Ce chapitre est encore en cours de rédaction, voire à l'état d'ébauche. N'hésitez pas a faire des remarques ou poser des questions sur le forum de Zeste de Savoir ou de OpenClassroom.

Les fichiers

Plusieurs façon de manipuler les fichiers (fonctionnalités système, fonction héritées du C), mais le plus simple est d'utiliser les flux (*stream*). Vous connaissez déjà les flux (*stream*) avec `std::cout` et `std::cin`. Pour rappel, vous pouvez envoyer des données vers un flux en utilisant l'opérateur `«` et lire des données en utilisant l'opérateur `»`.

Les fichiers peuvent donc être manipulés comme des flux, en utilisant `fstream` ("file stream") pour lecture et/ou écriture ou classes plus spécialisées `ofstream` (*output file stream*) et `ifstream` (*input file stream*) respectivement pour l'écriture dans un fichier et la lecture. Ces classes sont disponibles dans le fichier d'en-tête `<fstream>`.

Création et ouverture de fichier

Pour créer une flux sur un fichier, créer simplement une objet de type `stream` en passant en argument le nom du fichier. Par exemple, pour écrire dans un fichier qui s'appelle "test.txt" :

main.cpp

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

int main() {
    std::ofstream file { "test.txt" };
    return 0;
}
```

Si vous créer un fichier en écriture et que celui-ci n'existe pas, alors le

fichier sera créer. Dans Coliru.com, vous pouvez par exemple modifier la ligne de commande utilisée pour compiler et appeler le programme pour ajouter la fonction “ls” pour afficher la liste des fichiers.

```
clang++ -std=c++14 -Wall -Wextra -pedantic main.cpp &&
./a.out && ls
```

Si vous commentez la ligne de code contenant la variable `file` (en ajoutant `/ /` au début de la ligne), la fonction `ls` affiche :

```
a.out
main.cpp
```

Le fichier “a.out” est votre programme, qui est généré lors de la compilation. (Ce nom est celui donné par défaut par le compilateur. Il est bien sûr possible de le changer, mais cela n'est pas important pour le moment, vous verrez plus tard l’utilisation plus avancée des compilateurs).

Le fichier “main.cpp” est le fichier texte contenant votre code C++. (C'est le fichier créé automatiquement par Coliru.com, pour sauvegarder le code C++ que vous tapez dans l'éditeur en ligne).

Avec `std::ofstream`, la fonction `ls` affiche :

```
a.out
main.cpp
test.txt
```

Vous voyez maintenant un troisième fichier qui s'appelle “test.txt”.

En fait, avec une seule ligne de code, le programme fait plusieurs choses. Lors de la création de la variable de type `std::ofstream` :

- création du fichier “test.txt” ;
- ouverture de ce fichier ;
- création d'un flux pour écrire sur ce fichier.

Mais ce n'est pas tout. La variable `file` est une variable locale et suis donc les règles que vous avez vu concernant la portée et la durée de vie

des variables locales : elles existent à partir du moment où elles sont déclarées et sont détruites à la fin du bloc de code dans lequel elles sont déclarées.

Pour la variable `std::ofstream`, cela veut dire qu'elle est détruite à la fin de la fonction `main`.

```
int main() {
    std::ofstream file { "test.txt" };  // création de la variable "file"
    return 0;
}                                       // destruction de la variable "file"
```

En général, la destruction d'une variable (ou plus être plus exacte, “la fin de portée d'une variable”) permet simplement de libérer la mémoire utilisée par cette variable, pour qu'elle puisse être réutilisée pour d'autres variables.

Mais en fait, la destruction d'une variable peut faire beaucoup plus de choses que de simplement libérer la mémoire. Cela peut faire tout ce qui est nécessaire pour libérer correctement les ressources. Par exemple, pour un fichier, cela va permettre :

- d'écrire les données dans le fichier (souvenez-vous que les flux utilisent des tampons mémoires et sont écrit en asynchrone) ;
- d'enregistrer le fichier ;
- et de le fermer.

Resource Acquisition Is Initialization

Cette approche, qui consiste à faire tout ce qui est nécessaire pour qu'une variable soit directement utilisable lors de sa création (acquisition des ressources) et tout ce qui est nécessaire pour libérer correctement les ressources lorsque la variable est détruite est un concept très important en C++. Tellement important qu'il a un nom : “Resource Acquisition Is Initialization” (RAII).

Couplé avec l'utilisation des variables locales, qui permettent de détruite

automatiquement les variables (et donc libérer automatiquement les ressources), et vous avez le moyen le plus simple et le plus sûr pour gérer les ressources en C++.

Ce concept sera détaillé dans la partie sur la programmation orientée objet.

Il est possible d'ouvrir et fermer un fichier à n'importe quel moment, sans devoir créer et détruite une nouvelle variable `fstream` à chaque fois, en utilisant les fonctions `open` (*ouvrir*) et `close` (*fermer*). Par exemple, cela permet d'ouvrir un fichier en lecture, écrire dedans, le fermer ou le ré-ouvrir en écriture, tout cela avec une seule variable.

Dans ce cours, une importance particulière est mise sur la gestion des ressources via l'utilisation des variables locales. L'utilisation des fonctions `open` et `close` n'est pas détaillée. Pour écrire puis lire un même fichier, il est donc recommandé dans ce cours de créer deux variables locales, en utilisant des blocs de code pour gérer les ouvertures et fermetures de fichier.

main.cpp

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

int main() {
    {
        std::ofstream file { "test.txt" };  // ouverture du
fichier en écriture
        file << ...                         // écriture des
données
    }                                       // fermeture du
fichier
    {
        std::ifstream file { "test.txt" };  // ouverture du
fichier en lecture
        file >> ...                         // lecture des
données
    }                                       // fermeture du
fichier
    return 0;
```

```
}
```

Ecriture et lecture de données

Une fois que le fichier est ouvert en écriture avec `std::ofstream`, vous pouvez écrire directement dedans en utilisant l'opérateur `«`.

main.cpp

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

int main() {
    std::ofstream file { "test.txt" };
    file << "Hello word! " << std::endl;
    file << 1234 << std::endl;
    return 0;
}
```

Pour voir le contenu du fichier sur Coliru.com, vous pouvez utiliser la commande “cat nom_fu_fichier” à la place de “ls”.

```
clang++ -std=c++14 -Wall -Wextra -pedantic main.cpp &&
./a.out && cat test.txt
```

affiche :

```
Hello word!
1234
```

Pour rappel, `std::endl` force également l'écriture des données dans le tampon mémoire (*flush*). Pour éviter cela, il est possible d'utiliser le caractère “retour à la ligne” `\n`. Dans ce cours, les performances ne sont pas critiques, l'utilisation de `std::endl` sera privilégiée.

L’écriture sur un flux suit donc la forme générale suivante, pour envoyer une ou plusieurs valeurs.

```
FLUX << DATA;
FLUX << DATA << DATA << DATA << DATA ...
```

Pour lire un flux, vous avez déjà également vu que la forme générale est la suivante :

```
FLUX >> DATA;
FLUX >> DATA >> DATA >> DATA >> DATA ...
```

Une différence importante entre un flux en lecture et un flux en écriture est qu'il est pas possible de lire des données vers une littérale (une littérale est toujours une valeur constante).

```
int x { 123 };
file << x;    // ok
file >> x;    // ok

file << 123;  // ok
file >> 123;  // erreur
```

Pour lire les données depuis un fichier, vous pouvez donc créer un flux de type `std::ifstream` et utiliser l'opérateur `»`.

main.cpp

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

int main() {
    std::ifstream file { "test.txt" };
    std::string s {};
    file >> s;
    return 0;
}
```

Sur Coliru.com, à chaque fois que vous compiler un programme, l'espace de travail est nettoyé. Cela signifie que si vous écrivez un premier code utilisant `std::ofstream` pour créer un fichier, que vous l'exécutez, puis que vous créer un nouveau code utilisant `std::ifstream` pour lire le fichier créé, ce dernier ne sera plus disponible.
Pour tester l'utilisation des fichiers sur Coliru.com, il est donc préférable

d'écrire et lire un fichier dans un même programme. Dans la suite de ce cours, les codes d'exemples seront généralement écrit pour être exécuté sur Coliru.com, ce qui signifie que les codes contiendront une première série de ligne pour créer un fichier, puis les lignes de code pour lire ce fichier.

main.cpp

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

int main() {
    {   // creation du fichier
        std::ofstream out { "test.txt" };
        out << 123 << std::endl;
    }

    std::ifstream in { "test.txt" };
    int i {};
    in >> i;
    std::cout << i << std::endl;

    return 0;
}
```

affiche :

```
123
```

Ecrire plusieurs valeurs, espaces et retours a la ligne

Comme pour les autres flux en lecture, il est possible de lire plusieurs données dans un flux `ifstream` (Heureusement ! L'intérêt des fichiers seraient limité si il n'était possible d'écrire qu'une seule valeur par fichier).

Avec des nombres entiers

Si vous ajoutez plusieurs valeurs dans le code précédent, vous écrirez probablement le code suivant :

main.cpp

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

int main() {
    {   // creation du fichier
        std::ofstream out { "test.txt" };
        out << 123 << 456 << 789 << std::endl;
    }

    std::ifstream in { "test.txt" };
    int i {};
    in >> i;
    std::cout << i << std::endl;
    in >> i;
    std::cout << i << std::endl;
    in >> i;
    std::cout << i << std::endl;

    return 0;
}
```

Malheureusement, ce code n'affiche pas trois valeurs, comme vous pouvez vous y attendre, mais une seule valeur.

```
123456789
```

Si vous utiliser la commande `cat` pour voir le contenu du fichier, vous voyez qu'il contient également qu'un seule valeur. Et bien sur, lors de la lecture d'un tel fichier, il n'est pas possible de savoir que vous avez écrit trois valeurs de trois chiffres, ou neuf valeurs de un chiffre, ou encore une valeur de neuf chiffres.

Modifiez le code pour ajouter un retour a la ligne ou un espace entre deux valeurs.

main.cpp

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

int main() {
    {   // creation du fichier
        std::ofstream out { "test.txt" };
        out << 123 << ' ' << 456 << std::endl;
        out << 789 << std::endl;
    }

    std::ifstream in { "test.txt" };
    int i {};
    in >> i;
    std::cout << i << std::endl;
    in >> i;
    std::cout << i << std::endl;
    in >> i;
    std::cout << i << std::endl;

    return 0;
}
```

Maintenant, la commande `cat` indique que le fichier contient le texte suivant (ce qui correspond a ce que vous pouviez vous attendre) :

```
123 456
789
```

Le programme affiche :

```
123
456
789
```

Donc, dans ce code, la lecture du fichier est correcte et chaque valeur est lue individuellement. Les espaces et les retours a la ligne sont reconnus comme des séparateurs de valeurs et `ifstream` lit correctement le fichier.

Avec des nombres réels

Les flux permettent également d'utiliser des nombres réels et reconnaissent l'écriture scientifique.

main.cpp

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

int main() {
    {   // creation du fichier
        std::ofstream out { "test.txt" };
        out << 123.4546 << std::endl;
        out << 123.456e78 << std::endl;
    }

    std::ifstream in { "test.txt" };
    double x {};
    in >> x;
    std::cout << x << std::endl;
    in >> x;
    std::cout << x << std::endl;

    return 0;
}
```

affiche :

```
123.455
1.23456e+80
```

Vous voyez ici que la seconde valeur n'utilise pas le même format d'affichage que celui utilisée dans le code (et dans le fichier). La raison est que les valeurs n'ont pas de format d'affichage a proprement parlé. Ce sont les flux qui affichent les valeurs réelles en utilisant un format (que vous pouvez modifier, pour rappel, voir Les nombres à virgule flottante).

Donc la littérale "123.456e78" dans le code est convertie en une valeur qui ne possède pas de format lorsque le code est compilé. Puis l'écriture

de cette valeur dans un fichier utilise le format par défaut de `std::ofstream`. Cette valeur formatée dans le fichier est ensuite lue par `std::ifstream` (qui reconnait ce format) et est enregistrée dans la variable `x` sans son format. Et pour terminer, le flux `std::cout` affiche la valeur en utilisant son propre format par défaut.

L'alternance d'étapes qui ne conservent pas le format et de flux qui possèdent leur propre format par défaut explique que la valeur dans le code n'a pas le même format que la valeur affichée au final. Dans la majorité des cas, ce comportement par défaut convient et facilite les lectures et écritures de données.

En cas de probleme de lecture, n'hésitez pas a vérifier les formats utilisés en lecture et écriture.

Il faut egalement faire attention a ne pas melanger les nombres entiers et reels. Si vous essayez de lire un fichier contenant une valeur reelle pour l'enregistrer dans une variable entiere, le comportement sera indetermine.

main.cpp

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

int main() {
    {   // creation du fichier
        std::ofstream out { "test.txt" };
        out << 123.456e78 << std::endl;
    }

    std::ifstream in { "test.txt" };
    int i {};
    in >> i;
    std::cout << i << std::endl;

    return 0;
}
```

affiche :

```
1
```

L'inverse ne pose pas de probleme, un nombre entier est une forme particuliere de nombre reel (pour lequel la partie decimale est nulle).

Avec des chaines de caractères

probleme avec les espaces. Utilisation de getline

Options d'ouverture

Le premier paramètre est le nom du fichier. Attention, le chemin par défaut est celui de l'application, pas celui du fichier .cpp (s'ils ne sont pas au même endroit). Possibilité de mettre chemin relatif ou absolu :

```
std::ofstream file("out.txt");
std::ofstream file("../../out.txt");
std::ofstream file("C:/myapp/out.txt");
```

Remarque : normalement, sous Windows, le séparateur de chemin est la barre oblique inversée `\`. Vous pouvez l'utiliser, mais n'oublier pas que ce caractère correspond au caractère d'échappement en C++, il faut dont écrire `\\` ou utiliser une chaîne brute :

```
std::ofstream file("C:\\myapp\\out.txt");  // double barre
oblique inversée
std::ofstream file(R"(C:\myapp\out.txt)"); // chaîne brute
```

Le second paramètre permet de spécifier les options d'ouverture :

- `std::ios::binary` : ouverture en mode binaire ;
- `std::ios::in` : en lecture (pour fstream) ;
- `std::ios::out` : en écriture (pour fstream) ;
- `std::ios::app` (*append* = “ajouter”) : ajoute à la fin du fichier ;

- `std::ios::trunc` (*truncate* = “tronquer”) : supprimer le contenu ;
- `std::ios::ate` (*at the end* = “à la fin”) : ajoute à la fin lors de l'ouverture.

Validation a la compilation et typage fort
Supposons que vous recevez un `std::fstream` quelconque, sans rien savoir sur ce flux. Pour pouvoir travailler dessus, vous devrez tester lors de l'execution si le fichier est ouvert en ecriture et/ou lecture pour travailler dessus.

```
std::fstream f { ... };
assert(f.is_open());
assert(f.is_readable());
f << "hello world";
```

En toute rigueur, vous devrez faire cela dans tout les codes qui utilise `std::fstream`, mais c'est lourd.

Possible d'utiliser le typage fort du C++ pour simplifier le code. Au lieu d'utiliser un type generique comme `std::fstream`, utiliser un type plus specifique, qui apporte des garanties sur ce que l'on peut faire ou non avec le flux. Par exemple `std::ofstream` garantie que l'on peut ecrire dessus (s'il est valide). Pas besoin de tester.

```
std::ofstream f { ... };
assert(f.is_open());
f << "hello world";
```

La validation est faite a la compilation = plus performant, plus sur et code plus simple.

Et cette approche est interessante en C++, a faire des que possible. Par exemple, meme avec `ofstream`, vous devez tester si le flux est valide (ouvert). Au lieu d'ajouter un test systematique, creer une classe `valide_ofstream` :

```
std::fstream f { ... };
assert(f.is_open());
```

```
assert(f.is_readable());
f << "hello world";

// par

valide_ofstream f { ... };
f << "hello world";
```

Ecriture et lecture

Problème de séparation des caractères. De lecture d'une ligne

lecture complet dans un string ?

Caractère spéciaux : tabulation \t retour à la ligne \n

Problème lecture windows/linux : \n et \r\n

ouverture et fermeture manuelle ? (open, is_open, close)

Se positionner dans un fichier ? (eof, fpos, etc.)

Note : n'oubliez pas que les flux utilisent en général un tampon mémoire (*buffer*)

Gestion des erreurs, exceptions

Format texte et binaire

Permettre la lecture et l'écriture de fichiers. 2 types de fichiers : binaire et texte. Déjà vu les différentes représentation d'un nombre, en binaire :

main.cpp

```
#include <iostream>
#include <bitset>

int main() {
    int const i { 123456 };
```

```
    std::cout << "représentation numérique: " << i << std::
endl;
    std::cout << "représentation binaire: " << std::bitset<
sizeof(i)*8>(i) << std::endl;
    std::cout << sizeof(123) << std::endl;
    std::cout << sizeof("123") << std::endl;
    std::cout << sizeof(123456) << std::endl;
    std::cout << sizeof("123456") << std::endl;
    std::cout << sizeof(123456789) << std::endl;
    std::cout << sizeof("123456789") << std::endl;
}
```

affiche :

```
représentation numérique: 123456
représentation binaire: 00000000000000011110001001000000
4
4
4
7
4
10
```

En mode texte, chaque chiffre prend 1 octet (+ un octet de fin de chaîne), en binaire, toujours la même taille pour un int. Donc fichier texte permet d'être lu avec un simple éditeur, mais plus gros en taille.

Pratiquer

- fichier csv (comma separated values) : tableau avec séparation par `,`
- fichier texte excel : tabulation + retour à la ligne
- génération xml
- génération json
- lecture et écriture image ?

Chapitre précédent	Sommaire principal	Chapitre suivant