

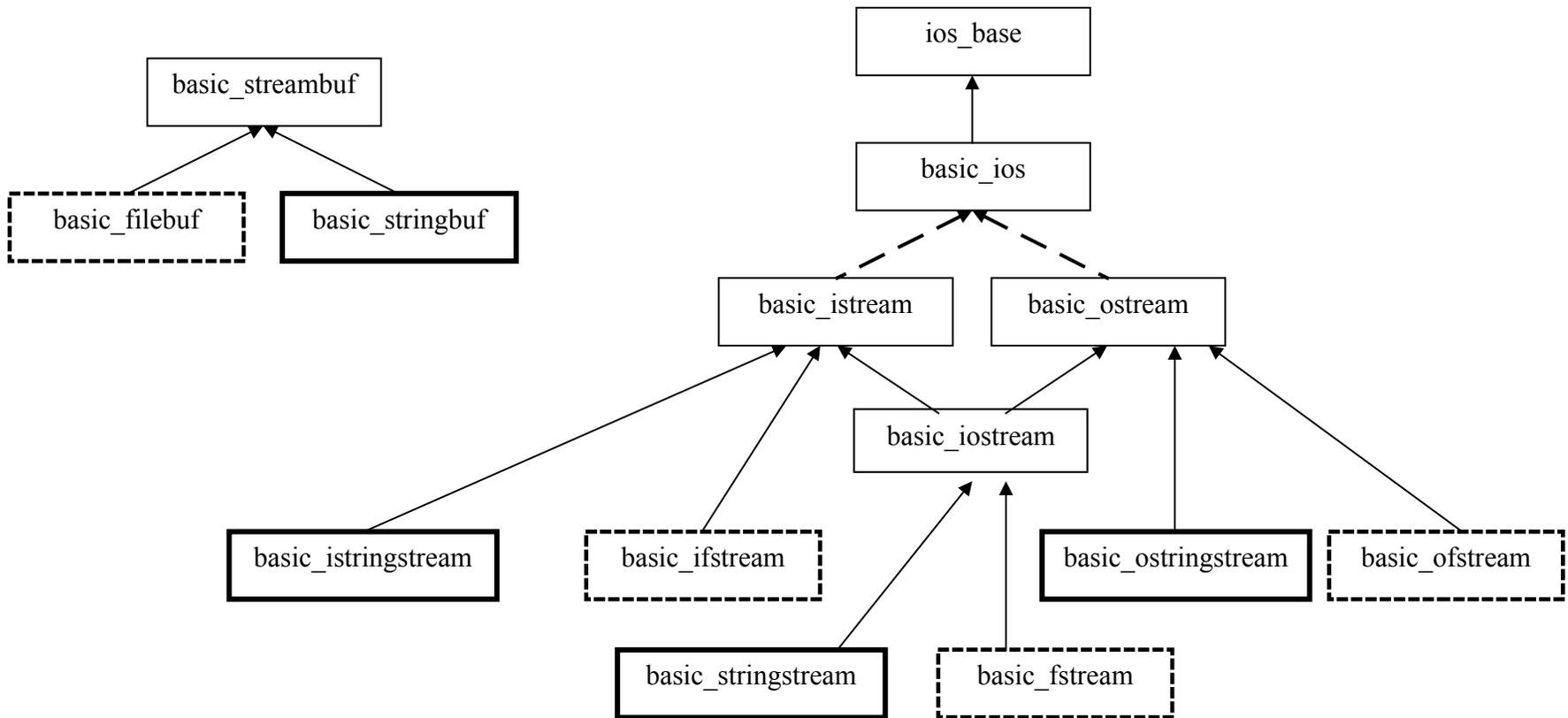
Les flux

1. Généralités

- Un flux (ou `stream`) est une abstraction logicielle représentant un flot de données entre une source produisant de l'information et une cible consommant cette information.
- Les instructions d'entrées/sorties ne font pas partie de la spécification de base du langage C++, leur implémentation peut varier d'un compilateur à un autre.
- Nous avons déjà étudié 3 types de flots: « `cin` », « `cout` », « `cerr` ». Dans ce chapitre, nous allons introduire des méthodes pouvant manipuler ces flots ainsi que leurs associations à des fichiers donnés.

2. Hiérarchie de « `iostream` »

- Le flux d'entrée est représenté par le mot clé « `iostream` » qui est un diminutif de « `InputOutputSTREAM` » c.-à-d. flux (« `stream` ») d'entrées (« `Input` ») et sorties (« `Output` »).
- Une représentation simplifiée de la hiérarchie « `iostream` » est comme suit:



Qui a un rapport avec la classe (i,o)fstream

Qui a un rapport avec la classe (i,o)stringstream

Qui a un rapport avec les classes stream

--- virtual --->
 Non virtual -->

- « **basic_fstream** » : Pour l'accès aux fichiers définis en mode lecture et écriture afin de permettre des opérations d'entrées et de sorties de et vers un fichier.
- « **basic_ifstream** » : Pour les fichiers définis uniquement en lecture afin de permettre des opérations d'entrées du fichier.
- « **basic_ofstream** » : Pour les fichiers définis uniquement en écriture afin de permettre des opérations de sortie vers le fichier.
- « **basic_stringstream** » : Il permet de simuler des opérations d'entrées/sorties avec des tampons (buffers) en mémoire centrale. Même principe que les fonctions du langage C, « `sprintf()` » et « `scanf()` ».
- « **basic_istringstream** » : La lecture dans un tampon mémoire (équivalent à un « `scanf` »).
- « **basic_ostringstream** » : L'écriture dans un tampon mémoire (équivalent à un « `sprintf` »).
- « **basic_streambuf** » : La classe à part sert à la lecture et écriture dans le tampon.
- « **basic_stringbuf** » : La manipulation de caractères.
- « **basic_filebuf** » : L'association de l'espace tampon avec le fichier.

3. Flot en entrée « istream »

- Mise en forme ou non de l'entrée dans un « streambuf »
 - Surdéfinir l'opérateur « >> » pour qu'il gère la lecture.
 - Par défaut l'opérateur « >> » ignore tous les espaces blancs. Le flag est « ios::skipws » : skip white space.
 - Gère les types prédéfinis du langage « C++ ».
- Au lieu d'utiliser l'opérateur « >> », on peut se servir d'autres fonctions pour la saisie des données.

3.1. La méthode « int get() »

- Retourne la valeur du caractère lu, sinon EOF si la fin du fichier est atteinte.

```
int main(){
    char c;
    while ((c=cin.get()) != EOF)
        cout << c;
    cout << "on sort de la boucle!" << endl;
    return 0;
}
```

- Le programme lit indéfiniment les caractères à partir du clavier.
- On arrête la lecture si les touches « CTRL » et « D » sont appuyées en même temps, sinon lorsqu'on appuie sur la touche retour chariot (`return`) du clavier.
- Après l'affichage, le programme se remet en état d'attente de lecture.
- Pour arrêter complètement le processus, et passer à la suite du programme (ici l'affichage de l'expression: `on sort de la boucle!`), il faudra appuyer sur « CTRL-D » une seconde fois.
- Un appel du style « CTRL-C » arrêtera complètement le programme sans exécuter la suite des instructions qu'il pourra contenir.

3.2. La méthode « `istream& get (char& c)` »

- Elle extrait le premier caractère du flux, même si c'est un espace et elle le place dans le caractère `c`.

```
#include <iostream>
using namespace std;
int main(){
    char c;
    while (cin.get(c))
        cout << c;
    cout << "on sort de la boucle!" << endl;
    return 0;
}
```

- Pour les touches « CTRL-D », voir le paragraphe précédent.

3.3. Les méthodes « `istream& get (char* str, streamsize count, char delim)` » ; « `istream& get (char* str, streamsize count)` »

- Extrait un nombre « `count-1` » caractères du flux et place le résultat à l'adresse pointée par la variable « `ch` ».
- La lecture s'arrête au délimiteur « `delim` » (la première version de la fonction) qui est par défaut le caractère « `'\n'` » (la seconde version de la commande) ou la fin du fichier.
- Le délimiteur n'est pas extrait du flux.

```
#include <iostream>
using namespace std;
int main(){

    char c[80];
    cin.get(c,79);
    cout << c << endl;
    return 0;
}
```

- Le caractère « `'\0'` » est inséré à la fin de la chaîne.
- Il faudra s'assurer que le « `buffer` » peut contenir « `count-1` » caractères.

3.4. Les méthodes « `istream& getline (char* str, streamsize count, char delim)` » ; « `istream& getline (char* str, streamsize count)` »

- La même chose qu'en 3.3, sauf qu'ici le délimiteur est extrait du flux, mais il n'est pas recopié dans le tampon.
- Le délimiteur est extrait du flux (lu) à condition de se trouver dans les « `count-1` » caractères à lire.

3.5. La méthode « `istream& read (char* str, streamsize count)` »

- Cette fonction extrait un bloc de taille « `count` » et elle le place dans la chaîne « `str` ».
- Il faut s'assurer que la chaîne « `str` » est d'une taille suffisante pour contenir ce bloc.
- Cette fonction est utilisée généralement pour la lecture binaire.

```
char c[] = "?????????";  
// on ne va lire que 4 valeurs.  
cin.read(c,4);  
cout << endl << c << endl;
```

- **En entrée:** 1234
- **En sortie:** 1234??????

3.6. La méthode « `streamsize gcount() const` »

- Cette fonction retourne le nombre de caractères non mis en forme, extraits lors de la dernière lecture.

```
char c[] = "?????????";  
cin.read(c,4);  
cout << endl << "buffer: " << c << " : \"  
      << cin.gcount() << endl;
```

- **En entrée:** 1234
- **En sortie:** buffer: 1234?????? : 4

3.7. La méthode « `int peek()` »

- Cette fonction met le prochain caractère dans le flux sans l'extraire de ce dernier.
- Elle retourne « EOF » (End Of File) s'il n'y a plus de caractères à lire du flux d'entrée.

```
cin.peek();  
// affiche le code ascii de la chaîne  
cout << cin.peek() << endl;  
cout << (char) cin.peek() << endl; // affiche sa valeur
```

- **En entrée:** 1234
- **En sortie:** 49
1

3.8. La méthode « `istream& putback(char c)` »

- Cette fonction retourne le caractère lu dans le « stream », pour être relu lors de la prochaine lecture (voir aussi la fonction un « `get` »).
- Pour qu'elle fonctionne correctement, le caractère réinséré doit être le dernier caractère lu.

```
char c = cin.get();  
cin.putback(c); // remet le caractère c dans le flux.  
cout << cin.peek() << endl; // il sera lu ici,  
cout << (char) cin.peek() << endl; // et aussi ici.
```

- **En entrée:** 1
- **En sortie:** 49
1

3.9. La méthode « `istream& ignore()` » ; « `istream& ignore(streamsize count)` » ; « `istream& ignore(streamsize count, int delim)` »

- Toutes ces fonctions extraient un ou plusieurs caractères et les ignorent par la suite.
- La première forme extrait un seul caractère, la seconde ignore jusqu'à « `count` » caractères, la troisième ignore « `count` » caractères, jusqu'à que le délimiteur soit extrait et ignoré.

- En plus de ces fonctions, il y a les fonctions « `tellg` » et « `seekg` ».
- Elles seront traitées plus en détail dans le paragraphe « Association d'un flux à un fichier ».

4. Flot en sortie « `ostream` »

- Mise en forme ou non de la sortie dans un « `streambuf` ».
- Surdéfinir l'opérateur « `<<` » pour qu'il gère l'écriture.
- Il gère les types prédéfinis du langage C++.
- Au lieu d'utiliser l'opérateur « `<<` », on peut se servir d'autres fonctions pour l'écriture des données.

4.1. La méthode « `ostream& put(char c)` »

- Cette fonction écrit l'argument `c` dans le flux sélectionné en sortie.

```
char c = 'a';  
// On pouvait écrire directement: cout.put('a');  
cout.put(c) << endl;
```

- **En sortie:** a

4.2. La méthode « ostream& write(const char* s, streamsize count) »

- Cette fonction écrit « count » caractères dans le flux de sortie.
- Il faudra s'assurer que la chaîne « s » contient effectivement « count » caractères sinon le comportement de la fonction est hasardeux!

```
const char *c = "comportement hasardeux!";  
cout.write(c,12) << endl;
```

- **En sortie** : comportement

4.3. La méthode « ostream& flush() »

- Cette fonction force l'affichage sur le flux de sortie du contenu de tous les tampons de données associés aux flux de sortie.
- Assez souvent, cette fonction est réalisée implicitement par l'emploi du délimiteur « '\n' » ou « endl ».
- On peut utiliser aussi la fonction « flush » pour le faire sauf que cette fonction ne permet pas un retour à la ligne comme dans le cas d'un « endl » par exemple.

```
const char *c = "comportement hasardeux!";  
  
cout << "ne marche pas bien ici: " << cout.write(c,12) << endl;  
  
cout << "par contre ici, marche très bien: " << flush;  
  
cout.write(c,12) << endl;
```

- **Sortie :** comportementne marche pas bien ici: 0xffffffff
par contre ici, marche très bien: comportement
- En plus de ces fonctions, il y a les fonctions « tellp » et « seekp ».
- Elles seront traitées plus en détail dans le paragraphe 6, « Association d'un flux E/S à un fichier ».

5. États d'erreurs de flux

- Diverses constantes du type « `iostate` » ont été définies dans la classe « `ios_base` ».
- Ces fonctions permettent de définir l'état d'un flux. Ces constantes sont comme suit:

Constante	Signification	Valeur
<code>badbit</code>	erreur fatale, état indéfini	1 (sinon 0)
<code>failbit</code>	erreur ; échec d'une opération E/S	1 (sinon 0)
<code>eofbit</code>	fin de fichier a été rencontrée	1 (sinon 0)
<code>goodbit</code>	tout est normal, aucun des bits précédents n'a été activé	0 (sinon 1)

- « `badbit` » : ce bit est mis à 1 si le flux est corrompu ou il y a une perte de données. Par exemple, on tente de lire dans un fichier avant le début du fichier.
- « `failbit` » : une opération n'a pas été correctement traitée.
Ce bit est mis à 1 par exemple lorsqu'on s'attend à lire un entier, mais l'utilisateur entre un caractère.
- Le bit « `failbit` » est parfois lié au bit « `eofbit` ». Par exemple, si la fin de fichier est rencontrée. En effet, lors de la prochaine tentative de lecture, « `eofbit` » est mis à 1 de même que « `failbit` ». En effet la lecture était un échec.
- Des méthodes ont été associées aux bits définis précédemment afin d'obtenir ou de modifier leur état.

- Ces méthodes sont :

- **bad()** : elle retourne vrai (`true`) si le bit « `badbit` » est levé, sinon elle retourne faux (`false`).
- **fail()** : elle retourne vrai (`true`) si le bit « `failbit` » ou le bit « `badbit` » est levé, sinon elle retourne faux (`false`).
- **eof()** : elle retourne vrai (`true`) si le bit « `eofbit` » est levé, sinon elle retourne faux (`false`).
- **good()** : elle retourne vrai (`true`) si l'état du flux (le bit « `goodbit` ») est à zéro, sinon elle retourne faux (`false`).
- **setstate(iostate valeur)** : elle lève le bit spécifié en argument ; par exemple l'appel suivant: `setstate(ios::failbit)`, il faut que le bit « `failbit` » soit levé.
- **clear(val)** : elle permet de rétablir le bon état d'un flux afin de permettre l'exécution des entrées/sorties sur ce flux. Par exemple cet appel « `clear(ios::failbit)` » met à l'état initial le bit « `failbit` ». Si aucun argument n'est passé à la fonction « `clear()` », l'état du flux est mis automatiquement à 0 (c.-à-d. « `goodbit` »).
- **rdstate()** : elle retourne l'état d'erreur du flux. Par exemple l'appel « `cin.rdstate()` » retourne l'état d'erreur du flux « `cin` ».

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

using namespace std;

// Cette fonction prend deux arguments:
// le flux en entrée « is », et une variable « couleur »
// qui a pour rôle de mémoriser la valeur entrée.

void GetCouleur (istream& is, string couleur) {

    string entree;
    is >> entree;

    // On teste si la couleur entrée du clavier est du
    // « rouge » ou du « vert »?

    if (entree == "rouge" || entree == "vert") couleur = entree;

    else {

        // Si la couleur n'est pas la bonne ...
        // La taille d'un string est donnée par le champ size().
        // L'instruction suivante positionne à entree.size()
        // en avant dans le flux d'entrée « is ». Il y aura plus de
        // détails sur seekg plus loin dans ce chapitre.

        is.seekg(-entree.size(), ios::cur);
    }
}
```

```
        // Puisque la couleur n'est pas celle recherchée,  
        // on lève le bit « failbit ».  
  
        is.setstate(ios::failbit);  
    }  
    return;  
}  
  
int main() {  
  
    string LaCouleur;  
  
    // Une autre écriture du for, boucle à l'infini tant  
    // qu'il n'y a pas eu un arrêt ; par exemple  
    // un break ou un exit.  
  
    for (;;) {  
  
        // On récupère une couleur à partir du flux en entrée.  
  
        GetCouleur(cin, LaCouleur);  
  
        // Si « goodbit » est à zéro, cela signifie que aucun des  
        // autres bits « eofbit », « badbit » et « failbit » n'a  
        // été levé. De ce fait « good » doit retourner  
        // vrai (true). Sinon false.  
        // Dans cet exemple, « good » retourne vrai,  
        // si la couleur est celle recherchée, car dans ce  
        // cas « failbit » n'a pas été levé.
```

```
        if (cin.good()) break;
        cout << "une autre chance ...: ";

        // Ici, puisqu'il y a appel de clear sans argument,
        // le bit « goodbit » est remis à son état initial 0,
        // de même que pour les autres bits, afin de permettre
        // une seconde lecture ...

        cin.clear();

        // On ignore tous les caractères qui n'ont pas été lus
        // et qui se trouveraient dans le flux.

        cin.ignore(80, '\n');
    }

    // On ne peut arriver ici que si « goodbit » est à 0,
    // c.-à-d. « failbit » reste dans son état initial 0 (donc il
    // n'a pas été levé). Ce cas arrive si nous avons trouvé la
    // bonne couleur.

    cout << "félicitations!\n";

    return 0;
}
```

6. Associer un flux en entrée et en sortie à un fichier

- Dans le paragraphe « 2 » de ce chapitre, nous avons présenté les 3 classes permettant de manipuler des fichiers. Ces classes sont:
 - `ifstream`: l'accès du flux en lecture seulement,
 - `ofstream`: l'accès du flux en écriture seulement,
 - `fstream`: l'accès du flux en entrée et en sortie.

6.1. Les méthodes associées aux fichiers

- `open(fic_nom)`: elle ouvre le fichier dont le nom est « `fic_nom` », en utilisant les options par défaut.
- `open(fic_nom, flags)`: elle ouvre le fichier « `fic_nom` », en utilisant les options spécifiées par l'argument « `flags` ».
- `open(fic_nom, flags, prot)`: elle ouvre le fichier « `fic_nom` », en utilisant les options spécifiées par l'argument « `flags` », et en modifiant, grâce à l'argument « `prot` », les paramètres de protection associés au fichier.
- `close()`: elle ferme le fichier
- `is_open()`: elle retourne vrai (`true`) si le fichier a été ouvert, sinon faux (`false`) en cas d'échec.

6.2. Les bits associés à l'opération d'ouverture de fichiers

bit	signification	Mode C
in	lecture, le fichier doit exister	"r"
out	si le fichier existe, écrire par dessus, le créer si nécessaire	"w"
out trunc	si le fichier existe son contenu est perdu, le créer si nécessaire	"w"
out app	ajouter à la fin du contenu actuel du fichier, le créer si nécessaire	"a"
in out	lecture et écriture : position de départ est le début de fichier, le fichier doit exister	"r+"
in out trunc	si le fichier existe, son contenu est perdu, lecture et écriture dans un fichier, le fichier est créé si nécessaire	"w+"

- On peut ajouter dans cette description le bit « binary » qui signifie mode d'entrée (et/ou) sortie binaire. Par exemple un « in | binary » signifie lecture (entrée) binaire, un « out | binary » écriture (sortie) binaire dans le fichier, etc.

6.3. Les modes de protection

- Le mode de protection indique les droits d'accès que l'on désire associer au fichier.
- Par défaut, cette protection a la valeur « 0644 », c.-à-d. lecture/écriture (rwx: 110 = 6) pour celui qui est propriétaire du fichier, et lecture uniquement (rwx:100 = 4) pour le groupe et le monde extérieur.
- Pour plus de détails, voir la commande Unix « man chmod » (change mode).

6.4. Exemple d'ouverture/fermeture de fichiers

```
#include <iostream>

#include <fstream>

using namespace std;

int main() {

    // Première manière de l'écrire ...

    // Fichier par défaut en mode de lecture.

    ifstream f1;
    f1.open ("essai1.tmp");
    f1.close();

    // Fichier par défaut en mode d'écriture.

    ofstream f2;
    f2.open("essai2.tmp");
    f2.close();

    // Fichier ouvert en mode de lecture et écriture.

    fstream f3;
    f3.open ("essai3.tmp",ios::in | ios::out);
    f3.close();
```

```
// Une seconde forme d'écriture, sur une seule
// ligne de commande.

ifstream f4("essai4.tmp");

ofstream f5("essai5.tmp");
f5.close();

fstream f6("essai6.tmp",ios::in | ios::out);
f6.close();

// Avant d'ouvrir le fichier essai4.tmp on teste
// s'il est présent dans le répertoire, sinon on
// sort avec un message d'erreur.

if (!f4.is_open()) {
    cerr << "ne peut pas ouvrir le fichier essai4.tmp\n";
    exit (1);
}
f4.close();

return 0;
}
```

- **Sortie :** ne peut pas ouvrir le fichier essai4.tmp
- En mode de lecture, le fichier doit exister avant d'être lu. Or, vu que le fichier «essai4.tmp» n'existe pas, le message d'erreur "ne peut pas ..." est affiché.

6.5. Accès aléatoire

- La table suivante présente les différentes fonctions permettant de donner ou de modifier les positionnements dans un flux.

Classe	fonction membre	tâche
basic_istream<>	tellg()	retourne la position courante dans le flot en lecture
	seekg(pos)	se positionne en lecture à l'index <code>pos</code>
	seekg(offset, dir)	se positionne en lecture à l'index <code>offset</code> , par rapport à la direction définie par <code>dir</code> .
basic_ostream<>	tellp()	retourne la position courante dans le flot en écriture
	seekp(pos)	se positionne en écriture à l'index <code>pos</code>
	seekp(offset, dir)	se positionne en écriture à l'index <code>offset</code> , par rapport à la direction définie par <code>dir</code> .

- Ces fonctions font la distinction entre les flux ouverts en lecture ou en écriture par des lettres, `p` (« put ») pour l'écriture et `g` (« get ») pour la lecture.
- Pour les accès relatifs (l'argument `dir` dans `seekg` et `seekp`), l'`offset` peut être relatif aux 3 positions suivantes:

Constante	Signification
<code>beg</code>	la position est relative par rapport au début du flot
<code>cur</code>	la position est relative par rapport à la position courante
<code>end</code>	la position est relative par rapport à la fin du flot

```
#include <istream>
#include <fstream>
#include <string>

using namespace std;

int main(){

    const char *filename = "./test.txt";
    string entree;
    fstream inout;
    int curpos_rd,curpos_wrt;

    // Si le fichier existe déjà, son contenu est effacé.

    inout.open(filename, ios::trunc| ios::in | ios::out);

    if (!inout.is_open()) {
        cerr << "erreur à l'ouverture du fichier!\n";
        exit(1);
    }
    // Écriture du texte dans le fichier test.txt
    inout << "Ceci" << endl;
    inout << "est" << endl;
    inout << "un" << endl;
    inout << "fichier" << endl;
    inout << "de" << endl;
    inout << "test" ;
```

```
// On affiche les positionnements actuels.

curpos_wrt = inout.tellp();

cout << "position courante en mode écriture: "\
      << curpos_wrt << endl;

curpos_rd = inout.tellg();

cout << "position courante en mode de lecture: "\
      << curpos_rd << endl;

// On se positionne au début du fichier afin de
// lire le contenu du fichier du début.

inout.seekg(0);

// Lecture du texte.

while (!inout.eof()){
    inout >> entree;
    cout << entree << endl;
}

// La fin de fichier étant atteinte, eofbit est à 1,
// si on veut réaliser d'autres opérations de lecture
// et d'écriture sur le fichier nous devons mettre ce bit
// (de même que « failbit ») à zéro. Un « clear » remet
// donc « goodbit » à zéro qui signifie que tout marche!

inout.clear();
```

```
// Positionnement de l'index lecture à la fin du fichier.

inout.seekg(0, ios::end);
cout << "g: " << inout.tellg() << endl;

// Positionnement de l'index écriture à la fin du fichier.

inout.seekp(0, ios::end);
cout << "p: " << inout.tellp() << endl;

// Texte à ajouter.

inout << "On" << endl;
inout << "ajoute du texte!";

// On ferme le fichier.

inout.close();

return 0;
}
```

Sortie :

```
position courante en mode écriture: 27
position courante en mode de lecture: 27
Ceci
est
un
fichier
de
test
g: 27
p: 27
```

Dans le fichier test.txt, nous aurons ce qui suit:

```
Ceci
est
un
fichier
de
testOn
ajoute du texte!
```