raw string litteral

Initialisation, concaténation et conversion des chaînes

Fondamentalement, l'informatique englobe tout ce qui concerne le traitement de l'information. Vous avez vu dans les chapitres précédents les bases du calcul numérique. Un autre type de données qui souvent manipulé en informatique sont les chaînes de caractères. Les chapitres suivants détaillent leurs manipulations en C++.

Les fonctions membres et non membres

parler plus tôt des fonctions. Et mieux expliquer, parce que là, c'est caca...

partie trop détaillée et pas à sa place

Les fonctions sont des traitements spécifiques que l'on peut appliquer sur les données. Vous avez déjà utilisé de telles fonctions dans les chapitres précédents, comme par exemple les fonctions mathématiques ou la fonction `size` pour connaître la taille d'une chaîne de caractères `string`.

Les fonctions peuvent s'écrire selon deux syntaxes différentes. Par exemple :

```
double const d { 123.456 };
std::cout << std::exp(d) << std::endl; // utilisation de la fonction exp

std::string s{ "hello, world!" };
std::cout << s.size() << std::endl; // utilisation de la fonction size
```

Le premier type de fonction, représenté par la fonction mathématique exponentiel `exp`, s'appelle une fonction non membre. Le service qu'elle rend (calculer l'exponentielle d'un nombre réel) s'applique à la valeur qui est passée en argument entre les parenthèses.

Le résultat du calcul est retourné directement par la fonction. Cela signifie que lors de l'évaluation de la fonction, la fonction est remplacée par le résultat du calcul. Ainsi, le code suivant :

```
double const result { std::exp(1.0) };
std::cout << std::exp(1.0) << std::endl;
```

sont équivalent au code suivant, après évaluation de la fonction `exp` :

```
double const result { 2.71828182845905 };
std::cout << 2.71828182845905 << std::endl;
```

Chaque fonction possède une signature spécifique, qui définit son nom, le nombre d'arguments qu'elle prend et leur type et le type de valeur qu'elle retourne. Une fonction peut également ne pas prendre d'argument ou ne pas retourner de valeur.

Le second type de fonction s'appelle une fonction membre et s'applique à un objet spécifique. Un exemple d'une telle fonction est la fonction `size` de la classe `string` :

```
std::string s{ "hello, world!" };
std::cout << s.size() << std::endl; // utilisation de la
fonction size
```

Ce type de fonction permet d'obtenir des informations sur un objet (c'est le cas de `size`) ou de modifier cet objet. Un par exemple de fonction qui modifie un objet pourrait être la fonction `clear`, qui efface une chaîne de caractères.

```
std::string s{ "hello, world!" };
s.clear(); // on efface s
std::cout << s << std::endl;
```

Ce code n'affiche rien, puisque la variable `s` ne contient plus aucune

chaîne après appel de la fonction `clear`.

Une fonction membre peut également prendre aucun, un ou plusieurs arguments et retourner ou non une valeur.

Une fonction membre ne peut pas être appelée comme une fonction non membre. Si vous essayer d'appeler la fonction `size` seule ou en lui passant une chaîne en argument, cela produira une erreur :

```
size(s); // erreur
```

Cependant, dans certain cas, il existe une fonction membre et une fonction non membre qui possède ne même nom. C'est par exemple le cas de la fonction `begin`, qui vous verrez par la suite. Vous pouvez donc écrire :

```
string s { "hello, world!" };
s.begin();
begin(s);
```

Les deux lignes avec `begin` sont équivalentes et font exactement la même chose. Par contre, n'oubliez pas qu'il s'agit bien de deux fonctions différentes (cela sera important lorsque vous écrirez vous même vos propres fonctions si c'est pas important maintenant, faut il en parler maintenant ?).

Dans tous les cas, il ne faut pas hésiter à se référer à la documentation, pour connaître la signature de chaque fonction.

Exercices

Lire la documentation de quelques fonctions, membre et non membre. Ecrire le code correcte pour utiliser ces fonctions, meme si on les connait pas, en fonction de la doc.

Initialisation et assignation d'une chaîne

Les chaînes de caractères en C++ sont manipulées via la classe `std::string` de la bibliothèque standard. L'initialisation ou la modification d'une chaînes se fait en utilisant l'initialisation avec des crochets et l'opérateur d'affectation `=` :

main.cpp

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

int main() {
    std::string const s1 { "hello, world!" }; //
initialisation avec une littérale
    std::string s2 {};                        //
initialisation par défaut
    s2 = "bonjour tout le monde !";           // affectation

    std::cout << "hello, world!" << std::endl; // afficher
une littérale
    std::cout << s2 << std::endl;              // afficher
une variable
}
```

Un point important concernant les littérales chaînes de caractères : elles ne sont pas de type `string`, mais du type `const char *` hérité du C. L'utilisation de ce type n'est pas recommandé, sauf pour initialiser une chaîne de type `string`. La classe `string` apporte des garanties plus forte que `const char *` (fuite mémoire) et offre beaucoup plus de fonctionnalités (que vous avez voir dans la suite de ce chapitre et dans les chapitres suivants).

Les littérales string

Dans la prochaine norme du C++14, il sera possible de créer directement une littérale chaîne de caractères de type `string`, en ajoutant le suffixe “s” après la chaîne. Cela permettra d'éviter les conversions de `'const char *` en `string` et permettra d'utiliser directement `auto` :

```
string s1 { "hello, world!"s };
auto s2 = "hello, world!"s;
```

La concaténation

Concaténation de chaînes de type string

En informatique, la concaténation est simplement l'opération qui consiste à créer une chaîne en associant plusieurs chaînes misent bout à bout. Par exemple, la chaîne “hello world” est la concaténation des chaînes “hello” et ” world”.

Dans de nombreux langages, l'opérateur `+` appliqué sur des chaînes permet de réaliser la concaténation. C'est également le cas pour les chaînes de type `string` en C++ :

main.cpp

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

int main()
{
    std::string const s1 { "hello" };
    std::string const s2 { " world" };
    std::string const s3 { s1 + s2 }; // concaténation de s1 et s2
    std::cout << s3 << std::endl;
}
```

affiche :

```
hello world
```

Il est possible de concaténé autant de chaînes que l'on souhaite de cette manière :

main.cpp

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

int main()
{
    std::string const s1 { "Bonjour" };
    std::string const s2 { " tout" };
    std::string const s3 { " le monde" };
    std::string const s4 { s1 + s2 + s3 };
    std::cout << s4 << std::endl;
}
```

affiche :

```
bonjour tout le monde
```

La concaténation est utilisable bien sûr lors de l'initialisation (comme ci-dessus), mais également pour modifier une chaîne avec l'opérateur d'affectation `=` :

main.cpp

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

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

Il est également possible de modifier une chaîne en ajoutant d'autres chaînes à la suite. La méthode évidente est d'utiliser les opérateurs de concaténation `+` et d'affectation `=` :

main.cpp

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

```cpp
#include <string>

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

Cette opération, qui consiste à réaliser un calcul sur lui-même, est assez classique et le C++ fournit l'opérateur `+=` pour cela.

```cpp
s1 += s2; // est équivalent à s1 = s1 + s2;
```

Une autre approche est d'utiliser la fonction membre `append` (*ajouter*) pour la concaténation. La fonction `append` permet de réaliser plus de choses que l'opérateur `+`, vous verrez cela dans les exercices.

```cpp
s1.append(s2);
```

Concaténation d'un chaîne avec autre chose

Il est possible d'utiliser la concaténation avec d'autres types que les chaînes de type `string`, mais avec des restrictions. Il est possible d'ajouter un caractère à une chaîne par exemple :

main.cpp

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

int main()
{
    std::string const s1 { "hello" };
    char c { '!' };
    std::string const s2 { s1 + c };
    std::cout << s2 << std::endl;
}
```

affiche :

```
hello!
```

Il est également possible de concaténer une chaîne avec une littérale chaîne ou une littérale caractère :

main.cpp

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

int main()
{
    std::string s1 { "hello" };

    s1 += " world"; // concaténation d'une littérale chaîne
    s1 += '!';      // concaténation d'une littérale
caractère

    std::cout << s1 << std::endl;
}
```

affiche :

```
hello world!
```

En revanche, si vous essayer de concaténer plusieurs littérales chaînes ou caractère, le compilateur va produire une erreur. Par exemple :

main.cpp

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

int main()
{
    std::string const s1 { "hello" + " world" }; // erreur
    std::cout << s1 << std::endl;
}
```

produit l'erreur :

```
main.cpp:6:36: error: invalid operands to binary expression
('const char *' and 'const char *')
    std::string const s1 { "hello" + " world" };
                           ~~~~~~~ ^ ~~~~~~~~
1 error generated.
```

Si vous essayer avec une littérale caractère, le compilateur produit un avertissement, mais pas d'erreur. Pour lui, appliquer l'opérateur `+` sur une littérale chaîne et une littérale caractère a un sens. Mais ce n'est pas une concaténation.

```
std::string const s1 { "hello" + '!' };
```

affiche l'avertissement :

```
main.cpp:6:36: warning: adding 'char' to a string does not
append to the string [-Wstring-plus-int]
    std::string const s1 { "hello" + '!' };
                           ~~~~~~~~^~~~
main.cpp:6:36: note: use array indexing to silence this
warning
    std::string const s1 { "hello" + '!' };
                                     ^
                           &       [    ]
1 warning generated.
```

Pour comprendre ce problème, il faut se rappeler un point : les littérales chaînes de caractères ne sont pas de type `string`, mais de type `const char *`. Et ce type, hérité du C, ne propose tout simplement pas les mêmes fonctionnalités que le type `string` pour gérer les chaînes de caractères. Ce qui explique pourquoi le C++ a introduit le type `string` (en C, il existe des fonctions spécifique pour réaliser la concaténation, ce qui est plus lourd à utiliser que l'opérateur `+`).

Conversions entre chaînes et nombres

De la même façon, si vous avez essayer de concaténer une chaîne avec un nombre, vous avez obtenu un avertissement lors de compilation. Par exemple :

main.cpp

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

int main()
{
    std::string s1 { "hello " };
    s1 += 1234.56;
    std::cout << s1 << std::endl;
}
```

afficher le message :

```
main.cpp:7:11: warning: implicit conversion from 'double' to
'char' changes value
from 1234.56 to 127 [-Wliteral-conversion]
    s1 += 1234.56;
    ~~    ^~~~~~
hello 
```

Cette opération est acceptée par le compilateur, mais il convertie le nombre en un caractère de type `char` (pour rappel, le type `char` est un type de nombre entier interprété comme un caractère). Cette conversion implicite d'un nombre en `char` est autorisée, mais ne produit pas le résultat attendu.

Pour réaliser correctement la concaténation d'un nombre dans une chaîne, il faut dans un premier temps convertir le nombre en `string` en utilisant la fonction `std::to_string` :

```cpp
s1 += std::to_string(1234.56);
```

Après conversion, la chaîne est correctement affichée :

```
hello 1234.560000
```

Il est également possible de réaliser l'opération inverse, c'est-à-dire convertir une chaîne contenant un nombre en un nombre. Il y a cependant une différence importante : lorsque l'on utilise la fonction `to_string`, le compilateur sait, lors de la compilation, quel type de

nombre il doit convertir en chaîne :

```
double const d { 123.456 };
std::to_string(d); // conversion d'un double en string

int const i { 123456 };
std::to_string(i); // conversion d'un int en string
```

Dans la cas d'une conversion d'une chaîne `string` en un nombre, le compilateur n'a aucun moyen de déduire le type du nombre qu'il faut convertir à la compilation (il faudrait qu'il parcourt le contenu de la chaîne, ce qui est possible que lors de l'exécution). Il faut donc indiquer explicitement le type de nombre que l'on souhaite obtenir à partir d'une chaîne `string`, ce qui explique pourquoi il existe plusieurs fonctions pour convertir une chaîne en un nombre, alors qu'il existe qu'une seule fonction pour l'opération inverse.

La signature des fonctions de conversion suit un même modèle, il est relativement simple de se souvenir de la fonction à utiliser en fonction du type que l'on souhaite obtenir. Le nom des fonctions s'écrit :

- “s” pour `string` ;
- “to”, que l'on peut traduire par “en”, “vers” ;
- un identifiant pour le type : “i” pour `int`, “f” pour `float`, etc.

Le tableau suivant résume les différentes fonctions :

Fonction	Type du nombre	Identifiant de type
`stoi`	`int`	`i`
`stol`	`long int`	`l`
`stoll`	`long long int`	`ll`
`stoul`	`unsigned long int`	`ul`
`stoull`	`unsigned long long int`	`ull`
`stof`	`float`	`f`
`stod`	`double`	`d`
`stold`	`long double`	`ld`

main.cpp

```
#include <iostream>
```

```cpp
#include <string>

int main()
{
    std::string int_str { "123" };
    int const i { std::stoi(int_str) };
    std::cout << i << std::endl;

    std::string double_str { "123.456" };
    auto const d = std::stod(double_str);
    std::cout << d << std::endl;
}
```

affiche :

```
123
123.456
```

La chaîne à convertir peut contenir des espaces devant le nombre à convertir et n'importe quel caractère après le nombre sans que cela ne pose de problème de conversion. Par contre, si le nombre est précédé d'un caractère autre qu'un espace, cela produit une erreur d'exécution :

```cpp
std::string int_str { " 123abc" }; // ok
std::string int_str { "a123" };    // erreur
```

La seconde ligne produit l'erreur suivante (“argument invalide”) :

```
terminate called after throwing an instance of
'std::invalid_argument'
  what():  stoi
bash: line 8: 30337 Aborted                 (core dumped)
./a.out
```

Exercices

- utiliser les autres formes de la fonction `append`

Cours, C++