Les entrées-sorties sur fichiers
(en mode séquentiel)

IMPORTANT: relire d'abord les E/S conversationnelles

Objectifs:

- Fédérer les concepts d'entrée-sortie
- Manipuler les fichiers en lecture/écriture
- Introduire le concept d'automate pour la lecture

Plan:

- vue d'ensemble: mémoire volatile et permanente
- opérations de base sur les fichiers
- difference entre "binaire" et "format"
- lecture/écriture formatée
- fichier de configuration: automate de lecture (Série0)

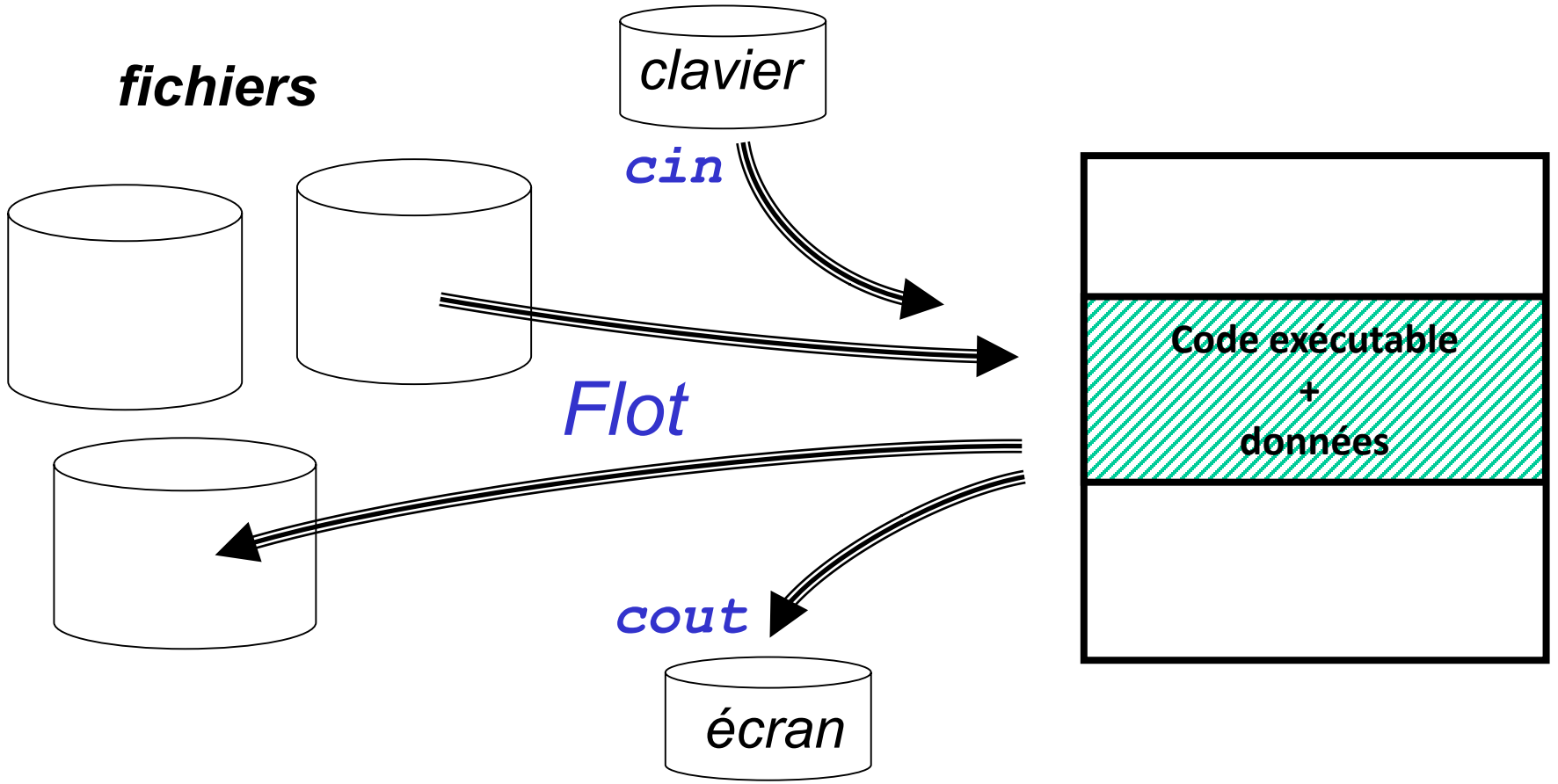
Mémoire permanente / Mémoire volatile et Flot

Mémoire externe (disque dur, clef USB..)

permanente

Mémoire centrale

volatile



Usage d'un flot en lecture d'un fichier, en accès séquentiel

```
#include <iostream>
#incude  <fstream>  // type ifstream pour ouvrir un fichier en lecture
                    // autres types: ofstream pour fichier en écriture, fstream pour lecture/ecriture
```

1) Ouverture du fichier "nomfichier.txt" en lecture à l'aide de la variable fichier puis test de l'échec

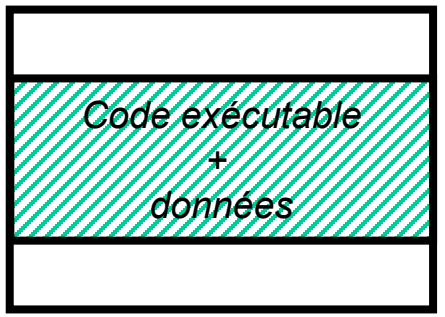
```
ifstream fichier("nomfichier.txt"); // open() est automatiquement appelée
if(fichier.fail()) exit();          // si le fichier n'existe pas
```

"nomfichier.txt"

La lecture commence au début du fichier et consomme son contenu dans l'ordre d'apparition (en séquence)

2) Une ou plusieurs opération(s) de lecture sur la variable fichier avec l'opérateur >>

Attention: seulement le passage par référence est autorisé pour une variable de type fstream



3) Fermeture du fichier `fichier.close();`

Quelques contraintes générales sur les flots

Par défaut, l'ouverture d'un fichier en écriture efface son contenu

Il faut ouvrir le flot en mode `ofstream::app` pour ajouter du contenu à la suite

Exclusivité de l'accès à un fichier:

Un flot ouvert sur un fichier bloque l'accès à ce fichier par un autre flot

Il faut fermer le fichier avant de l'ouvrir avec un autre flot

Exclusivité de l'usage d'un flot :

Un flot ne peut être ouvert que sur un seul fichier à la fois

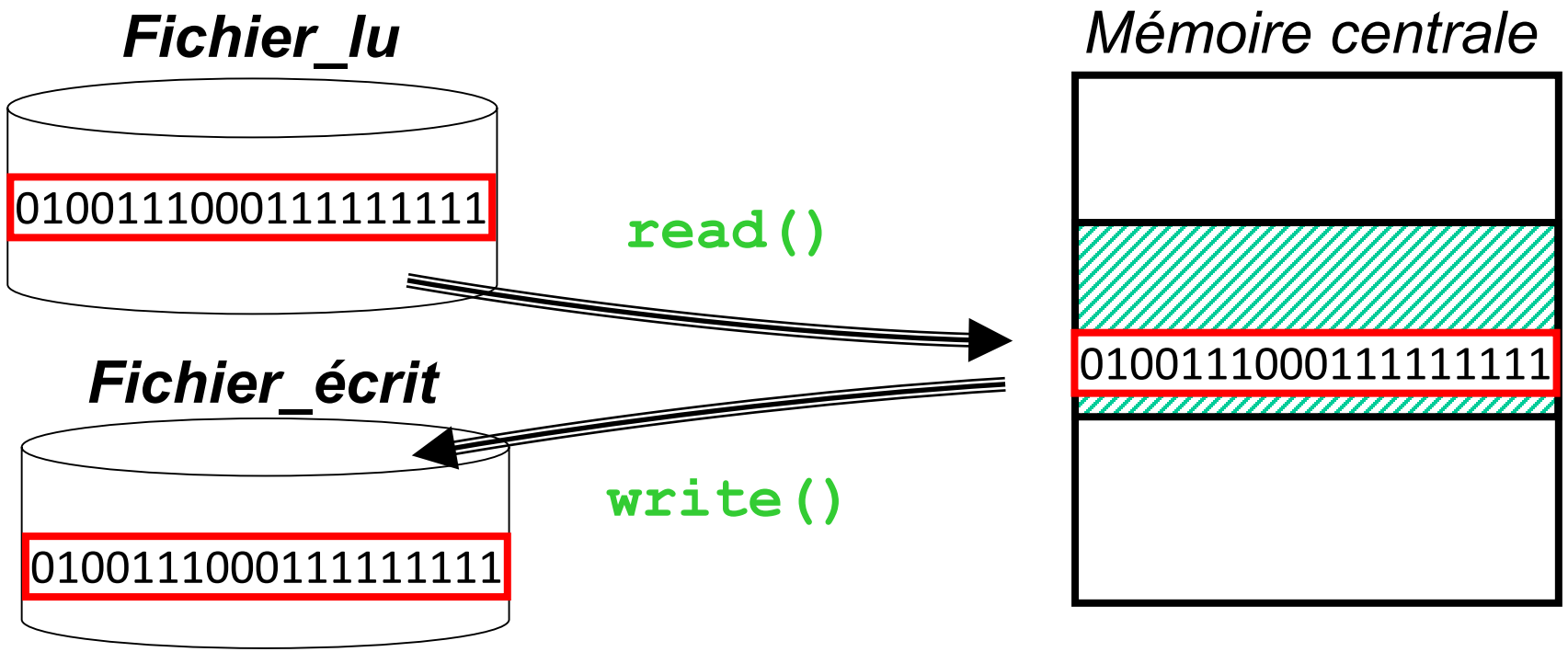
Il faut fermer le fichier avant d'ouvrir un autre fichier avec ce flot F

Une simplification

Un fichier est automatiquement fermé
dès la fin de vie de la variable (locale) du flot qui l'a ouvert

Transfert du motif binaire brut (mode **`binary`**)

Avec le mode **`binary`** un transfert du motif binaire brut est effectué



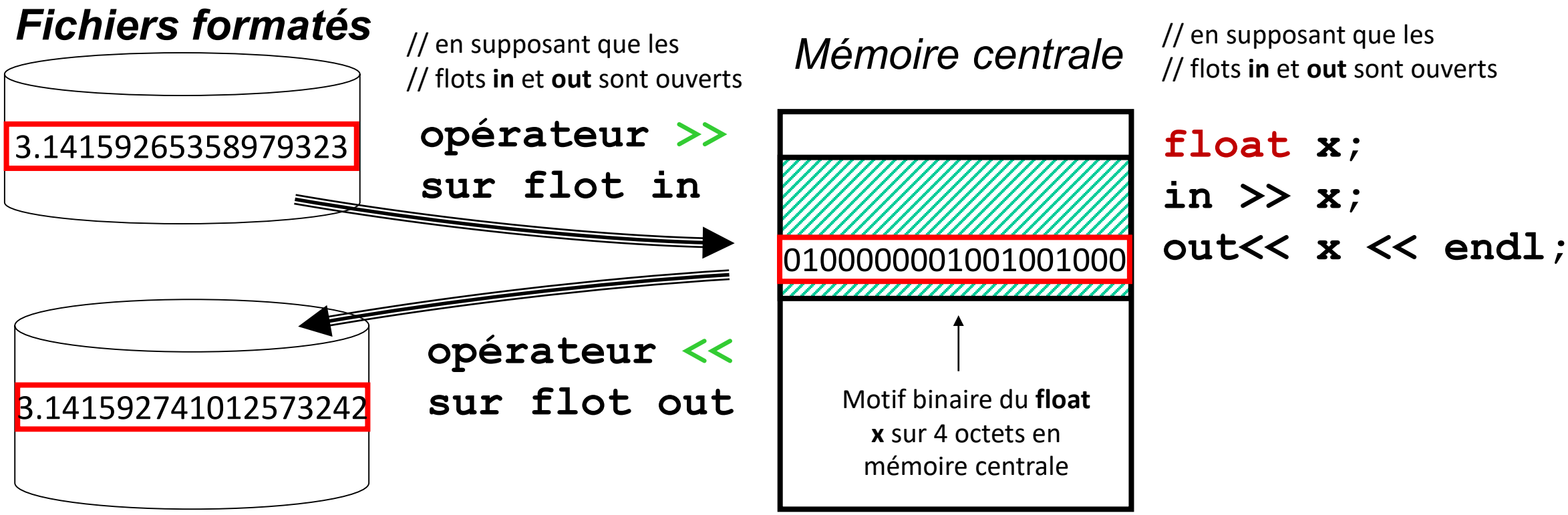
Avantage: Aucune altération des données

Inconvénient: Pas pratique pour éditer un fichier de test

Un tel fichier ne peut pas être consulté ou modifié avec un éditeur de texte

Entrées-Sorties formatées (projet)

Les entrées/sorties formatées sont celles que nous connaissons déjà avec **cin** et **cout**:
les données sont converties en une suite de caractères alphanumériques (e.g. code ASCII)



Avantage: consultation/ modification avec un éditeur de texte

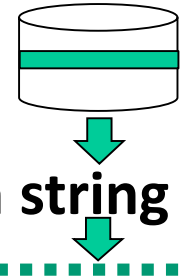
Y a-t-il un inconvénient particulier pour cet exemple ?

Lecture séquentielle d'une ligne entière à la fois avec **getline()** suivi par l'analyse de cette ligne avec un «string stream»

```
#incude <sstream>
```

```
getline(ifstream& in, string& ligne)
```

Renvoie l'équivalent de true si la lecture s'est bien passée et false en cas d'erreur



*L'**input string stream** est utilisable comme le buffer d'entrée du sem1*

Buts:

1) systématiquement extraire une ligne complète à la fois du fichier dans un **string**
2) Initialiser un **input string stream** avec cette ligne lue
3) Lire dans cet **input string stream** *ce qui nous intéresse*
4) *en cas d'erreur*, ignorer la ligne car elle a déjà été extraite du fichier

Précision:

- **getline()** prend tous les caractères jusqu'au premier passage à la ligne inclus
- Mais le caractère du passage à la ligne lui-même n'est pas copié dans la string
- Si la ligne «lue» n'a qu'un seul passage à la ligne, il est extrait et la string est vide

Recommandation:

- Filtrer les séparateurs avec la syntaxe: `getline(in >> ws, ligne)`

E/S formatée: lecture avec getline() et un string stream

```
// lecture & affichage d'une liste d'entiers

string line;
int valeur;

ifstream fichier("nomfichier.txt");
if(fichier.fail()) exit(EXIT_FAILURE);

while(getline(fichier >> ws, line))
{
     istringstream data(line);
     while(data >> valeur)
          cout  << valeur << endl;
}

fichier.close();
```

exécution

Exemple de fichier texte pouvant être traité avec ce programme

```
33
1  44 -2

100


99 2
```

Affichage

```
33
1
44
-2
100
99
2
```

Fichier de configuration d'une application

Un **fichier de configuration** permet d'initialiser une application complexe. Il doit être organisé selon un format sans ambiguïtés et contenir toutes les données pour restituer l'état désiré du programme:

- configuration initiale pour un scénario de test
- sauvegarde d'une configuration intermédiaire du programme

Pseudo-code de lecture d'un fichier de configuration

avec gestion d'une variable `etat` indiquant le format de décodage

```
etat = DEBUT_LECTURE
Tant que (fin du fichier pas atteinte)
     lire une ligne entière avec getline()
     décoder la ligne lue avec un string stream
              selon l'etat courant
     mise à jour éventuelle de etat
```

Exemple : format de fichier de configuration (développé en série0)

Un programme doit tenir à jour des listes de LIVREUR, de VEHICULE et de LIVRAISON. Le programme fait une sauvegarde régulière de son état sous forme d'un fichier formaté selon la structure ci-dessous.

voici le format choisi pour un tel fichier:

```
# ceci est un commentaire qui commence en début de ligne

Nombre de LIVREURs
Nom_livreur  disponibilité(1/0)
...
Nombre de VEHICULEs
Numéro_vehicule disponibilité(1/0)
...
Nombre de LIVRAISONs
Nom_livreur Numéro_vehicule
...
```

Un LIVREUR par ligne

Un VEHICULE par ligne

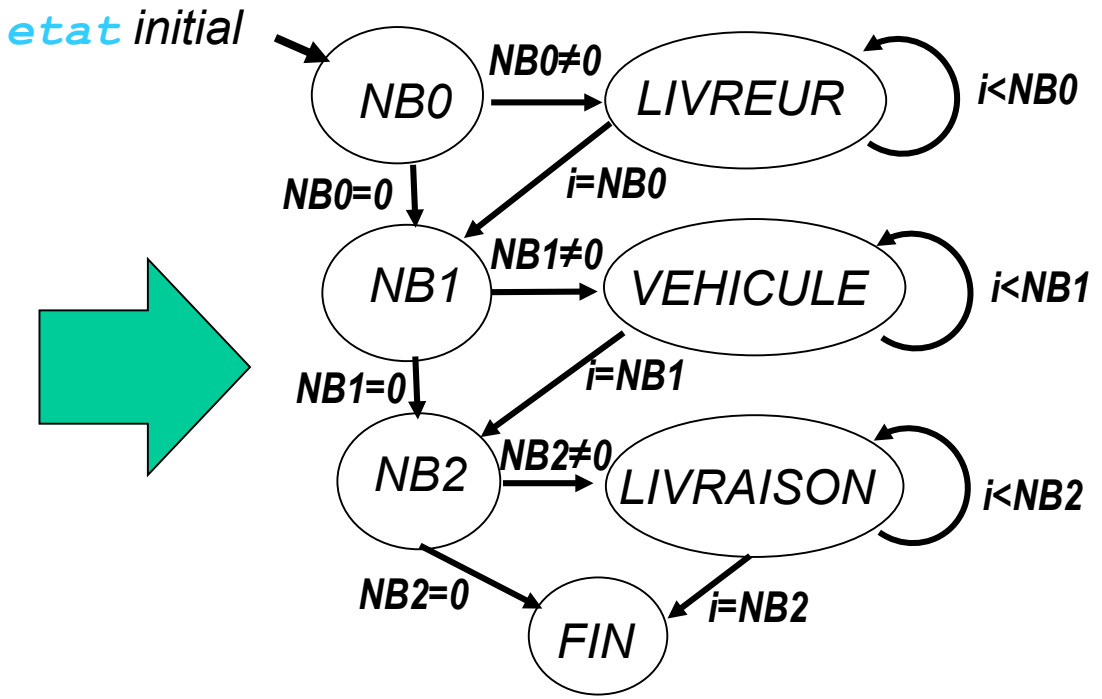
Une LIVRAISON par ligne

Exemple: méthode avec **automate de lecture (Série0)**

Format de fichier

```
# ceci est un commentaire
Nombre de LIVREURs
Nom_livreur  disponibilité(1/0)
...
Nombre de VEHICULEs
Numéro_vehicule disponibilité(1/0)
...
Nombre de LIVRAISONs
Nom_livreur Numéro_vehicule
...
```

Convertir le format de fichier en un **graphe** faisant apparaitre des **états** distincts et les **transitions** entre états



Chaque `etat` de **l'automate** correspond à un format différent.
Les **transitions** sont les conditions logiques à remplir pour changer d'`etat`

`etat` initial

`etat`	type des données lues
`NB0`	`unsigned`
`LIVREUR`	`string bool`
`NB1`	`unsigned`
`VEHICULE`	`unsigned bool`
`etc...`	…

Exemple: lecture du fichier

Série_0: ce code y est analysé en détail et ré-organisé avec plusieurs fonctions (lisibilité)

```
1   enum Etat_lecture {NB0,LIVREUR,NB1,VEHICULE,NB2,LIVRAISON,FIN};
2   int etat(NB0),total(0), count(0);
3   bool disponible(false);
4   string line, name;
5   ifstream fichier("nomfichier.txt");
6   if(!fichier.fail())
7   {
8     while(getline(fichier >> ws,line))
9     {
10      if(line[0]=='#')  continue;
11      istringstream data(line);
12      switch(etat)
13      {
14      case NB0: data >> total; count=0 ;
15         if(total==0) etat=NB1; else etat=LIVREUR ; break;
16      case LIVREUR: data >> name >> disponible; ++count;
17         if(count == total) etat=NB1 ;
18         cout << "Livreur " << count+1 << ": " << name << endl; break;
19      case NB1: data >> total; count=0 ;
20         if(total==0) etat=NB2; else etat=VEHICULE ; break;
... etc ...
```

Exemple: écriture du fichier en délégant à des modules leur part d'écriture

```
1  int total, i;
2   ofstream fichier("nomfichier.txt");
3  if(!fichier.fail())
4  {
5      livreur_save(fichier);
6       vehicule_save(fichier);
7      livraison_save(fichier);
8   }
9   fichier.close();
```

Ensuite chaque module dispose d'un accès facile aux données qu'il gère

```
1  void livreur_save(ofstream& fichier)
2  {
3       fichier << livreur_db.size() << endl;
4       for(const auto& livreur : livreur_db)
5       {
6            fichier << livreur.getNom()
7                    << livreur.getDisponible() << endl;
8       }
9  }
```

*Hypothèse: le module **livreur.cc** mémorise toutes les instances de la classe **Livreur** dans un static vector **livreur_db***

Résumé

- **toutes les Entrées-Sorties sont traitées avec des flots.**
- **les sorties formatées ont l'avantage de pouvoir être éditées avec un éditeur de texte MAIS le formatage peut introduire une perte de précision pour les nombres à virgule flottante.**
- **getline() et l'usage d'un string stream permettent de lire séquentiellement un fichier, du début à la fin, une ligne à la fois.**
- **nous exploitons les E-S formatées pour travailler avec des fichiers de configuration pour le projet.**