que vous avez vu dans ce chapitre) et les classes à sémantique d'entité. Vous apprendrez dans la partie “programmation orientée objet” comment créer ces types de classes.

Le type de chaîne char*

Comme vous l'avez vu dans le chapitre sur les littérales, les littérales chaînes de caractères ne sont pas de type `string`, mais de type `const char*`. Ce type est un héritage du C++ historique et du langage C. Pourquoi ne pas utiliser ce type en C++ ?

Faisons un test simple. Essayons de comparer deux chaînes :

main.cpp

```cpp
#include <iostream>

int main() {
    std::cout << std::boolalpha << ("b" < "a") << std::endl;
}
```

Ce code affiche :

```
main.cpp: In function 'int main()':
main.cpp:4:43: warning: comparison with string literal
results in unspecified behaviour [-Waddress]
     std::cout << std::boolalpha << ("b" < "a") << std::endl;
                                            ^
true
```

Premier problème, le compilateur affiche un message d'avertissement pour prévenir que la comparaison de littérales chaînes produit un comportement indéterminé (astuce : si vous ne comprenez pas bien l'anglais, n'hésitez pas à vous servir d'un traducteur en ligne comme Google Translate ou de faire une recherche sur internet en copiant le message d'erreur). On parle de comportement indéterminé (*Undefined Behavior* ou *UB*) lorsque le comportement n'est pas défini dans la norme C++. Donc ce code pourra fonctionner différemment selon le compilateur, donner le résultat correct ou un résultat aléatoire, produire une erreur, etc. Dans l'idée d'écrire du code C++ moderne (donc de qualité), on évitera bien sûr d'écrire du code qui produit un comportement indéterminé.

Le second problème est que le résultat est vrai, alors que la lettre b est supérieur (dans l'ordre alphabétique) à la lettre a. Vous pouvez essayer avec n'importe quelle lettre (avec Clang sur Coliru), le résultat sera toujours vrai. L'explication de ce comportement nécessite de comprendre le fonctionnement des pointeurs (ce qui sort du cadre de ce cours débutant), mais le principal est de comprendre que cela ne donne pas le résultat attendu.

Le code similaire avec `string` ne posera pas de problème :

main.cpp

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

int main() {
    std::string const s1 { "b" };
    std::string const s2 { "a" };
    std::cout << std::boolalpha << (s1 < s2) << std::endl;
}
```

affiche :

```
false
```

En effet, la classe `string` implémente l'opérateur `<` en respectant la sémantique habituelle de cet opérateur (alors que le type `char*` utilise la sémantique des pointeurs). L'utilisation de `char*` est donc a éviter en C++ moderne.

Le code précédent fonctionne aussi si l'on déclare qu'une seule variable de type `string` :

main.cpp

```
std::string const s { "b" };
std::cout << std::boolalpha << (s < "a") << std::endl;
```

Dans ce cas, c'est bien la sémantique de `string` qui est utilisée, pas celle de `char*`. Pourquoi le compilateur utilise correctement l'opérateur `<` de `string` et le le fait pas lorsque l'on écrit `“b” < “a”` ?

Le compilateur procède de la façon suivante :

- Il commence par rechercher s'il existe un opérateur `<` qui correspond aux types utilisés. Donc `string` et `char*` dans un cas et deux `char*` dans l'autre cas. Cet opérateur existe pour les deux `char*` (mais s'applique sur les pointeurs, pas sur le contenu de la chaîne de caractères) et est donc utilisé. Dans les cas de `string` et `char*`, aucun opérateur ne convient.
- S'il n'existe pas d'opérateur, le compilateur essaie de convertir

l'un des types pour trouver un opérateur `<` qui existe. Toutes les types ne sont pas convertissables dans n'importe quel autre type. Chaque classes spécifient les conversions qui sont autorisée ou non. Elles spécifient également si la conversion peut être implicite (le compilateur décide tout seul s'il réalise la conversion ou non) ou explicite (l'utilisateur doit écrit spécifiquement la conversion).

Dans le cas qui nous intéresse, il existe une conversion implicite de `char*` en `string`. Cela signifie que lorsque le compilateur rencontre un type `char*`, il est autorisé à le convertir en `string` sans demander l'autorisation à l'utilisateur. Le compilateur résout donc le code `s < “a”` en convertissant la littérale chaîne en `string`, puis appelle l'opérateur `<` qui s'applique sur deux `string`. Le code `s < “a”` est donc interprété par le compilateur de la même façon que le code `s1 < s2`.

Il est également possible d'écrire explicitement la conversion d'un type dans un autre en initialisant avec le type. Par exemple :

```
std::string const s { "b" };
std::cout << std::boolalpha << (s < std::string { "a" }) <<
std::endl;
// ou
std::cout << std::boolalpha << (std::string { "b" } < "a")
<< std::endl;
```

Vous apprendrez dans la partie sur la programmation orientée objet comment créer des classes autorisant les conversions implicites et explicites.

Chapitre précédent	Sommaire principal	Chapitre suivant

Cours, C++