Université de Nice-Sophia Antipolis
Polytech ELEC4

Mardi 1er octobre 2019
Durée: 2h

C++ – Fichiers
Travaux Dirigés – Séance n. 4

---

1 Objectif

Dans ce TD, nous verrons comment manipuler les flux (fichiers standard ou non) en lecture ou en écriture.

2 E/S standard et Fichiers

On a déjà vu que l'espace des noms standard (`std`) fournissait deux objets nommés `cin` et `cout` qui représentent, respectivement, l'entrée standard et la sortie standard. Dans cet espace, il existe aussi `cerr` qui désigne la sortie d'erreur standard. Ces objets sont de type `istream` (pour `cin`) et `ostream` (pour `cout` et `cerr`). Ils définissent des fichiers de caractères qui offrent des conversions implicites de type. Rappel : ils sont accessibles après inclusion de `iostream`.

Les fichiers d'octets sont représentés par des objets de type `ifstream` (en lecture) ou `ofstream` (en écriture).

Ainsi, les deux déclarations ci-dessous construisent deux objets, accessibles par `ios` et `oos` pour manipuler, respectivement, les fichiers *entrée* en lecture et *sortie* en écriture. Ces objets sont accessibles après inclusion de `fstream`.

```
ifstream ios("entrée");
ofstream oos("sortie");
```

Notez que deux déclarations ne font pas de vérification sur la validité de l'accès aux fichiers dans le système d'exploitation. La méthode `is_open` permet de faire cette vérification. Par exemple, on pourra écrire le fragment de code suivant pour vérifier la bonne ouverture du fichier *entrée* :

```
#include <fstream>
...
ifstream ios("entrée");
if (!ios.is_open()) {
  perror("entrée");
  exit(EXIT_FAILURE);
}
```

Enfin, la méthode `close` permet la fermeture d'un fichier ouvert en lecture ou en écriture.

3 Manipulation caractère à caractère

La méthode `get` permet de lire le prochain caractère disponible dans un fichier. La méthode arrête la lecture lorsque la fin de fichier est atteinte. Réciproquement, la méthode `put` écrit un caractère à la fin du fichier. Par exemple, le programme suivant recopie l'entrée standard sur la sortie standard.

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main(void) {
  char c;
  while (cin.get(c))
    cout.put(c);
  return EXIT_SUCCESS;
}
```

Notez qu'il existe une autre version de `get` sans paramètre, similaire à celle de C, qui renvoie le caractère lu. On pourra tester la valeur lue avec `EOF` pour déterminer la fin de fichier. Notez aussi que lorsque la fin de fichier est atteinte, la fonction `eof()` appliquée sur le fichier d'entrée renvoie la valeur *vrai*.

exercice 1) Écrivez la procédure `copier` qui prend deux chaînes de caractères (`string`) en paramètre qui représentent les noms des fichiers d'entrée et de sortie, et qui procède à la copie du contenu du fichier d'entrée sur le fichier de sortie.

exercice 2) Testez votre fonction `copier` dans une fonction `main`. Vous lirez les noms des 2 fichiers sur l'entrée standard (`cin`).

4 Manipulation par bloc de caractères

Les méthodes `read` et `write` permettent de lire (respectivement écrire) n caractères placés (respectivement à placer) dans un tableau de caractères. Elles ont deux paramètres. Le premier est le tableau et le second le nombre de caractères à lire ou à écrire. Si le nombre de caractères disponibles est inférieur à n, la fin de fichier est atteinte et `eof()` est vrai, et la fonction `gcount()` renvoie le nombre de caractères *effectivement* lus.

exercice 3) Écrivez la procédure `copierBuf` qui, comme précédemment, copie un fichier d'entrée sur un fichier de sortie, mais cette fois-ci par blocs de `BUFSIZ` caractères.

exercice 4) Testez votre procédure `copierBuf` dans une fonction `main`. Vous lirez les noms des deux fichiers sur l'entrée standard.

5 Écriture et lecture d'objets

Les méthodes `read` et `write` permettront également de lire et écrire des objets (définis par des classes). Nous souhaitons, par exemple, écrire dans un fichier de nom *"fc"* des nombres complexes décrits par la classe `complexe` du précédent TD. Notez que ces complexes seront écrits dans un format binaire.

Le premier paramètre de la méthode `write` de `ofstream` est un pointeur sur caractère. Si nous souhaitons écrire un objet de type complexe, il faudra donc d'abord le convertir. Le second paramètre est le nombre de caractères (octets) à écrire, c'est-à-dire la taille d'un complexe. Le fragment de code suivant ouvre le fichier *"fc"* et y écrit le complexe *(3.1, -4.5)*.

```
ofstream oos("fc");
complexe c(3.1, -4.5);
oos.write((char *) &c, sizeof(complexe));
oos.close();
```

De façon similaire, la fonction `read` de `ifstream` permettra de lire un objet, *e.g.* un complexe. Le complexe contenu dans le fichier *"fc"* pourra être relu de la façon suivante :

```
ifstream ios("fc");
```

```
complexe c;
ios.read((char *) &c, sizeof(complexe));
ios.close();
```

exercice 5) Écrivez un programme qui écrit n complexes dans un fichier *"fc"*. Le nombre n de complexes à écrire et les parties réelles et imaginaires de chacun des complexes sont lues sur l'entrée standard.

exercice 6) Écrivez un second programme qui relit le fichier *"fc"* et affiche les nombres complexes au format *(r,i)* sur la sortie standard. Dans ce second programme, on considère qu'on ne connaît pas le nombre de complexes enregistrés dans le fichier.

Peut-on utiliser les opérateurs `<<` et `>>` pour faire des écritures ou des lectures d'objets ? La réponse est *oui* à condition de *surcharger* ces deux opérateurs. Nous avons vu dans le TD précédent comment surcharger les opérateurs `+` ou `-` pour additionner ou soustraire deux complexes.

Pour les opérateurs `<<` et `>>`, nous devrons écrire dans `complexe.hpp` les prototypes suivants :

```
friend std::ofstream& operator<< (std::ofstream& f, const complexe& c);
friend std::ifstream& operator>> (std::ifstream& f, complexe& c);
```

Nous verrons ultérieurement à quoi correspond le mot-clé **friend**.

L'écriture de la méthode **operator<<** dans le fichier `complexe.ccp` pourra s'écrire :

```
std::ofstream& operator<< (std::ofstream& f, const complexe& c)  {
  f.write((char *) &c, sizeof(complexe));
  return f;
}
```

Notez bien l'absence de `complexe::` devant **operator<<**. L'opérareur `<<` *n'est pas* une fonction membre de la classe `complexe`.

Ainsi, on pourra récrire le code précédent qui écrit le complexe *(3.1, -4.5)* dans le fichier `"fc"` beaucoup plus simplement :

```
ofstream oos("fc");
oss << complexe(3.1, -4.5);
oos.close();
```

exercice 7) Inspirez-vous de la surcharge de l'opérateur `<<` pour écrire celle de l'opérateur de lecture `>>`.

exercice 8) Récrivez les deux programmes pour écrire et lire les n complexes en utilisant les deux opérateurs surchargés.

exercice 9) À l'aide de la commande unix `od --format=fD`, visualisez le contenu de fichier `"fc"` que vous avez crée.

Les complexes que nous avons écrits et lus dans *"fc"* sont dans un format binaire. Si nous voulons les écrire dans un format textuel (suite de caractères), il sera également possible de surcharger les opérateurs `<<` et `>>` avec les prototypes suivants :

```
friend std::ostream& operator<<(std::ostream& f, const complexe& c);
friend std::istream& operator>>(std::istream& f, complexe& c);
```

exercice 10) Placez ces deux prototypes dans `complexe.hpp` et écrivez ces deux méthodes dans `complexe.cpp`. Vous pourrez réutiliser la méthode `ecrireComplexe` du td précédent et définir une nouvelle méthode `lireComplexe` qui lit la partie réelle et imaginaire sur l'entrée standard.

exercice 11) Récrivez les deux programmes qui écrivent et qui lisent les n complexes en utilisant la surcharge précédente.

Vos fonctions `ecrireComplexe` et `lireComplexe` écrivent et lisent, respectivement, sur `cout` et `cin`. Il serait plus habile de les paramétrer de telle façon qu'elle puissent écrire sur un `std::ostream` et sur un `std::istream`.

exercice 12) Modifiez ces deux fonctions pour faire ce paramétrage et modifiez la surcharge des opérateurs qui les utilisent et testez vos programmes.

6 Manipulateur de flux

Un manipulateur de flux permet de modifier le comportement du flux. Au TD1, nous avons vu un premier manipulateur, `boolalpha` et `noboolalpha`, qui permet de modifier un flux de sortie pour afficher de façon alphabétique ou non un booléen. Il en existe d'autres. Certains sont accessibles après inclusion du fichier `iomanip`.

Les manipulateurs `hex`, `oct` et `dec` permettent d'écrire des entiers au format hexadécimal, octal et décimal.

exercice 13) Écrivez un programme qui lit un entier et l'écrit sur la sortie standard selon les 3 formats précédents.

exercice 14) Testez si ces manipulateurs agissent également sur le flux d'entrée standard.

Les nombres réels peuvent être traités soit de façon standard (par défaut), soit en notation scientifique, manipulateur `scientific`, soit en notation fixe, manipulateur `fixed`. En lecture, les 3 formats sont admis sans avoir à spécifier un manipulateur. Pour annuler, `scientific` ou `fixed` et revenir au comportement par défaut, il faut annuler le manipulateur en appliquant la méthode `unsetf(ios_base::scientific)` ou `unsetf(ios_base::fixed)`.

exercice 15) Écrivez un programme qui lit un nombre réel et l'écrit sur la sortie standard selon les 3 formats précédents.

Les manipulateurs `showpoint` et `noshowpoint` permettent d'imposer ou pas le point dans l'écriture du réel quand la partie décimale est nulle.

`showpos` et `noshowpos` permettent d'imposer ou non le `+` devant les nombres (entiers ou réels) positifs. Le manipulateur `setprecision(`n`)` spécifie le nombre n de chiffres à écrire.

exercice 16) Déclarez la constante `pi=3.141592653` et écrivez sa valeur sur la sortie standard sur 5 chiffres.

Voici d'autres manipulateurs pour faire des alignements. `setw(`n`)` permet de réaliser un cadrage sur n caractères, `setfill(`c`)` remplit l'espace avec le caractère c, `left` et `right` permettent, respectivement un cadrage à gauche et à droite.

exercice 17) Complétez le programme suivant :

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
using namespace std;
int main(void) {
  const double pi=3.141592653;
  const int n=30;
  cout << ..............
  .....
  return EXIT_SUCCESS;
}
```

afin de produire l'affichage suivant sur la sortie standard (notez que le mot *pi* est cadré à droite, et sa valeur est sur 4 chiffres, cadrée à gauche) :

```
+--------------+--------------+
|            pi|3.142         |
+--------------+--------------+
```