PG208
Programmation Orientée Objet / Langage C++
TP2

N. Ajmi Y. Bornat

January 22, 2025

1 Chaînes de caractères

Les chaînes de caractères étaient implémentées comme des tableaux de type `unsigned char` en C, il est donc logique qu'en C++, ce soit un conteneur équivalent. C'est effectivement le cas, mais pour en améliorer la lecture, un type spécial a été défini : il s'agit du type `std::string` accessible dans la bibliothèque `<string>`.

1.1 Détails

Ce que nous avons vu sur les conteneurs reste valable. Une variable de type `std::string` peut être utilisée avec toutes les techniques vues pour les vecteurs. Cela simplifie beaucoup de tâches de manipulation. Mais le plus pratique, c'est qu'il est maintenant possible de concaténer des `string`s de façon assez intuitive, avec l'opérateur '+'. Ainsi, pour dire bonjour au monde, il est possible d'écrire le code suivant, ou toute autre variante :

```
1   #include <iostream>
2   #include <string>
3   using namespace std::literals;
4
5   int main()
6   {
7       std::string mystring;
8       mystring = "Hello "s;
9       mystring += "World"s;
10    std::cout << mystring << std::endl;
11      return 0
12  }
```

1.2 Usage

En plus de toutes celles liées aux vecteurs, les chaînes de caractères sont associées aux fonctions suivantes :

- `.length()` : comme `.size()`, mais c'est plus cohérent de parler de longueur que de taille.
- `.find(str)` : si `str1` et `str2` sont deux chaînes de caractères, `str1.find(str2)` renvoie la position de `str2` dans `str1`. Éventuellement, il est possible d'ajouter la position à partir de où il faut chercher. La fonction est alors `.find(str, pos)`.
- `.find_first_of(str)` : `str1.find(str2)` cherche dans `str1` la première fois où on rencontre un caractère de `str2`. Comme pour `.find()`, on peut préciser une position de départ.
- `.find_last_of(str)` : `str1.find(str2)` cherche dans `str1` la dernière fois où on rencontre un caractère de `str2`. Tout pareil, on peut préciser une position de départ, mais cette fois-ci, on part de la fin de la chaîne.

- `.substr(pos, len)` : renvoie un extrait de la chaîne qui commence à la position `pos` et dont la longueur est `len`.
- `.at(n)` : si `str` est une chaîne de caractères, `str.at(n)` renvoie une référence vers `str[n]` (alors que `str[n]` renvoie une copie)

Malgré le grand intérêt que présente le type `std::string`, il arrive qu'il soit encore nécessaire d'utiliser la représentation du C (un pointeur vers une zone de données qui contient des caractères, puis le caractères nul pour marquer la fin). Pour différencier les représentations on utilise le terme *C-string* pour préciser qu'on manipule un `char *`. La conversion d'une *C-string* en `std::string` se fait de façon transparente (par exemple : `std::string Snew {C_string}` où la variable `C_string`) est de type `char *`. Mais pour l'inverse, on utilise les fonctions `.C_str()` et `.data()` qui sont équivalentes. Elles renvoient une copie de la chaîne de caractère au format *C-string*.

D'ailleurs, lorsqu'une chaïne de caractère est entrée sous forme de litéral (écrite en dur), elle est toujours implémentée sous forme de *C-string* pour assurer la compatibilité avec le C. Pour qu'un litéral soit implémenté sous forme de `std::string`, il faut lui ajouter le suffixe `s`, comme aux lignes 7 et 8 de l'exemple de code précédent. Cette notation nécessite d'inclure la ligne `using namespace std::literals;`.

Il existe également des fonctions plus classiques pour aider les conversions. Elles s'utilisent plus simplement avec une chaîne de caractère en argument.

- `std::stoi(str)` : convertit une chaine de caractère en entier (`int`)
- `std::stol(str)` : convertit une chaine de caractère en long entier (`long`)
- `std::stoll(str)` : convertit une chaine de caractère en `long long`
- `std::stoul(str)` : convertit une chaine de caractère en `unsigned long`
- `std::stoull(str)` : convertit une chaine de caractère en `unsigned long long`
- `std::stof(str)` : convertit une chaine de caractère en flottant (`float`)
- `std::stod(str)` : convertit une chaine de caractère en `double`
- `std::stold(str)` : convertit une chaine de caractère en `long double`

Pour chacune de ces fonctions, il est possible d'ajouter un pointeur vers un entier `std::stoi(str, &pos)`. Cet entier est alors mis à jour avec la position à laquelle s'arrête la valeur dans la chaîne. Si ce pointeur est null (0 ou `NULL`), il est ignoré.

Pour les conversions vers des types entiers, il est possible de spécifier un troisième argument de type entier. Il s'agit alors de la base de conversion. Il est ainsi possible de convertir directement des valeurs binaires, octales ou hexadécimales.

ATTENTION: Selon votre version de référence (C++17, C++11, ou précédent), la présence d'un caractère nul en fin de chaine n'est pas garantie, et même s'il est présent, l'interprétation d'un caractère nul comme fin de chaîne n'est pas garantie non plus. Il faut bien dissocier les usages propres à chaque type de donnée.

1.3 Exercice

Écrivez une fonction `calculatrice` qui convertit en flottant une chaine de caractères contenant des flottants reliés par les opérations +,-,*,/. Il n'est pas nécessaire de gérer les parenthèses, par contre, le nombre d'opérations est quelconque (pas juste une opération donc) et les priorités doivent être respectées.

Pour cela, le plus simple est d'utiliser la récursivité :

- On commence par chercher le caractère '*' ou '/'
- Si il a été trouvé
  - On extrait la sous-chaîne de caractères avant l'opération
  - On calcule de résultat de la sous-chaîne récursivement

  - On extrait la sous-chaîne de carcatères après l'opération
  - On calcule de résultat de la deuxième sous-chaîne récursivement
  - Il n'y a plus qu'à calculer l'opération avec les deux résultats
  - On retourne le résultat
- On cherche le caractère '+' ou '-'
- Si il a été trouvé, on effectue le même genre d'opération que précédemment
  - Extraction / calcul sur les sous-chaînes
  - ...
  - Calcul de l'opération
  - Retour du résultat
- A ce niveau, on devrait avoir un litéral (valeur numérique)
- Retirer les espaces au début et à la fin de la chaîne
- Si la chaîne est vide : renvoyer zero
- Appeler `std::stof` sur la chaîne restante

2 Gestions de fichiers

Comme précisé dans le TP1, la gestion des fichiers se fait sous la forme de Flux d'entrée et de sortie. Ces flux permettent de nombreuses subtilités, mais nous allons surtout nous focaliser sur un usage équivalent au C. Pour accéder aux fonctions liées aux flux d'entrée et de sortie, il faut inclure la bibliothèque `<fstream>`. Les flux d'entrée et de sortie correspondent respectivement aux types `std::ifstream` (pour l'entrée, donc la lecture) et `std::ofstream` (pour la sortie donc l'écriture).

Cela fait notre première différence avec le C, où on utilisait le même type de donnée pour les fichiers en lecture et en écriture (un *descripteur de fichier* ou `FILE *`),

2.1 Écrire un fichier

2.1.1 Syntaxe

Avant d'écrire dans un fichier, il faut l'ouvrir, c'est à dire créer le flux en sortie. Pour cela, la syntaxe de l'initialisation uniforme est très pratique : `std::ofstream fichier {"fichier.txt"}` Notez bien la différence entre l'affectation et l'initialisation : le flux `fichier` ne vaut pas la chaîne de caractère `"fichier.txt"`, mais l'utilise pour son initialisation.

Un programme qui écrit dans un fichier va donc s'écrire comme suit :

```
#include <fstream>

int main()
{
    std::ofstream fich {"fichier_test.txt"};
    fich << "hello";
    return 0;
}
```

Vous remarquerez que des concepts du C ont disparu. En effet, on ne parle plus d'ouvrir et de fermer le fichier. Techniquement ces opérations sont toujours présentes, mais elles sont cachées par la nature même du type `std::ofstream`. En effet l'ouverture du fichier se fait au moment de l'initialisation et la fermeture se fait à la fermeture de l'accolade du groupe, lorsque la variable `fich` est détruite. Dans certains cas particuliers (ou si vous êtes masochiste) il reste possible d'effectuer ces opérations manuellement. Le code ci-dessous vous montre comment :

```cpp
#include <fstream>

int main()
{
   std::ofstream fich;
   fich.open("fichier_test.txt");
   fich << "hello";
   fich.close();
   return 0;
}
```

2.1.2 Fichiers déjà existants

On ne maitrise pas tout dans la gestion de fichiers. Il se pourrait notamment qu'une erreur se produise lors de l'initialisation du flux. Pour repérer ce genre de chose, le flux se comporte comme un bouléen. Il est donc possible de le tester pour voir si l'ouverture s'est bien passée :

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

int main()
{
   std::ofstream fich {"fichier_test.txt"};
   if(fich) {
       fich << "hello";
   } else {
       std::cout << "Oooops ! Erreur de fichier" << std::endl;
   }
   return 0;
}
```

De la même façon, il est possible d'ajouter un paramètre pour indiquer comment le programme doit se comporter si le fichier existe déjà :

- `ios::trunc` : écrase le fichier qui existe déjà
- `ios::ate` : positionne l'écriture en fin de fichier
- `ios::app` : positionne l'écriture en fin de fichier et interdit de revenir en arrière

Ainsi, si vous ouvrez le fichier par `std::ofstream fich {"toto.txt", ios::app}`, il sera impossible de modifier ce qui est déjà écrit dedans. Ces options sont ignorées quand le fichier n'éxiste pas encore.

2.1.3 Se positionner dans le fichier

Comme en C, il est possible de se déplacer dans le fichier pour modifier des données déjà écrites. Pour connaitre la position en cours, on utilise la fonction `.tellp()`. Si on est en fin de fichier, `.tellp()` correspondra donc à la taille de ce dernier. Pour modifier la position, on utilise `.seekp(pos, ref)`. `pos` est un entier (potentiellement négatif), et `ref` définit la référence à utiliser :

- `ios::beg` : Le début du fichier (`pos` est positif ou nul)
- `ios::end` : La fin du fichier (`pos` est négatif ou nul)
- `ios::cur` : La position en cours (`pos` est non nul)

On retrouve alors les codes suivants :

```cpp
   fich.seekp( 0, ios::beg); // back to start of file
   fich.seekp(-1, ios::end); // to last byte of file
   fich.seekp( 1, ios::cur); // one byte forward (if not already at end)
```

2.1.4 Exercice

Écrivez un programme qui crée un fichier contenant les 20 premiers éléments de la suite de Fibonacci. Dans ce fichier, chaque ligne sera constituée du numéro de l'élément suivi de l'élément, avec le caractère `';'` en tant que séparateur. le fichier ressemblera donc à :

```
1;1
2;1
3;2
4;3
5;5
6;8
7;13
...
```

Pour la culture, ce format est généralement mentionné `.csv`, même s'il ne correspond à aucun standard ou norme. Il est très pratique car lisible directement depuis un tableur et facile à gérer *à la main*.

2.2 Lire un fichier

2.2.1 Syntaxe

Sans surprise, pour lire un fichier, on utilisera la syntaxe suivante :

```cpp
#include <fstream>

int main() {
    std::ifstream fich {"data.txt"};
    if(fich) {
        int a;
        std::string s;
        fich >> a;
        fich >> s;
    } else {
        cout << "ERREUR";
    }
}
```

La lecture en utilisant la notation de flux permet d'interpréter les données. C'est à dire que si la variable de destination est un entier, cet entier sera déduit des caractères constituant le fichier. Si cette interprétation échoue, il reste possible d'utiliser `.clear()` et `.ignore()` pour régler le problème.

Il est également possible de lire le flux comme une suite de caractères. La fonction `.get(c)` modifie `c` et lui donne la valeur du caractère suivant. Cette fonction renvoie un bouléen qui permet de savoir si on est arrivé à le fin du fichier. Si c'est le cas, la valeur de `c` n'est pas significative.

La dernière technique est la lecture ligne par ligne grâce à la fonction `std::getline(f, str)` qui place dans `str` une ligne complète du fichier. Cette ligne est affectée sans le saut de ligne final. Comme d'habitude, cette fonction renvoie un bouléen pour identifier les erreurs ou les fins de fichier. `str` doit être de type `std::string` et `f` de type `std::ifstream`.

2.2.2 Se positionner dans le fichier en lecture

Pas de grande surprise, la syntaxe est très proche des flux en écriture, juste avec des noms un peu différents `.tellg()` et `.seekg(pos, ref)`.

2.3 Nom de fichier paramétrable

Dans la grosse majorité des cas, les noms de fichier à lire ou écrire ne sont pas codés en dur dans le programme. Et là, c'est le drame ! En effet, l'initialisation des flux a besoin de connaître le nom de fichier au format *C-string*. À partir du moment ou votre nom de fichier est mémorisé dans une variable de type `std::string`, il faudra donc la convertir avec la fonction `.c_str()`. Bon ça va, rien de catastrophique, mais un réflexe à retenir.

2.4 Exercice

Écrivez un programme qui prend deux noms de fichiers en ligne de commande. Le premier fichier sera lu (nous l'appellerons *source*), le second fichier sera écrit (et donc écrasé s'il existe déjà) nous l'appellerons *destination*. Votre programme doit copier *source* dans *destination* ligne par ligne, en inversant l'ordre. Ainsi, la première ligne de source doit être la dernière de destination et vice versa. Si vous êtes joueur, vous pouvez tester votre programme sur vos propres fichiers source. Sinon, il y a le fichier `jourdain.txt` sur la page de ressources.

Rappel : pour accéder aux arguments de la ligne de commande, il n'y a pas de syntaxe C++ particulière. On utilisera donc la syntaxe du C qui consiste à déclarer `main` par la ligne suivante : `int main (int argc, char *argv[])`. Où `argc` est le nombre d'éléments dans `argv`, et `argv` un tableau de *C-string*.

3 Les espaces de nom

C'est fatigant de taper tous ces `std::` n'est-ce pas ? À quoi cela correspond-il ? Il s'agit de l'espace de noms. Voilà. C'est tout.

3.1 Pour quoi faire ?

Il ne faut pas oublier que l'esprit du C++ est de permettre la migration en douceur d'un projet en C. Et à quelques exceptions près, c'est réussi. Le souci est que de (très) nombreux mot-clés ont été ajoutés dans la syntaxe du C++, et ces mot-clés étaient potentiellement déjà utilisés dans des programmes en C. Il fallait donc trouver une technique qui permet de préciser, en cas de conflit, quelle est la définition à retenir pour le nom. En demandant de préciser `std::` devant chaque mot-clé, il devient clair que c'est la version du standard C++ qui est demandée puisque cette syntaxe est illégale en C. Cette syntaxe permet notamment de créer une fonction ou un type nommé `vector` sans qu'il y ait de problème de conflit avec `std::vector`.

Un autre avantage, c'est que pour les gros projets, il est possible de créer son propre espace de nom. Par exemple, pour les bibliothèques système, graphique ou de calcul scientifique, il est fréquent d'avoir énormément de fonctions à gérer. Vu leur grand nombre, il faut soit des noms particulièrement longs, soit un espace de nom différent. Dans ce dernier cas, il faut simplement que l'étiquette de l'espace de nom soit courte, et différente de celle du voisin :-).

3.2 C'est vraiement indispensable ?

Un programmeur étant, par nature, avare de frappes de touches, il existe une ligne permettant de s'en affranchir : `using namespace std;`. Pour le bloc qui contient cette ligne, le compilateur considérera que tout l'espace de nom `std::` est fusionné avec l'espace de nom courant. Et hop ! tout le code est plus clair. D'autant qu'on peut aussi importer d'autres espaces de noms.

Si cette ligne est présente en dehors d'un bloc de code, (en début de fichier), c'est tout le fichier qui considèrera comme connus les noms de l'espace `std`.

3.3 Pourquoi le garder ?

La technique précédente, bien que très confortable, n'est pas considérée comme une bonne pratique de codage. Dès que le projet devient conséquent (en nombre de lignes, en nombre de participants ...) les espaces de noms rendent beaucoup de services :

- Face à un identifiant inconnu, on sait immédiatement dans quelle doc rechercher. Si c'est `std::`, on cherche dans la doc du C++, s'il n'y a pas d'espace de nom, c'est dans la doc du projet en cours, si c'est autre espace de nom ...
- C++ version 2008 contenait environ 550 mot-clefs dans l'espace de nom standard, pour C++11, c'est 1100. C++14 ? 1220 (petite année). C++23 : 1820. Qu'est-ce qui garantit que les noms de fonction que vous avez choisi aujourd'hui seront encore disponibles l'année prochaine ? Le simple fait que tous ces nouveaux mot-clés sont protégés par un espace de nom différent.

- La remarque précédente s'applique à tout autre sous-projet régulièrement mis à jour. En milieu professionnel, on déteste les surprises, car la loi de Murphy fait qu'elles sont rarement bonnes. Un projet en bonne santé utilise les espaces de nom.

La politique pour cet enseignement est de ne pas favoriser la fusion des espace de nom, mais (pour une fois) de ne pas l'interdire non plus. Faites votre choix, et assumez-le.

4 Tableaux associatifs

Maintenant, on va attaquer des choses un peu plus bizarres. Jusqu'ici, les tableaux que vous avez utilisé en programmation associaient les éléments stockés à un entier. Pour un tableau de taille n, les éléments étaient donc identifiés par une valeur allant de 0 à $n-1$[1]. Et pourquoi d'abord ? Et si on veut faire autrement ? Les tableaux associatifs permettent d'identifier les cases du tableau avec des entiers arbitraires, comme des numéros de téléphone, ou même des types qui ne sont pas des entiers. Juste pour le plaisir.

Les tableaux associatifs peuvent être ordonnés, ou pas. C'est selon les besoins du programmeur. Nous ne passerons du temps que sur les tableaux non ordonnés, car leur implémentation est plus efficace. Dans la suite du sujet, il seront mentionnés sous leur nom de déclaration : `unordered_map`.

4.1 Syntaxe

Un peu de terminologie d'abord : pour chaque élément stocké dans le `unordered_map`, son identifiant est appelé *clé*. La clé peut être un entier, un flottant, ou même une chaîne de caractères (c'est ça qui est beau). L'élément stocké sera simplement nommé *element*.

La déclaration se fait en mentionnant le type de la clé d'abord puis le type de l'élément ensuite. Par exemple, pour un `unordered_map` dont les clés sont des `std::string` et dont les éléments sont des `int` on se retrouvera avec la déclaration suivante : `std::unordered_map<std::string, int>`.

Le code suivant donne un exemple d'usage complet :

```
#include <iostream>
#include <unordered_map>

int main()
{
    // declaration :
    //    keys     : string
    //    elements : float
    std::unordered_map<std::string, float> eval;

    // element insertion
    eval["Bornat"]  =  1.5;
    eval["Kolbl"]   =  3.5;
    eval["Renaud"]  =  2.5;
    eval["Taris"]   =  3.0;
    eval["Valade"]  = 17.25;
    eval["Vincent"] =  2.75;

    // element read
    std::cout << eval["Taris"]  << std::endl;
}
```

Comme tous les conteneurs, il est fourni avec les fonctions `.size()`, `.empty()` et `.at()`. En voici d'autres un peu plus spécifiques :

- `.count(key)` : renvoie 1 si la clé `key` est utilisée ou 0 sinon.
- `.insert({key, elt})` : Insère un nouvel élément dans le tableau. La différence avec l'affectation classique est que l'insertion échoue si la clé est déjà utilisée. Notez bien que `.insert()` ne prend pas deux arguments mais un seul constitué des deux parties.
- `.erase(key)` : Efface l'élément dont la clé est donnée en argument

[1] Qui a parlé de MATLAB ? c'est quoi MATLAB ?

- `.clear()` : Efface tout le tableau

A quoi ça sert ? Essentiellement à ranger des structures en fonction d'une valeur. Imaginez que vous avez une structure qui permet d'identifier des étudiants, des livres ou des véhicules. Sur un système bien ordonné, chaque élément sera associé à un entier qui servira d'identifiant, et qui correspondra certainement à la case du tableau dans lequel sont rangées toutes les informations disponibles. Mais comment faire pour retrouver un élément si vous ne connaissez pas ce numéro ? Soit il faut parcourir tous les éléments, soit il faut créer un système de classification qui soit un peu plus efficace. C'est concrètement ce que fait `unordered_map`, donc pas besoin de le refaire vous-même. Les avantages sont que le travail est déjà fait, il n'y a pas de bugs, la documentation est abondante et les autres contributeurs potentiels sont déjà opérationnels.

4.2 Itération

Il peut arriver qu'il soit nécessaire de faire une itération sur l'ensemble des éléments d'un `unordered_map`. Pour cela est-ce que l'on obtient les clés ou les éléments ? Ben les deux. Pour itérer sur un tel conteneur, il faut utiliser le type `std::pair` qui contient essentiellement deux attributs : `.first` et `.second`. `.first` correspond à la clé, et `.second` à la valeur de l'élément. Voici à quoi cela ressemble dans l'exemple précédent :

```
1    for (std::pair<std::string, float> item : eval) {
2        std::cout << item.first << " : " << item.second  << std::endl;
3    }
```

Petite remarque : quand vous testez ce code, l'ordre d'affichage (donc l'ordre d'itération) n'a rien à voir avec l'ordre dans lequel les éléments sont entrés, ou tout autre ordre de classement. Cela est dû à l'organisation interne de la strucure. Si vous voulez qu'un ordre soit conservé, il faut alors utiliser `map`, mais dont l'usage est plus gourmand en ressources.

A propos du type `std::pair`, il s'agit du cas particulier d'un type utilisé assez courant en objet, le *tuple*. Nous y reviendrons en temps voulu, ou pas, selon l'avancée tout au long des séances.

4.3 Exercice

La page de ressources contient plusieurs textes déjà traités pour éviter les soucis. Notamment, il ne contiennent que des lettres, des chiffres et les caractères suivants : `-,.:;'"!?()#`. Vous choisirez celui de votre choix.

- Écrivez un programme qui donne la distribution des lettres de ce texte. C'est à dire, le nombre de fois ou chaque lettre est utilisée (Nous considèrerons qu'une lettre accentuée est différente de la même lettre non accentuée ou avec un autre accent).
- Écrivez un programme qui donne la distribution de mots de ce texte. C'est à dire, pour chaque mot, le nombre de fois qu'il apparait. Nous considèrerons que les mots sont des suites de caractères séparés par des espaces ou des sauts de ligne, toute forme de ponctuation est un séparateur de mot.

4.4 Question philosophique

Pourquoi est-il dangereux d'utiliser des flottants comme clé pour un tableau associatif ?

5 Conclusion

Avec cette séance, on a commencé à voir que le C++ contenait beaucoup plus de structures de données différentes que le C *tout court*. La bonne nouvelle est que chacune de ces structures permet une gestion très efficace d'un problème en natif. Beaucoup de choses que vous aviez besoin d'écrire en C sont certaintement déjà présente en C++. La mauvaise nouvelle, est que pour être efficace en C++, il faut connaitre ces structures, leurs possibilités et surtout faire les bons choix dès le départ.

Il n'y a donc pas de racourci magique : être efficace en C++ nécessite de la pratique (beaucoup de pratique). comme un peu tout le reste, non ?