Les flux

Introduction

Les flux sont le moyen utilisé pour réaliser des entrées/sorties en C++

Ils offrent un mécanisme standardisé permettant de lire ou d'écrire sur n'importe quel type de périphérique.

Intérêt

Le mécanisme des flux offre

- une plus grande souplesse : une donnée est traitée en fonction de son type, les conversions nécessaires devant être « automatiques »

- une extensibilité permettant
  - d'effectuer des E/S pour les nouveaux types de données de manière identique aux types de données prédéfinis
  - d'effectuer des E/S de la même manière que cela soit sur une console, un fichier ou dans une mémoire, etc…

Flux prédéfinis

- **cout** : flux de sortie écran (stdout du C)
- **cin** : flux d'entrée clavier (stdin du C)
- **cerr** : flux d'erreur écran (stderr du C)

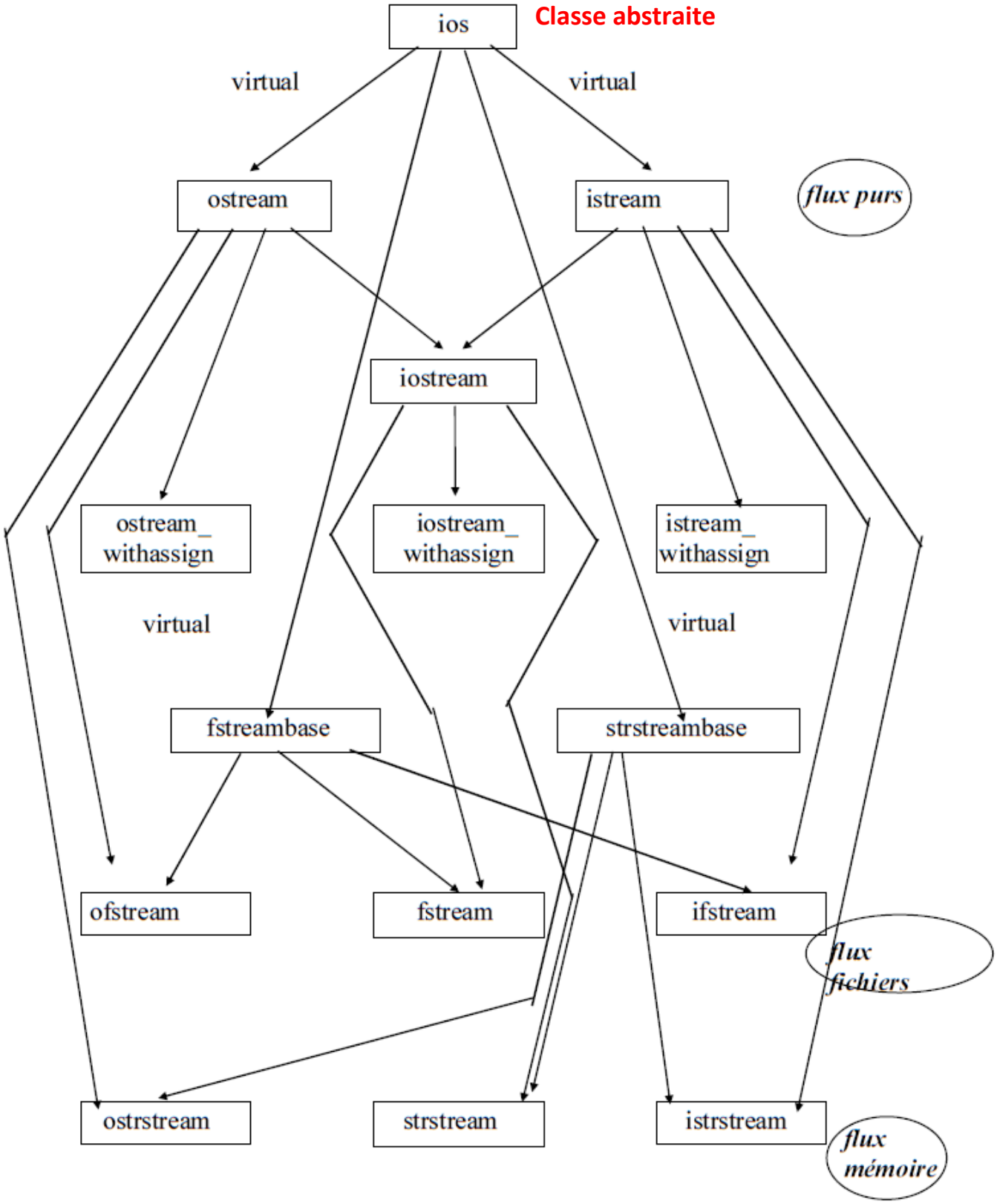
Définition

Un **flux** est une abstraction logicielle représentant un **flot de données** entre une source (le producteur) et une cible (le consommateur)

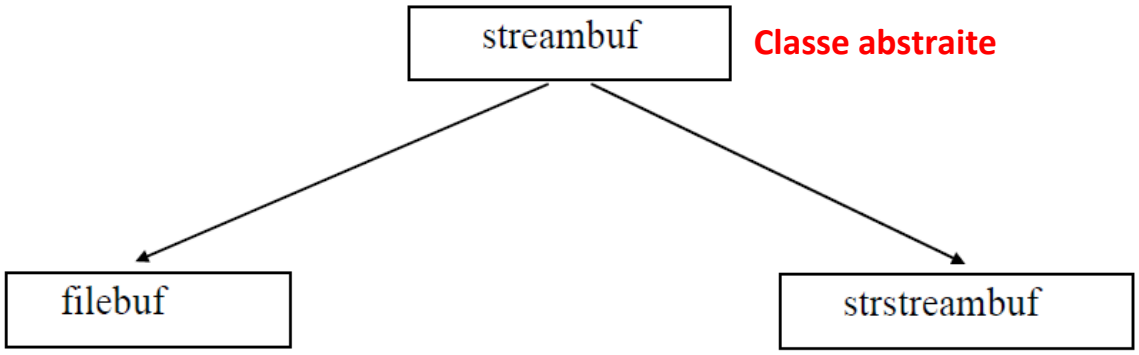
En fait, un **flux** (stream en anglais) correspond à tout organe avec lequel il est possible de communiquer des informations, en lecture et/ou en écriture

Les flux purs (classes **???stream**) ne communiquent en général pas directement avec les fichiers (ou les zones mémoires) : ils le font par l'intermédiaire d'un buffer (classes **???buf**) qui gère un mécanisme de tampons.

La hiérarchie des E/S formatées :



La hiérarchie des E/S non formatées :



La classe de base des E/S bufferisées : streambuf

Les E/S ne peuvent être effectivement réalisées qu'au moyen d'un buffer. Ce buffer est une instance d'une classe dérivée de la classe abstraite **streambuf**.

Cette classe fournit les mécanismes fondamentaux des E/S classiques utilisant un buffer.

Elle contient :

- un buffer d'E/S et des pointeurs de lecture et d'écriture

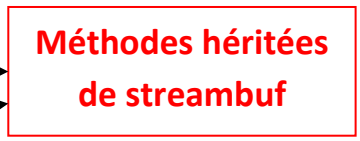
- des méthodes de lecture et d'écriture. Par exemple :
  - **int sgetc()** : envoie le caractère sous le pointeur de lecture, EOF s'il n'y en a plus.
  - **int snextc()** : avance le pointeur de lecture et fournit le caractère auquel on est parvenu, EOF si l'opération est impossible
  - **int sputc(char c)** : écrit le caractère précisé dans le buffer et avance le pointeur d'écriture

- Des méthodes de positionnement du type « seek »

Inutile d'entrer plus dans le détail de cette classe car il s'agit d'une classe **abstraite**. Ce sont les **classes dérivées** que nous utiliserons en pratique.

Exemple (la classe filebuf avec un peu d'avance ☺) :

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

int main()
{
  filebuf f;

  f.open("Fich.txt",ios::in);
  do
  {
    char c = f.sgetc();
    cout << c;
  }
  while(f.snextc() != EOF);
  f.close();

  cout << endl;
  return 0;
}
```



dont l'exécution fournit

```
bash-3.00$ cat Fich.txt
abcdefghijklmnopqrstuvwxyz
bash-3.00$ a.out
abcdefghijklmnopqrstuvwxyz

bash-3.00$
```

La classe **streambuf** constitue ce que l'on appelle un **flux « bas niveau »** (on travaille au niveau du byte), elle matérialise la connexion d'E/S.

Dans cet exemple, la classe filebuf matérialise la connexion avec le fichier « Fich.txt ».

La classe de base des E/S formatées : ios

Il s'agit d'une classe abstraite, qui est la classe de base commune à toutes les classes de la hiérarchie. Elle rassemble les caractéristiques communes à tous les flux formatés.

Elle contient :

- **La matérialisation d'une connexion d'E/S** : il s'agit en fait d'un **pointeur sur un objet de la classe streambuf**. L'accès à ce pointeur s'effectue à partir de la valeur renvoyée par la méthode

```
streambuf* rdbuf(void) ;
```

- **Le contrôle du flux** : méthodes permettant de **connaître l'état du flux à un instant donné**. L'état du flux est mémorisé dans une variable membre privée « state ». Les méthodes d'accès sont

```
int bad() ;

int fail() ;

int eof() ;

int good() ;

int rdstate() ;
```

De plus, l'opérateur « ! » et l'opérateur de casting « void* » ont été surchargés, ce qui permet de tester si le flux est dans l'état « bad » ou « fail », ou si une opération s'est bien déroulée ou pas :

```
if (!cout) … ;

if (cout << nb) … ;
```

Exemple

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

int main()
{
  if (!cout) cerr << "cout ne fonctionne pas..." << endl;
  else cout << "cout fonctionne" << endl;

  if (cout << "Essai affichage" << endl)
    cout << "Affichage reussi" << endl;
  else cerr << "Probleme d'affichage..." << endl;

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
cout fonctionne
Essai affichage
Affichage reussi
bash-3.00$
```

- **Le formatage des données** : il s'agit de la gestion des différents formats de données. Celle-ci s'effectue via une énumération qui reprend les différentes constantes permettant les différentes conversions :

```
// formatting flags
enum
{
  skipws = 0x0001, // skip whitespace on input
  left = 0x0002, // left-adjust output
  right = 0x0004, // right-adjust output
  internal = 0x0008, // padding after sign or base indicator
  dec = 0x0010, // decimal conversion
  oct = 0x0020, // octal conversion
  hex = 0x0040, // hexadecimal conversion
  showbase = 0x0080, // use base indicator on output
  showpoint = 0x0100, // force decimal point (floating output)
  uppercase = 0x0200, // upper-case hex output
  showpos = 0x0400, // add '+' to positive integers
  scientific= 0x0800, // use 1.2345E2 floating notation
  fixed = 0x1000, // use 123.45 floating notation
  unitbuf = 0x2000, // flush all streams after insertion
  stdio = 0x4000 // flush stdout, stderr after insertion
};
```

Cette énumération est manipulée par les fonctions

- `long setf(long flags) ;`
- `long unsetf(long flags) ;`

Exemple

```
int main()
{
  int n=100;

  // Affichage en decimal
  cout.setf(ios::dec);
  cout << "Decimal = " << n << endl;
  cout.unsetf(ios::dec);

  // Affichage en octal
  cout.setf(ios::oct);
  cout << "Octal = " << n << endl;
  cout.unsetf(ios::oct);

  // Affichage en hexadecimal
  cout.setf(ios::hex);
  cout << "Hexadecimal = " << n << endl;
  cout.unsetf(ios::hex);

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
Decimal = 100
Octal = 144          // 001 100 100
Hexadecimal = 64     // 0110 0100
bash-3.00$
```

D'autres méthodes sont disponibles :

- `int width(int largeur) ;`
- `char fill(char caractere_de_remplissage) ;`
- `int precision(int precision) ;`

Exemple

```
int main()
{
  float n=123.974;

  cout << "Normal = " << n << endl;

  cout.precision(10);
  cout << "precision(10) = " << n << endl;

  cout.setf(ios::scientific);
  cout << "scientific = " << n << endl;

  cout.width(20);
  cout.fill('-');
  cout << n << endl;

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
Normal = 123.974
precision(10) = 123.973999
scientific = 1.2397399902e+02
----1.2397399902e+02
bash-3.00$
```

Les manipulateurs

But : les manipulateurs sont des fonctions opérant sur un flux et dont le but est de **simplifier**, ou de rendre plus souple, **le formatage des données**.

Définition

Un manipulateur est une fonction opérant sur un flux d'E/S et qui est invoquée au sein même d'une insertion ou d'une extraction.

Exemple

Au lieu d'écrire

```
cout.precision(10) ;
cout << n << endl ;
```

On peut utiliser le **manipulateur « setprecision »** et écrire :

```
cout << setprecision(10) << n << endl ;
```

La souplesse acquise est évidente. Mais, **endl** est aussi un manipulateur ! → il existe bon nombre de manipulateurs prédéfinis

Manipulateurs associées à la classe ios

a) Manipulateurs non paramétrés

- dec
- hex
- oct

qui permettent d'écrire :

```
cout << dec << n << hex << n << endl ;
```

Le prototype de ces manipulateurs est :

```
ios& dec(ios &) ;
ios& hex(ios &) ;
ios& oct(ios &) ;
```

En effet,

```
cout << hex << n ;
```

est équivalent à

```
(cout.operator<<(hex)).operator<<(n) ;
```

et la classe ostream, dont **cout** est une instance, possède la méthode

```
ostream& operator<< (ios& (*pf)(ios&));
```

Exemple :

```
int main()
{
  int n=100;

  cout << "Decimal = " << dec << n << endl;
  cout << "Octal = " << oct << n << endl;
  cout << "Hexadecimal = " << hex << n << endl;

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
Decimal = 100
Octal = 144
Hexadecimal = 64
bash-3.00$
```

b) Manipulateurs paramétrés

- `setw(int largeur) ;`
- `setfill(int char caractere_de_remplissage) ;`
- `setprecision(int precision) ;`
- `setiosflags(long flags) ;`
- …

qui nécessitent l'inclusion de la librairie **iomanip.h**

Par exemple,

```
cout << setprecision(10) << n;
```

est équivalent à

```
operator<<(cout,setprecision(10)).operator<<(n);
```

Exemple :

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

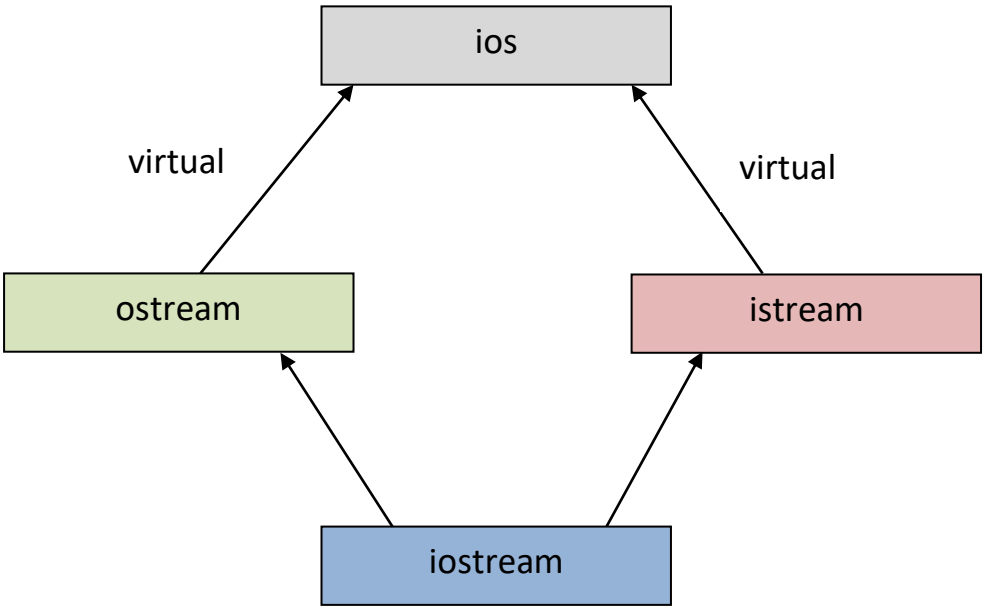
```
int main()
{
  float n=123.974;

  cout << "Normal = " << n << endl;
  cout << "precision(10) = " << setprecision(10) << n << endl;
  cout << "scientific = " << setiosflags(ios::scientific) << n << endl;
  cout << setw(20) << setfill('-') << n << endl;

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
Normal = 123.974
precision(10) = 123.973999
scientific = 1.2397399902e+02
----1.2397399902e+02
bash-3.00$
```

Les flux d'entrée/sortie : istream, ostream, iostream

Les classes **ostream** et **istream** fournissent tout ce qui est nécessaire à une opération d'E/S **sans format particulier**.



1) La classe ostream

Cette classe contient toutes les surcharges classiques de l'opérateur d'insertion **"<<"**.

Ces surcharges réalisent toutes les conversions nécessaires pour les données numériques alors que les chaînes de caractères sont affichées telles quelles.

Si la donnée insérée est l'adresse d'un objet streambuf, c'est l'ensemble du flux qui est inséré sur le flux cible.

Méthodes définies dans la classe **ostream** :

```
ostream& operator<< (signed char);
ostream& operator<< (unsigned char);

// for the following, insert character representation of numeric value
ostream& operator<< (short);
ostream& operator<< (unsigned short);
ostream& operator<< (int);
ostream& operator<< (unsigned int);
ostream& operator<< (long);
ostream& operator<< (unsigned long);
ostream& operator<< (float);
ostream& operator<< (double);
ostream& operator<< (long double);

// insert the null-terminated string
ostream& operator<< (const signed char *);
ostream& operator<< (const unsigned char *);

// insert character representation of the value of the pointer
ostream& operator<< (void *);

// extract from streambuf, insert into this ostream
ostream& operator<< (streambuf *);

// manipulators
ostream& operator<< (ostream& ( *_f)(ostream&));
ostream& operator<< (ios& ( *_f)(ios&));
```

Constructeur :

```
ostream(streambuf *buffer);
```

Autres méthodes gérant les sorties :

```
ostream& put (char caractere);
ostream& write (char *chaine, int longueur);

long tellp();
ostream& seekp (long position_absolue);
```

```
ostream seekp (long deplacement, seek_dir origine);
// où stream_seek est une enumeration
// enum seek_dir { beg=0, cur=1, end=2 };

ostream& flush();
// → vide le buffer. Quel buffer ? celui qui est associé à la classe
// par son constructeur.

int opfx() ;  // vérifie que le flux est en état
void osfx() ; // nettoie le buffer associé
```

Enfin, à la classe **ostream** sont associés **3 manipulateurs** :

- **endl** : qui insère un retour de chariot sur le flux
- **ends** : qui insère un zéro de fin de chaîne sur le flux
- **flush** : qui vide le buffer de sortie

dont les prototypes sont

```
ostream& endl(ostream&);
ostream& ends(ostream&);
ostream& flush(ostream&);
```

2) Créer ses propres manipulateurs

Il est possible de définir ses propres manipulateurs. Leur syntaxe doit avoir la forme suivante

```
ostream& nom_du_manipulateur(ostream& parametre_flux) ;
```

Cela est possible car la classe ostream dispose de la méthode

```
ostream& operator<< (ostream& (*_f)(ostream&));
```

Exemple

```
ostream& tab(ostream& s)
{
  s << "\t";
  return s;
}

ostream& inverse(ostream& s)
```

```
{
  s << (char)27 << '[' << "7m";
  return s;
}

ostream& normal(ostream& s)
{
  s << (char)27 << "[0m";
  return s;
}

int main()
{
  int a=1,b=2,c=3;

  cout << a << tab << b << tab << c << endl;
  cout << inverse << a << tab << b << tab << normal << c << endl;

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
1       2       3
1       2       3
bash-3.00$
```

3) La classe istream

Cette classe contient toutes les surcharges classiques de l'opérateur d'extraction **">>"**, réalisant les conversions nécessaires :

```
istream& operator>> (signed char *);
istream& operator>> (unsigned char *);
istream& operator>> (unsigned char&);
istream& operator>> (signed char&);
istream& operator>> (short&);
istream& operator>> (int&);
istream& operator>> (long&);
istream& operator>> (unsigned short&);
istream& operator>> (unsigned int&);
istream& operator>> (unsigned long&);
istream& operator>> (float&);
istream& operator>> (double&);
istream& operator>> (long double&);

istream& operator>> (streambuf *);
```

```
istream& operator>> (istream& ( *_f)(istream&));
istream& operator>> (ios& ( *_f)(ios&) );
```

Constructeur :

```
istream (streambuf * buffer);
```

Autres méthodes gérant les entrées :

```
istream& get(char caractere);
int get();

istream& get(char *chaine, char delimiteur = '\n');
// lecture jusqu'au délimiteur

istream& get(char *chaine, int longueur, char delimiteur='\n');
// lecture du nombre précisé de caractères à moins de
// rencontrer le délimiteur; le zéro de fin de chaîne est ajouté

istream& getline(char *chaine,int longueur, char delimiteur='\n');
// idem, mais c'est le délimiteur qui est copié dans la chaîne

istream& read(char *chaine, int longueur);

long tellg();
istream& seekg(long position_absolue);
istream seekg(long déplacement, seek_dir origine);

int ipfx(int n = 0);
// vide le flux si n est nul ou si n est inférieur au
// nombre de caractères disponibles.
void isfx()
// nettoie le buffer
```

4) La classe iostream

Cette classe hérite à la fois de **istream** et de **ostream** (et donc aussi de ios de manière virtuelle). Elle contient donc l'ensemble des méthodes permettant d'accéder de manière standard à un périphérique d'entrée ou à un périphérique de sortie.

Elle sera utilisée comme classe de base :

- de la classe **fstream** : classe spécialisée dans les opérations d'E/S sur fichier(s)
- de la classe **strstream** : classe spécialisée dans les opérations d'E/S en mémoire

Le buffer des fichiers classiques : la classe filebuf

La classe filebuf permet de créer un objet qui matérialise la connexion avec un fichier.

Son **constructeur** le plus simple est :

```
filebuf();
```

Dans ce cas, l'ouverture du fichier s'effectuera à l'aide de la méthode :

```
filebuf* open(const char* nom_fichier,int mode_ouverture) ;
```

La fonction open ouvre le fichier dans le mode d'ouverture qui lui est spécifié et l'attache à l'objet filebuf courant.

Les différents modes d'ouverture sont définis à l'aide de l'énumération suivante (de la classe ios) :

```
enum open_mode
{
  in = 0x01,         // open for reading
  out = 0x02,        // open for writing
  ate = 0x04,        // seek to eof upon original open
  app = 0x08,        // append mode: all additions at eof
  trunc = 0x10,      // truncate file if already exists
  nocreate = 0x20,   // open fails if file doesn't exist
  noreplace= 0x40,   // open fails if file already exists
};
```

La méthode

```
filebuf* close() ;
```

vide le buffer, ferme le fichier et détruit le lien qui reliait celui-ci au buffer.

Enfin la méthode

```
int is_open() const ;
```

teste si le fichier est ouvert ou pas.

Exemple : réécriture « simplifiée » de la commande cat

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

int main(int argc,char* argv[])
{
  filebuf f;

  f.open(argv[1],ios::in);
  if (f.is_open()) cout << &f;
  else cout << "*** [fichier " << argv[1] << " non trouve]" << endl;
  f.close();

  return 0;
}
```

dont un exemple d'exécution fournit

```
bash-3.00$ cat Fich.txt
abcdefghijklmnopqrstuvwxyz
bash-3.00$ a.out Fich.txt
abcdefghijklmnopqrstuvwxyz
bash-3.00$
```

Gestion classique des fichiers : classes [i|o]stream et filebuf

Pour traiter un fichier, il faut associer une instance de filebuf à un flux (de type output, input, ou les deux selon le cas).

La démarche consiste donc à

1. **Ouvrir un fichier et l'attacher à un buffer de type filebuf**

```
filebuf fichier;
if (!fichier.open(nom_fichier,mode_ouverture))
{
  // Erreur d'ouverture
  exit(1) ;
}
```

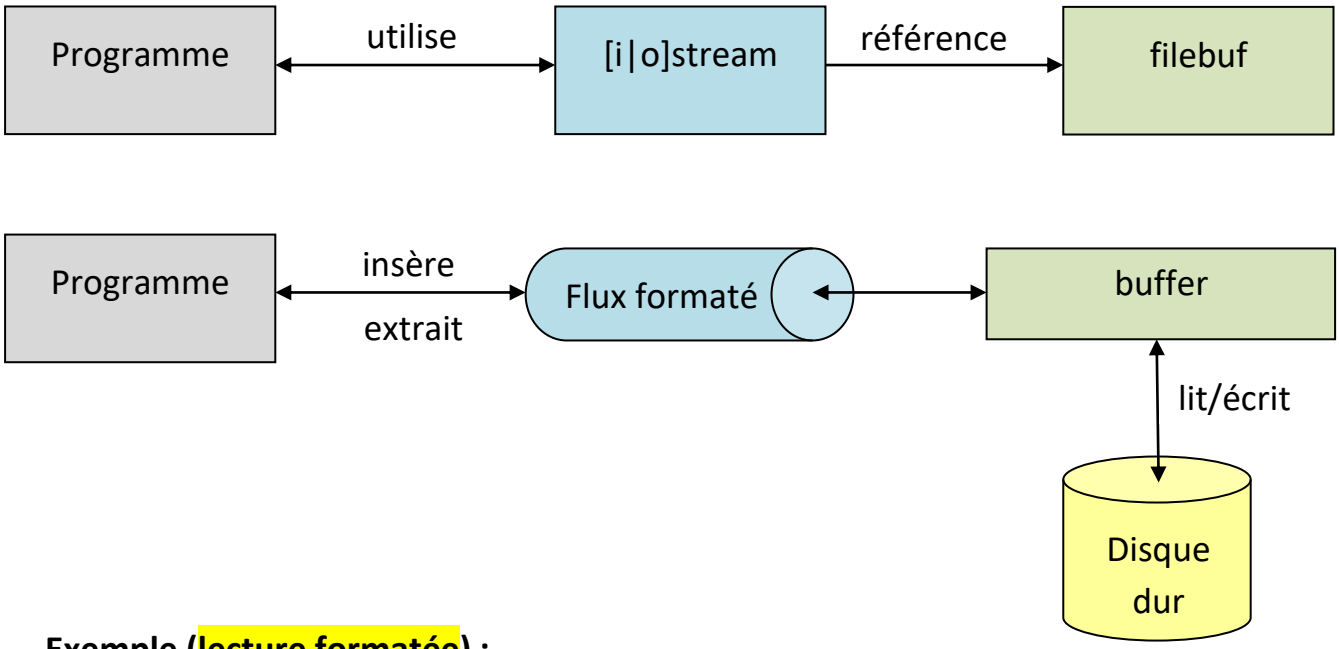
2. **Associer ce filebuf à un flux au moyen du constructeur de ce flux**

```
ostream flux(&fichier);
```

Cela revient à dire que l'on associe un objet « flux » à un objet « buffer » :

- Le premier s'occupe du formatage des données
- Le second réalise les E/S de bas niveau

Schématiquement, nous avons un **modèle à 2 couches** :



Exemple (lecture formatée) :

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

int main(int argc,char* argv[])
{
  filebuf fichier;

  if (!fichier.open(argv[1],ios::in))
  {
    cout << "Erreur d'ouverture du fichier...";
    return 1;
  }

  istream flux(&fichier);
  char c;
  flux >> c; // lecture formatée dans le fichier
  while(!flux.eof())
  {
    cout << c;
    flux >> c; // lecture formatée dans le fichier
  }
```

```
  cout << endl;

  fichier.close();
  return 0;
}
```

dont un exemple d'exécution fournit

```
bash-3.00$ cat Fich.txt
abcdefghijklmnopqrstuvwxyz
bash-3.00$ a.out Fich.txt
abcdefghijklmnopqrstuvwxyz
bash-3.00$
```

Exemple (lecture non formatée) :

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

int main(int argc,char* argv[])
{
  filebuf fichier;

  if (!fichier.open(argv[1],ios::in))
  {
    cout << "Erreur d'ouverture du fichier...";
    return 1;
  }

  istream flux(&fichier);
  char c;
  flux.read(&c,sizeof(char)); // lecture non formattée
  while(!flux.eof())
  {
    cout << c;
    flux.read(&c,sizeof(char)); // lecture non formattée
  }
  cout << endl;

  fichier.close();
  return 0;
}
```

dont un exemple d'exécution fournit

```
bash-3.00$ cat Fich.txt
abcdefghijklmnopqrstuvwxyz
bash-3.00$ a.out Fich.txt
abcdefghijklmnopqrstuvwxyz

bash-3.00$
```

Exemple (écriture/lecture non formatée) :

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

int main(int argc,char* argv[])
{
  filebuf fichier;

  if (!fichier.open("essai_i",ios::out))
  {
    cout << "Erreur d'ouverture du fichier...";
    return 1;
  }

  ostream flux(&fichier);
  int n;
  for (int i=10 ; i>0 ; i--)
  {
    n = i*5 - 6;
    cout << i << " (I) on ecrit " << n << endl;
    flux.write((char*)&n,sizeof(int)); // écriture non formatée
  }
  flux.flush();
  fichier.close();

  cout << "relecture" << endl;

  if (!fichier.open("essai_i",ios::in))
  {
    cout << "Erreur d'ouverture du fichier...";
    return 1;
  }

  istream fluxi(&fichier);
  fluxi.read((char*)&n,sizeof(int)); // lecture non formatée
  while(!fluxi.eof())
  {
```

```
    cout << "- on a lu " << n << endl;
    fluxi.read((char*)&n,sizeof(int));
  }
  fichier.close();

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
10 (I) on ecrit 44
9 (I) on ecrit 39
8 (I) on ecrit 34
7 (I) on ecrit 29
6 (I) on ecrit 24
5 (I) on ecrit 19
4 (I) on ecrit 14
3 (I) on ecrit 9
2 (I) on ecrit 4
1 (I) on ecrit -1
relecture
- on a lu 44
- on a lu 39
- on a lu 34
- on a lu 29
- on a lu 24
- on a lu 19
- on a lu 14
- on a lu 9
- on a lu 4
- on a lu -1
bash-3.00$ cat essai_i
,'"
ÿÿÿÿ▒_-3.00$
▒_-3.00$
```

On constate que le fichier créé est binaire, aucun formatage n'a été réalisé.

Exemple (écriture/lecture formatée) :

```
int main(int argc,char* argv[])
{
  filebuf fichier;

  if (!fichier.open("essai_i",ios::out))
  {
    cout << "Erreur d'ouverture du fichier...";
    return 1;
  }

  ostream flux(&fichier);
  int n;
  for (int i=10 ; i>0 ; i--)
  {
    n = i*5 - 6;
    cout << i << " (I) on ecrit " << n << endl;
    flux << n << endl; // écriture formatée
  }
  flux.flush();
  fichier.close();

  cout << "relecture" << endl;

  if (!fichier.open("essai_i",ios::in))
  {
    cout << "Erreur d'ouverture du fichier...";
    return 1;
  }

  istream fluxi(&fichier);
  fluxi >> n;  // lecture formatée
  while(!fluxi.eof())
  {
    cout << "- on a lu " << n << endl;
    fluxi >> n;  // lecture formatée
  }
  fichier.close();

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
10 (I) on ecrit 44
9 (I) on ecrit 39
8 (I) on ecrit 34
7 (I) on ecrit 29
6 (I) on ecrit 24
5 (I) on ecrit 19
4 (I) on ecrit 14
3 (I) on ecrit 9
2 (I) on ecrit 4
1 (I) on ecrit -1
relecture
- on a lu 44
- on a lu 39
- on a lu 34
- on a lu 29
- on a lu 24
- on a lu 19
- on a lu 14
- on a lu 9
- on a lu 4
- on a lu -1
bash-3.00$ cat essai_i
44
39
34
29
24
19
14
9
4
-1
bash-3.00$
```

On constate ici que le fichier créé est au format texte, l'écriture ayant été formatée.

Gestion classique des fichiers : classes [i|o]fstream

Les flux

- **fstream** (filebuf + stream)
- **ofstream** (filebuf + stream en sortie)
- **ifstream** (filebuf + stream en entrée)

sont des flux spécialement orientés vers les fichiers sur disques.

Ces trois classes dérivent de la classe fstreambase qui définit bon nombre de méthodes communes aux trois classes. On y trouve, en particulier, une variable membre privée de type **filebuf**.

Les constructeurs de ces classes créent automatiquement cet objet filebuf et l'attachent à l'objet flux ainsi instancié → le lien avec l'objet (filebuf) gérant le buffer associé au fichier s'effectue automatiquement.

Ce buffer est accessible au moyen de la méthode

```
filebuf* rdbuf()  const ;
```

Constructeurs :

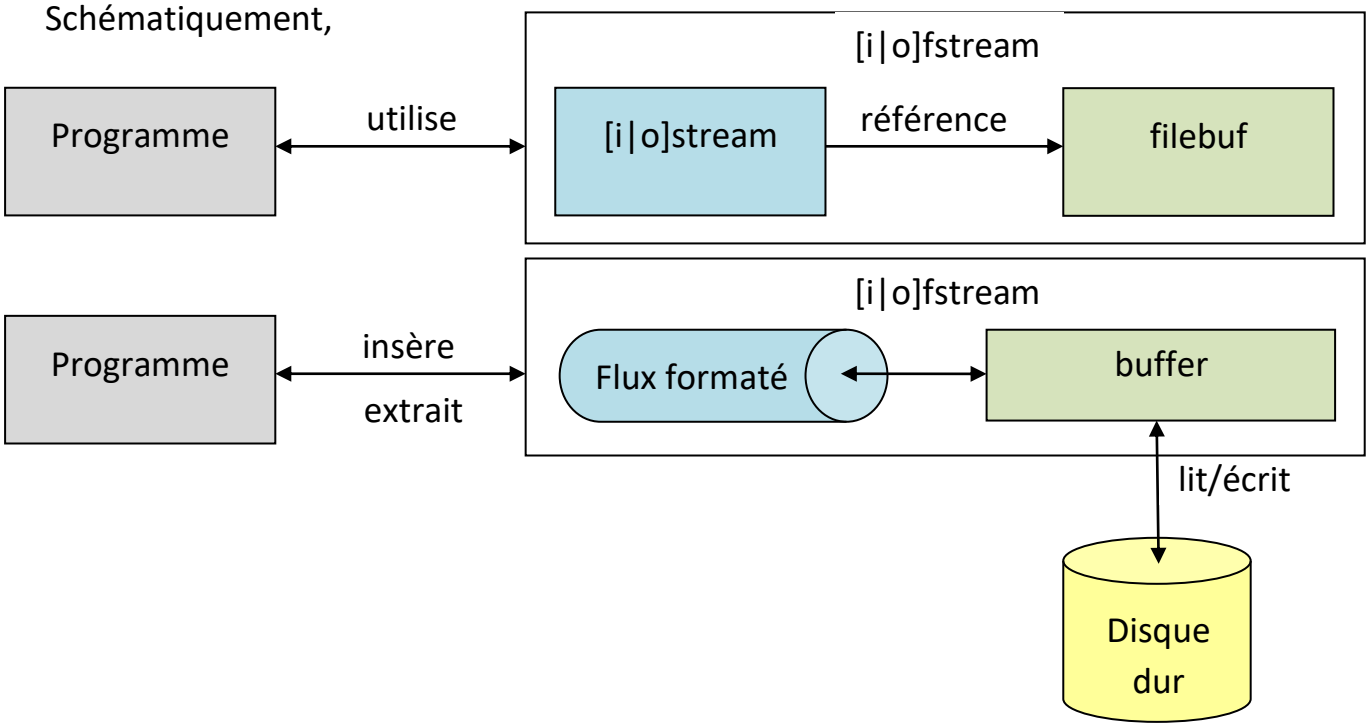
Les deux formes les plus simples sont :

- `(i|o)fstream(const char* nom_fichier,int mode_ouverture);`
- `(i|o)fstream()` : qui n'ouvre pas immédiatement le fichier. Celui-ci pourra être ouvert ultérieurement au moyen de la méthode héritée **open()**.

La méthode **is_open()** permet à nouveau de vérifier le succès ou l'échec de l'opération.

La méthode héritée **close()** referme évidemment le fichier.

Schématiquement,



Exemple :

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

struct copain
{
  char szNom[20], szPrenom[25];
  unsigned int nbreEnfants;
  float flArgentPrete;
  char chSur;

  void affiche();
};

void copain::affiche()
{
  cout << szNom << " " << szPrenom << endl;
  cout << " *** enfants : " << nbreEnfants;
  cout << " *** somme : " << flArgentPrete;
  cout << " *** ";
  if (strchr("OoYy",chSur)) cout << " confiance";
  else cout << " pas confiance";
  cout << endl;
}


void ecritFichier(char * nf);
```

```
void litFichier(char * nf);

int main()
{
  char nomFic[40];

  cout << "Nom du fichier : ";
  cin >> nomFic;
  ecritFichier(nomFic);
  litFichier(nomFic);

  return 0;
}

void ecritFichier(char * nf)
{
  ofstream fichier(nf, ios::out);
  if (!fichier)
  {
    cout << "erreur d'ouverture !" << endl;
    exit(1);
  }
  char nom[20];
  copain unCopain;
  cout << "Premier nom : "; cin >> nom;
  while (strcmp(nom,"/"))
  {
    strcpy(unCopain.szNom, nom);
    cout << "Prenom : ";cin >> unCopain.szPrenom;
    cout << "Combien d'enfants ? "; cin >> unCopain.nbreEnfants;
    cout << "Argent prete : "; cin >> unCopain.flArgentPrete;
    cout << "Est-il sur ? ";
    do
      cin >> unCopain.chSur;
    while (!strchr ("ONonYy",unCopain.chSur));
    fichier.write((char *)&unCopain, sizeof(copain));
    cout << "Nom suivant (/ si fini) : ";cin >> nom;
  }
  fichier.close();
}

void litFichier(char * nf)

{
  ifstream fichier(nf,ios::in);
  if (!fichier.rdbuf()->is_open())
  {
    cout << "erreur d'ouverture !" << endl;
    exit(1);
  }
```

```
  copain unCopain;
  fichier.read((char *)&unCopain, sizeof(copain));
  while (!fichier.eof())
  {
    unCopain.affiche();
    fichier.read((char *)&unCopain, sizeof(copain));
  }
  fichier.close();
}
```

dont un exemple d'exécution fournit

```
bash-3.00$ a.out
Nom du fichier : Copains.dat
Premier nom : Merce
Prenom : Denys
Combien d'enfants ? 3
Argent prete : 120
Est-il sur ? o
Nom suivant (/ si fini) : Vilvens
Prenom : Claude
Combien d'enfants ? 2
Argent prete : 300
Est-il sur ? o
Nom suivant (/ si fini) : /
Merce Denys
 *** enfants : 3 *** somme : 120 ***  confiance
Vilvens Claude
 *** enfants : 2 *** somme : 300 ***  confiance
bash-3.00$ ls -l Copains.dat
-rw-r--r--   1 student  other        120 oct. 28 17:01 Copains.dat
bash-3.00$
```

Les flux mémoires

De la même manière que l'on a associé un flux à un buffer « fichier », on peut associer un flux à un buffer « mémoire » qui est traité comme une chaîne de caractères.

On réalise ainsi l'équivalent des fonctions **sprintf** et **sscanf** du C.

1) Un buffer pour une zone mémoire

La classe **strstreambuf**, dérivée de streambuf, permet la gestion d'une zone mémoire comme une sorte de tableau.

Une méthode utile est

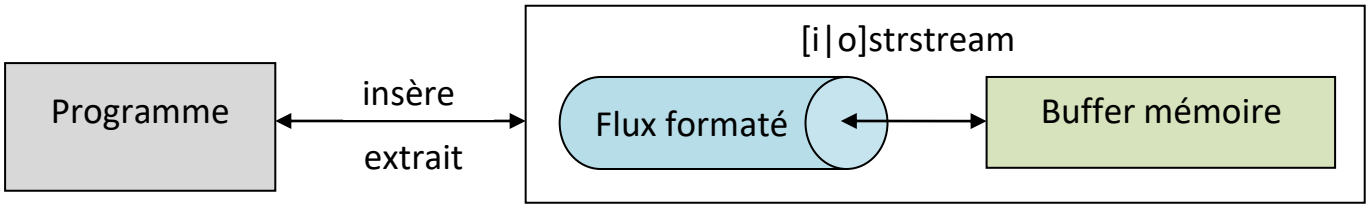
```
char* str();
```

qui renvoie un pointeur sur ce tableau interne.

2) Les classes [i|o]strstream

Ces classes jouent un rôle similaire aux classes [i|o]fstream, la zone mémoire remplaçant le fichier.

Schématiquement,



On y trouve une variable membre de type strstreambuf.

Constructeurs :

- **`[i|o]strstream()`** : qui utilise un buffer dynamique interne, initialement vide.

Exemple :

```
ostrstream maPhrase;
maPhrase << « Travaillez, prenez de la peine » << ends ;
cout << maPhrase.str() << endl ;
```

- **`[i|o]strstream(char* chaine,int longueur_chaine)`** : qui crée un flux mémoire de longueur_chaine caractères à partir de l'adresse donnée par chaine.

Exemple :

```
char phrase2[60] ;
ostrstream maPhrase2(phrase2,60) ;
maPhrase2 << « Des tetes bien faites » << ends ;
cout << phrase2 << endl ;
```

L'exemple complet :

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

int main()
{
  char *phrase;
  ostrstream maPhrase;

  maPhrase << "Travaillez, prenez de la peine" << ends;
  phrase = maPhrase.str();
  cout << phrase << endl;

  char phrase2[60];
  ostrstream maPhrase2(phrase2,60);
  maPhrase2 << "Des tetes bien faites" << ends;
  cout << phrase2 << endl;

  return 0;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
Travaillez, prenez de la peine
Des tetes bien faites
bash-3.00$
```

3) La sauvegarde dans une zone mémoire

L'intérêt majeur des flux en mémoire est de sauvegarder un affichage formaté en mémoire, à la manière du **sprintf en C**.

Exemple :

```
#include <iostream>
#include <iomanip>
#include <strstream>
```

```
using namespace std;

#include <math.h>
const int NV = 6;

int main()
{
  char phrase[200];
  ostrstream maphrase(phrase,sizeof(phrase));

  maphrase << "Travaillez, prenez de la peine" << ends;
  cout << maphrase.str() << endl;

  maphrase.seekp(0,ios::beg);

  int tab[NV];
  maphrase << "Vecteur de base : (";
  for (int i=0; i<NV; i++)
  {
    tab[i] = i*5+20 ;
    cout << tab[i] << endl;
    maphrase << setprecision(4) << setw(8) << sqrt(tab[i]) << ",";
  }

  maphrase << '\x08';
  maphrase << ")" << ends;

  cout << phrase << endl;
}
```

dont l'exécution fournit

```
bash-3.00$ a.out
Travaillez, prenez de la peine
20
25
30
35
40
45
Vecteur de base : (   4.472,       5,   5.477,   5.916,   6.325,   6.708)
bash-3.00$
```

Les flux mémoires : correspondances avec la classe string

Les classes **ostrstream** et **istrstream** permettent de manipuler des flux mémoire pour créer ou « parser » des chaines de caractères au format **« char [] »**. Cependant, le C++ offre des outils puissants comme la classe **string** pour gérer les chaînes de caractères, simplifiant et rendant les développements plus rapides. Il serait donc dommage de s'en passer. Des flux mémoire associés à la classe **string** existent donc :

- **ostringstream**
- **istringstream**

qui sont les équivalents des classes ostrstream et istrstream mais pour la classe string.

Utilisations classiques

- **Formatage et « déformatage » (« parsing ») de chaînes** : **ostringstream** est souvent utilisé pour créer des chaînes de caractères formatées, tandis que **istringstream** est utilisé pour analyser ou lire des données formatées à partir d'une chaîne.
- **Conversion de types** : Ces classes sont utiles pour convertir des types en chaînes de caractères et vice versa, par exemple, convertir un entier en chaîne ou extraire un entier d'une chaîne.
- **Traitement des données structurées** : **istringstream** est couramment utilisé pour **parser des fichiers texte** où les données sont structurées sous forme de chaînes, comme un **fichier CSV**.

Ces classes simplifient beaucoup les tâches de manipulation de chaînes en permettant un flux de données vers et depuis des chaînes de caractères.

La classe std::ostringstream

La classe **std::ostringstream** permet de créer une chaîne de caractères en formatant et en ajoutant différentes données comme on le ferait avec **std::cout** → particulièrement pratique lorsqu'on a besoin de construire dynamiquement une chaîne de caractères à partir de plusieurs données de types différents (entiers, flottants, chaînes, etc.).

Soit le fichier **Test_ostringstream.cpp** :

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main() {
  ostringstream oss;
  int age = 30;
  string name = "Alice";

  oss << "Name: " << name << ", Age: " << age;

  string result = oss.str();  // Convertir le flux en string

  cout << result << endl;     // Affiche: "Name: Alice, Age: 30"
  return 0;
}
```

où on observe que

- L'objet **oss** s'utilise exactement de la même façon que **cout**, et peut être configuré comme lui (voir précédemment) pour le formatage
- La méthode **str()** de la classe **ostringstream** permet de récupérer l'objet string formaté correspondant

Sa compilation et son exécution fournissent :

```
[student@moon]$ g++ Test_ostringstream.cpp -o Test_ostringstream
[student@moon]$ Test_ostringstream
Name: Alice, Age: 30
[student@moon]$
```

La classe istringstream

La classe **std::istringstream** permet de lire des données d'une chaîne de caractères, ce qui est utile pour parser ou extraire des valeurs d'une chaîne formatée → particulièrement pratique lorsque vous avez une chaîne de caractères contenant des données structurées (comme des nombres ou des mots séparés par des espaces), et que vous voulez extraire ces données.

Soit le fichier **Test_istringstream1.cpp** :

```
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

int main() {
  string data = "Alice 30";
  istringstream iss(data);

  string name;
  int age;

  iss >> name >> age;

  cout << "Name: " << name << ", Age: " << age << endl;
  return 0;
}
```

où on observe que

- Le flux **iss** de type **istringstream** a été « monté » sur la chaîne de caractères **data**, donnée que l'on souhaite parser
- L'objet **iss** s'utilise alors comme **cin**

Sa compilation et son exécution fournissent

```
[student@moon]$ g++ Test_istringstream1.cpp -o Test_istringstream1
[student@moon]$ Test_istringstream1
Name: Alice, Age: 30
[student@moon]$
```

Il faut cependant remarquer que si les données ne sont pas correctement formatées dans la variable **data**, la ligne

```
iss >> name >> age ;
```

peut provoquer des erreurs si age ne correspond pas à un entier par exemple. Il est donc préférable de récupérer les « token » sous forme de chaînes de caractères et d'ensuite tenter les conversions.

Le fichier **Test_istringstream2.cpp** en fournit un exemple :

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

```
using namespace std;

int main() {
  string data = "Alice;30;Engineer";

  // Créer un istringstream à partir de la chaîne de caractères
  istringstream iss(data);

  string name;
  string ageStr;  // Nous l'extrayons d'abord comme une chaîne
  string profession;

  // Lire les champs séparés par des ;
  getline(iss, name, ';');
  getline(iss, ageStr, ';');
  getline(iss, profession, ';');

  // Conversion de l'âge en entier
  int age = stoi(ageStr);

  // Afficher les valeurs extraites
  cout << "Name: " << name << ", Age: " << age << ", Profession: " <<
profession << endl;
  return 0;
}
```

où observe que

- Nous avons utilisé la fonction **std::getline()** à la place de **l'opérateur >>**, assurant ainsi une lecture correcte de la chaîne → on retrouve ici la manière de procéder de la fonction **strtok() du C**
- La fontion **std::stoi()** permet de convertir proprement une chaîne de caractère en **int** → dans le cas où **ageStr** ne correspondrait pas à un int, une exception serait lancée et il faudrait la gérer par un try…catch

La compilation et l'exécution du fichier fournit :

```
[student@moon]$ g++ Test_istringstream2.cpp -o Test_istringstream2
[student@moon]$ Test_istringstream2
Name: Alice, Age: 30, Profession: Engineer
[student@moon]$
```