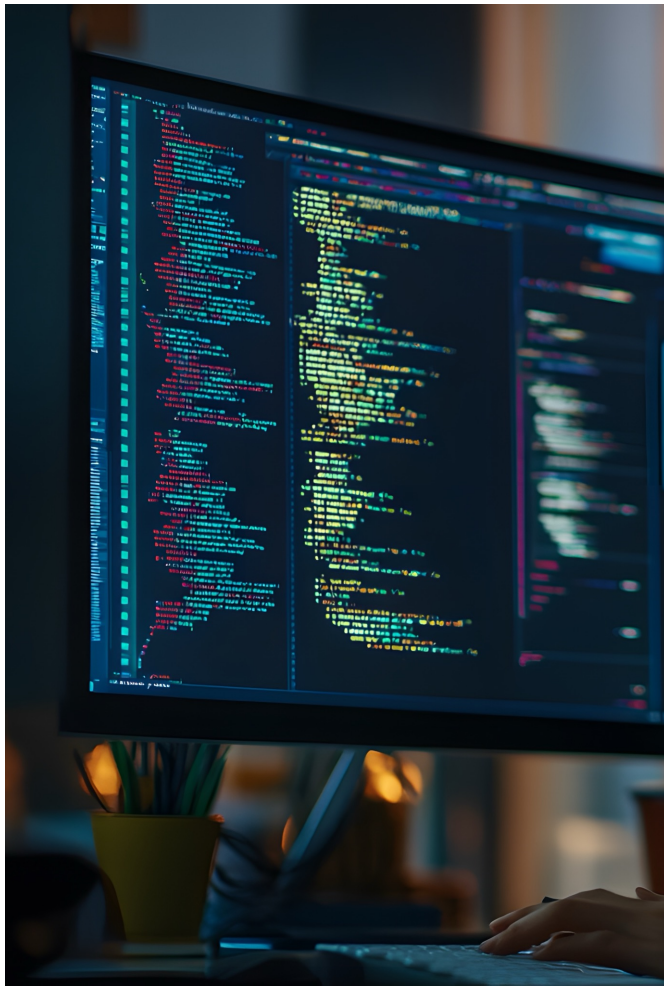
Programmation Orientée Projet
COM-112(a)

Semaine 3

Rafael Pires
rafael.pires@epfl.ch

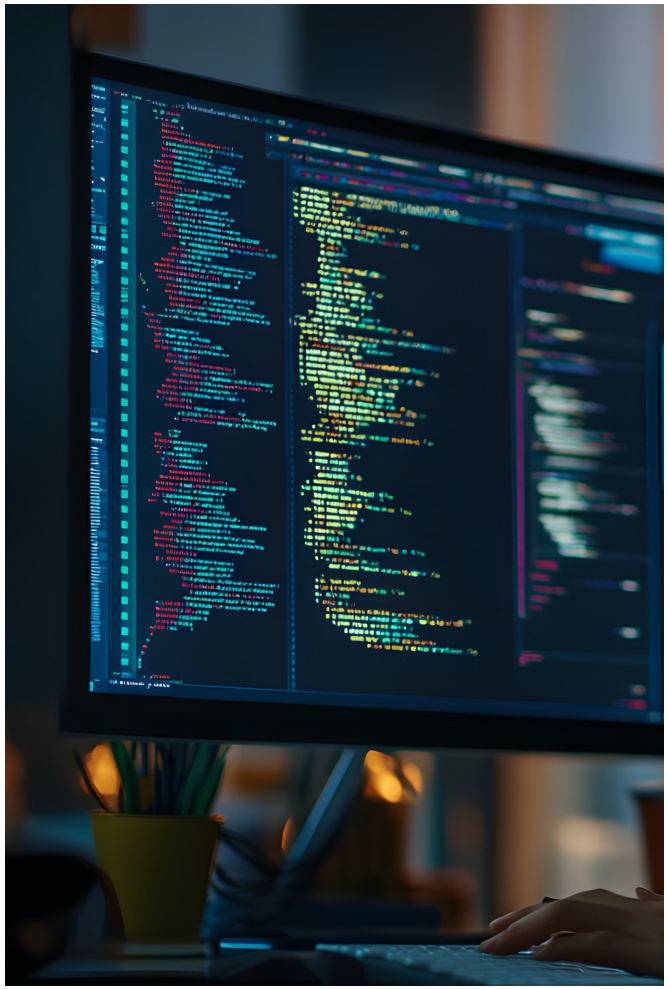
Lausanne, 06.03.2026

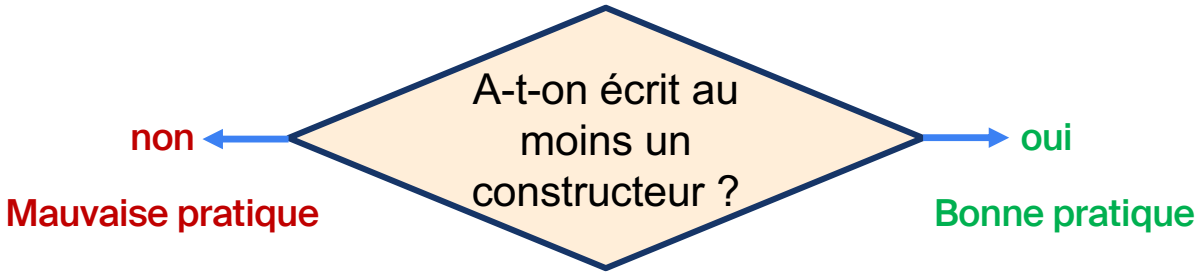
Constructeurs et destructeurs



- **Buts :**
  - Gérer l'initialisation des instances
  - Quand faut-il s'intéresser au constructeur de copie et au destructeur ?
- **Plan :**
  - Les bonnes pratiques en matière de constructeur
  - Exemple pour lequel il faut un constructeur de copie et un destructeur

Constructeurs et destructeurs

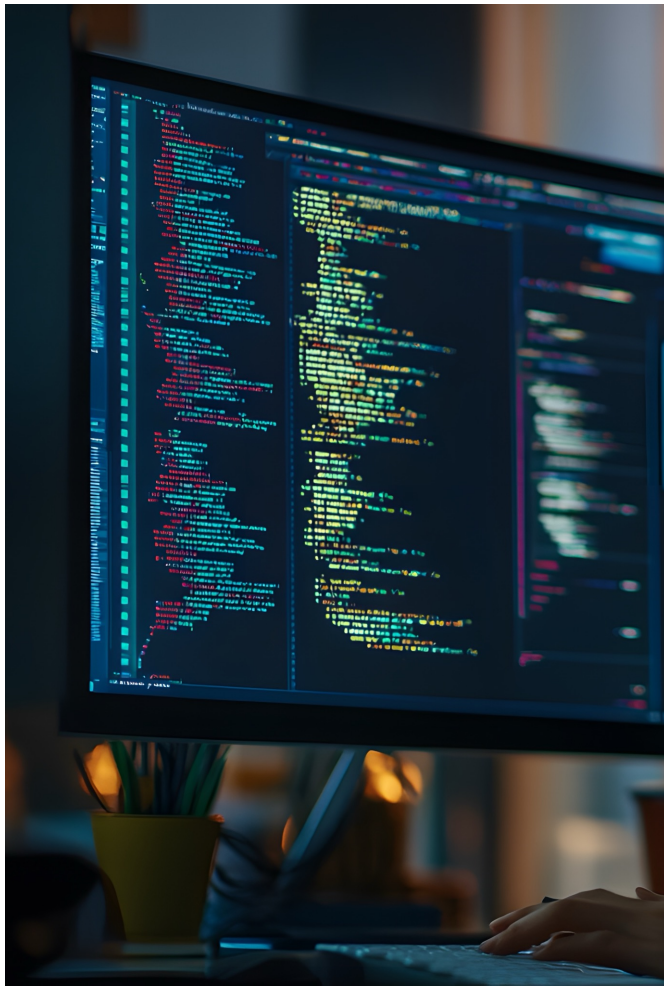




- Le compilateur C++ met en place un constructeur par défaut par défaut
- Un attribut défini par une classe est initialisé avec son constructeur par défaut (p. ex. string est initialisé avec la chaîne vide).
- Un attribut ayant un type de base (`int`, `char`, `bool`, `double`) obtient un motif binaire quelconque : source de bugs.

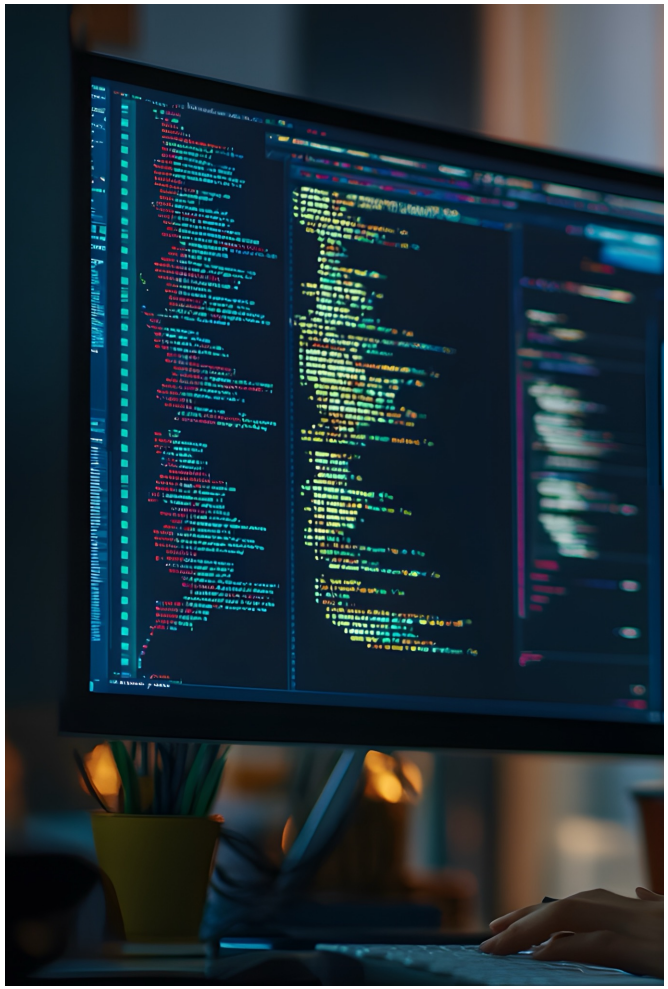
- Le constructeur par défaut est utilisé pour une déclaration sans aucune transmission de valeur d'initialisation.
- La surcharge permet de définir autant de constructeurs que nécessaire.
- Un constructeur peut en appeler un autre dans la liste d'initialisation (section deux points après les paramètres et avant le bloc).

Constructeurs et destructeurs



- Allocation dynamique de mémoire
- Ressources partagées (fichiers, périphériques etc.)

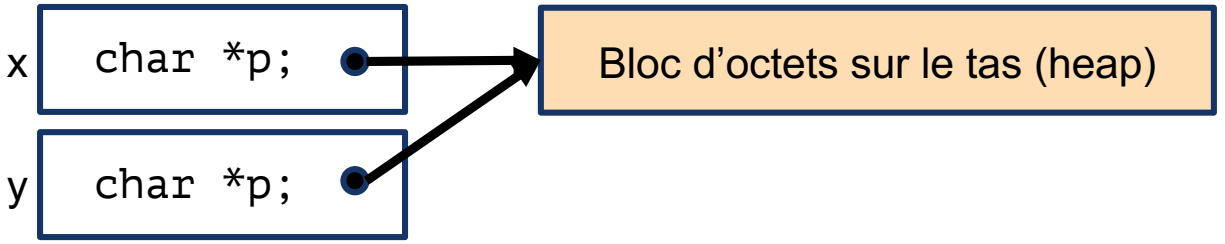
Constructeur de copie



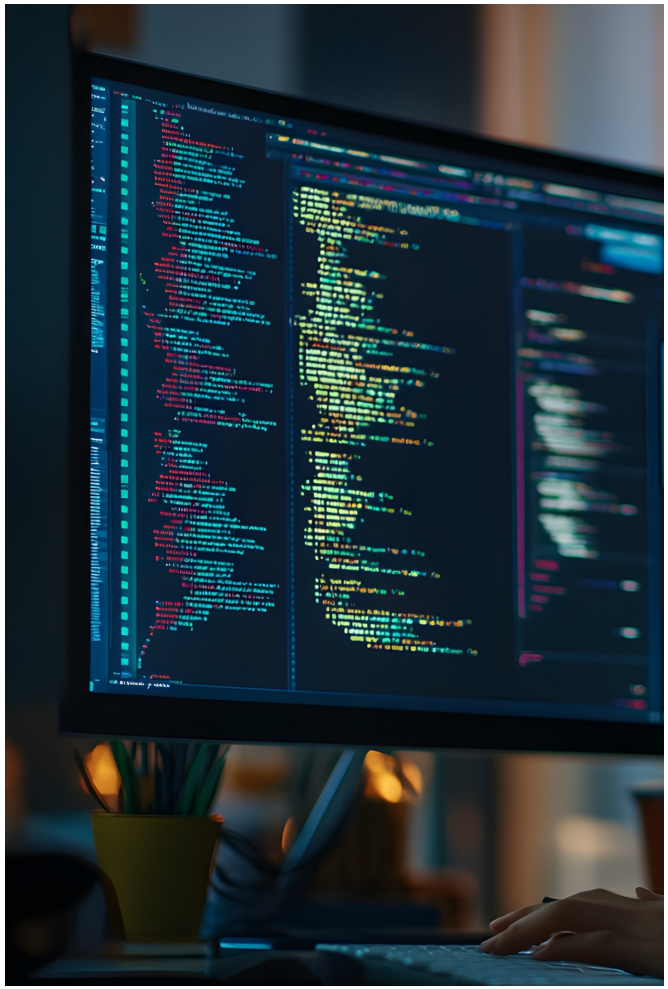
- Constructeur de copie **par défaut**
  - Copie **superficielle** : copie la valeur des attributs terme à terme

```
Class_avec_alloc_dyn x(50); // attribut pointeur
                            // vers bloc de 50 octets
```


```
Class_avec_alloc_dyn y(x); // copie superficielle
```



Le destructeur (il n'y en a qu'un par classe)



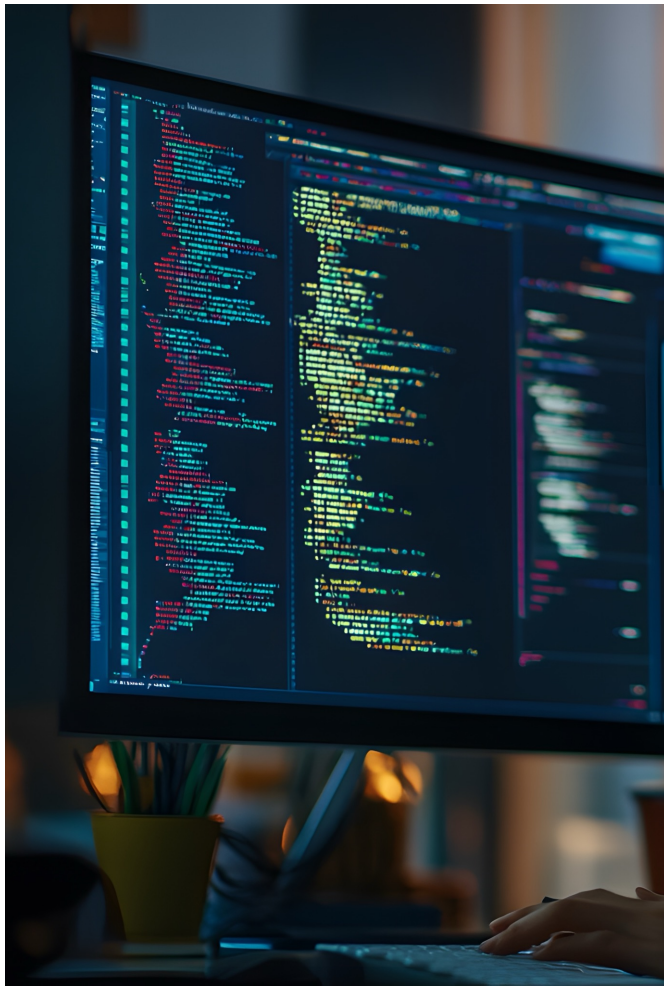
- En général, pas nécessaire
- Le **destructeur par défaut** ne gère pas la mémoire allouée dynamiquement

```
{ Class_avec_alloc_dyn x(50); // attribut pointeur
                              // vers bloc de 50 octets
```


```
} // fin de portée
```

Plus aucun accès au bloc : **fuite de mémoire**

Constructeurs et destructeurs : Règle à mémoriser



- Si on modifie l'une des trois méthodes ci-dessous, **il faut** vérifier si les deux autres doivent être aussi adaptées

  - Constructeur de copie
  - Destructeur
  - Opérateur d'affectation (leçon ultérieure)

rect.h

```
class Rect
{
public:
    Rect(double width, double height);
    double surf() const;
private:
     double width;
     double height;
};
```

rect.cc

```
#include <iostream>
#include "rect.h"
using namespace std;

Rect::Rect(double width, double height)
          :width(width),height(height) {}

double Rect::surf() const{
          return width*height;
}
```

prog.cc

```
#include <iostream>
#include "rect.h"
using namespace std;

int main()
{
      Rect r;
      cout << r.surf() << endl;
      return 0;
}
```

Quiz1 : ce code

A: s'execute et affiche une valeur quelconque à cause du constructeur par défaut par défaut

B: s'execute et affiche 0.

C: ne compile pas à cause de la liste d'initialisation

D: s'execute et affiche une valeur quelconque car les paramètres masquent les attributs

E: ne compile pas car pas de constructeur par défaut

rect.h

```
class Rect
{
public:
    Rect(double width =0.,
         double height=0.);
    double surf() const;
private:
     double width  =1.;
     double height =1.;
};
```

rect.cc

```
#include <iostream>
#include "rect.h"
using namespace std;

Rect::Rect(double width, double
height)
          :width(width),height(height)
{}

double Rect::surf() const{
          return width*height;
}
```

prog.cc

```
#include <iostream>
#include "rect.h"
using namespace std;

int main()
{
     Rect r;
     cout << r.surf() << endl;
     return 0;
}
```

Quiz2 : ce code

A: ne compile pas car pas de constructeur par défaut

B: s'execute et affiche 0.

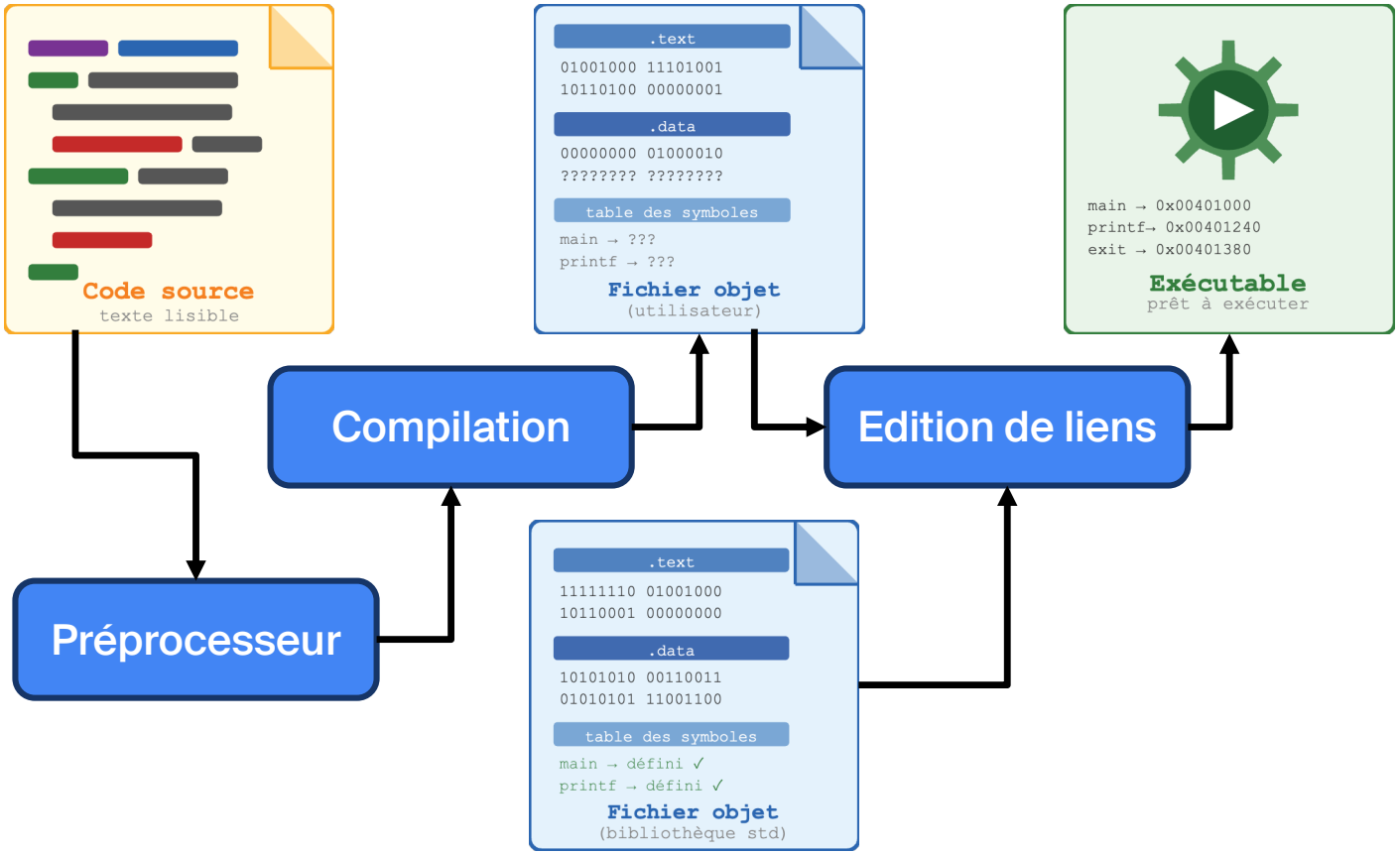
C: s'execute et affiche 1.

D: aucune des autres réponses

Le préprocesseur

- Objectifs :
  - Gérer les constantes en programmation modulaire
  - Découvrir quelques outils de mise au point
- Plan :
  - Où intervient le préprocesseur ?
  - `include` et fichier en-tête
  - Les constantes et l'usage de `define`
  - Compilation conditionnelle
  - Traiter le problème de l'inclusion multiple

Production d’un exécutable



Le préprocesseur : Vue d'ensemble

- Un petit nombre de # directives est traduit par le préprocesseur qui produit le code source expansé (seulement instructions en C++)
- Intérêts :
  - Intègre l'interface des modules avec `#include`
  - Empêcher les doubles définitions avec `#ifndef`, `#endif` et `#define`
  - Gestion de versions avec `#ifdef`
  - Outil de mise au point. Ex: `NDEBUG`

Le préprocesseur : `#include`

- Standard C++ (ou avec option de compilation `-I` suivi d'un de répertoire)

```
#include <iostream>
```

- Interface d'un module : par défaut, le fichier en-tête doit se trouver dans le même répertoire que le code source

```
#include "mon_fichier.h"
```

Le préprocesseur : `#define`

- Les deux conséquences d'une directive `#define`
  - Elle **toujours** crée un symbole, encore appelée une « variable du préprocesseur ». D'autres directives permettent de tester si un tel symbole existe.
  - **Optionnellement** elle associe un texte au symbole. Ce texte remplacera le symbole partout.

```
#define VITESSE_ROBOT 2.5

#define SYMBOLE_SANS_TEXTE
```

- Les directives `#define` ont beaucoup été utilisées pour remplacer les magic number.
  Actuellement on utilise plutôt `constexpr` et `enum`.

Le préprocesseur : `#define`

- Un symbole peut être créé sur la ligne de compilation

```
g++ -D NDEBUG -std=c++11 -c projet.cc
```

- Est équivalent à écrire la directive ci-dessous dans `projet.cc`

```
#define NDEBUG
```

- Conséquences :
  - Le symbole `NDEBUG` existe
  - D'autres directives peuvent tester son existence (compilation conditionnelle)

Le préprocesseur et la compilation conditionnelle

- Plusieurs directives peuvent inclure/exclure des morceaux de code à l'étape de la compilation si un symbole est défini ou pas.

- Intérêt : centraliser plusieurs versions d'un code source dans un même fichier, tout en ne compilant que le code correspondant à la version ciblée. Exemples de cas d'usage :
  - versions mise au point / rendu / commerciale
  - versions dépendant de la plateforme cible
  - versions dépendant d'options marketing

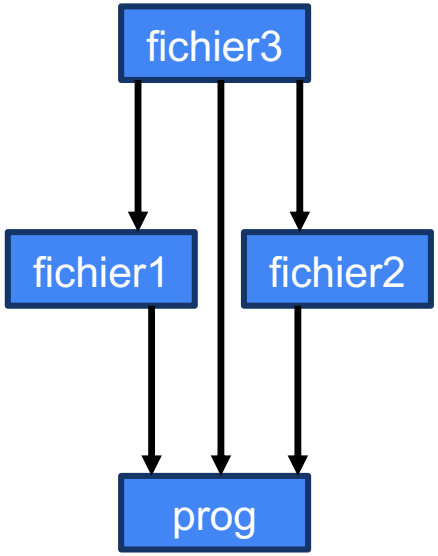
Le préprocesseur : #ifdef et #endif

- Le code source entre #ifdef et #endif est inclus dans le fichier source expansé seulement si le symbole qui suit #ifdef existe.
- Exemple :

```
#define MYDEBUG
…
#ifdef MYDEBUG
    // ici du code supplémentaires, par ex
    // pour l'affichage de messages non
    // destinés au rendu officiel
#endif
```

Architecture modulaire et pathologie de l'inclusion multiple

- Un projet organisé avec de nombreux modules peut conduire à des cas d'inclusions multiples de certains fichiers en-tête, donc à des **définitions multiples**.
- **Problème**: la définition multiple d'un symbole ou d'un nom de type ou de fonction conduit à une erreur syntaxique.
- Solution: systématiquement mettre en place un **header guard** pour chaque fichier en-tête de façon au n'autoriser qu'une seule inclusion de son contenu dans le code source à compiler.



Header guard

- Le code entre `#ifndef` et `#endif` est inclus dans le code source expansé seulement si le symbole qui suit `#ifndef` n'existe pas.
- Exemple : l'interface `nom.h` doit être structurée comme suit:

```
#ifndef NOM_H
#define NOM_H
   // ici le contenu habituel de nom.h
   // avec les prototypes des fonctions,
   // etc...
#endif
```

Danger de collision de nom !
Le symbole utilisé comme « garde » (ici c'est `NOM_H`) doit être unique sur l'ensemble du programme

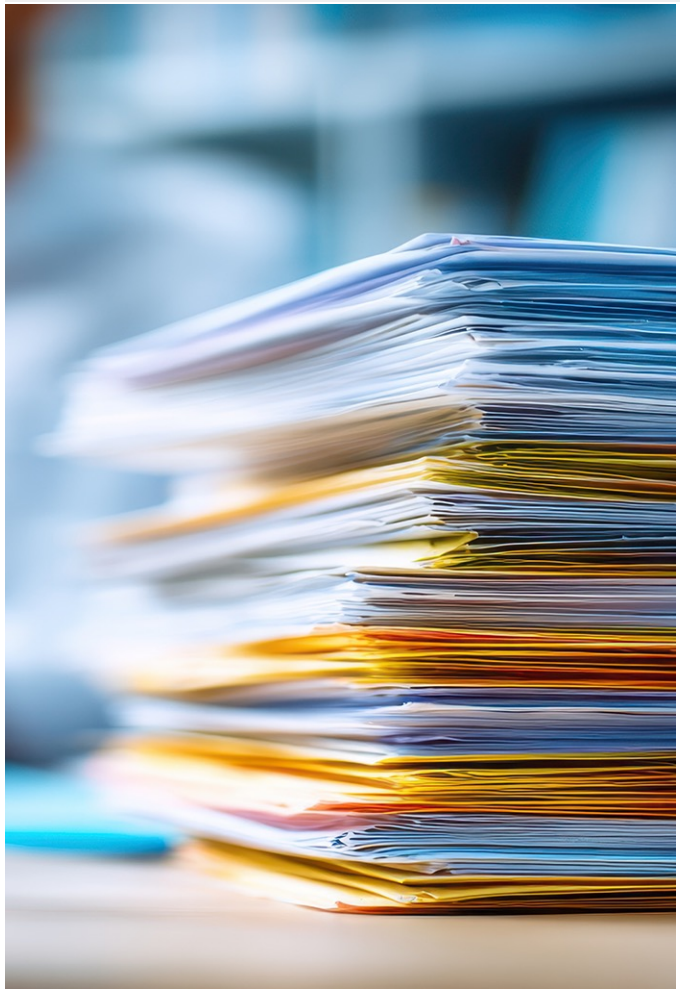
Outils de mise au point : `assert()`, NDEBUG et autres symboles

- Bonne pratique: une **assertion** est une condition qui doit toujours être vraie pour le bon fonctionnement du programme. Elle est mise en œuvre avec `assert(condition);`
- Si elle est fausse, l'exécution s'arrête en indiquant la condition, le fichier et la ligne du code source.
- A savoir : le symbole `NDEBUG` veut dire « no debug » ; s'il est défini avant `#include <cassert>` alors le préprocesseur désactive les asserts (pour la version finale sans bug)

Autres symboles pour documenter la mise au point :

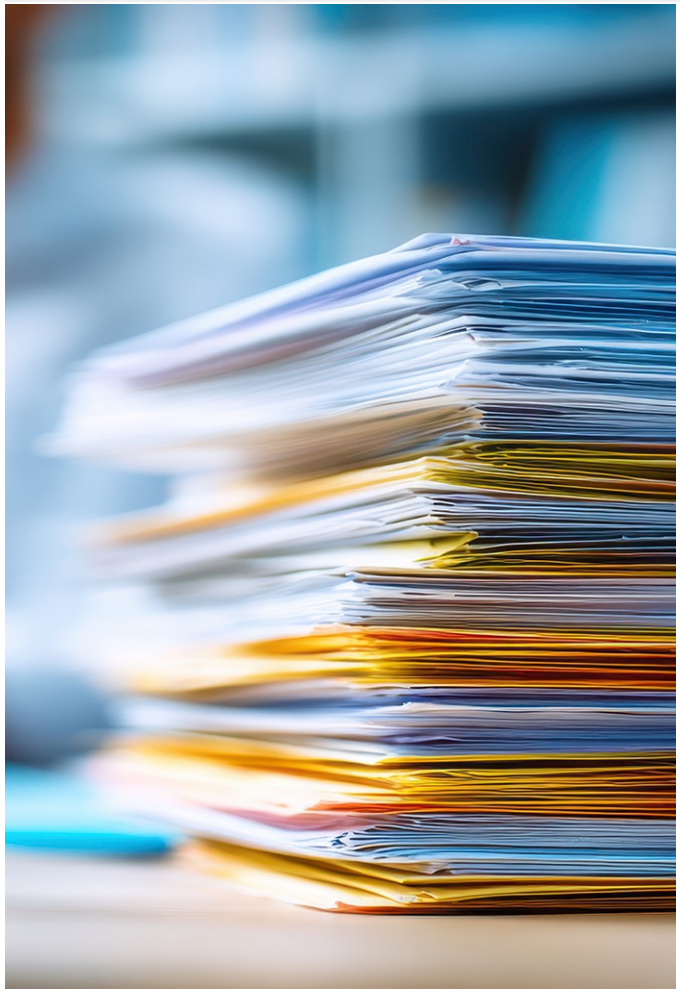
`__func__`	nom de la fonction où se trouve le symbole
`__FILE__`	nom du fichier source où se trouve le symbole
`__LINE__`	numéro de ligne où se trouve le symbole
`__TIME__`	heure de la dernière compilation du fichier source
`__DATE__`	date de la dernière compilation du fichier source

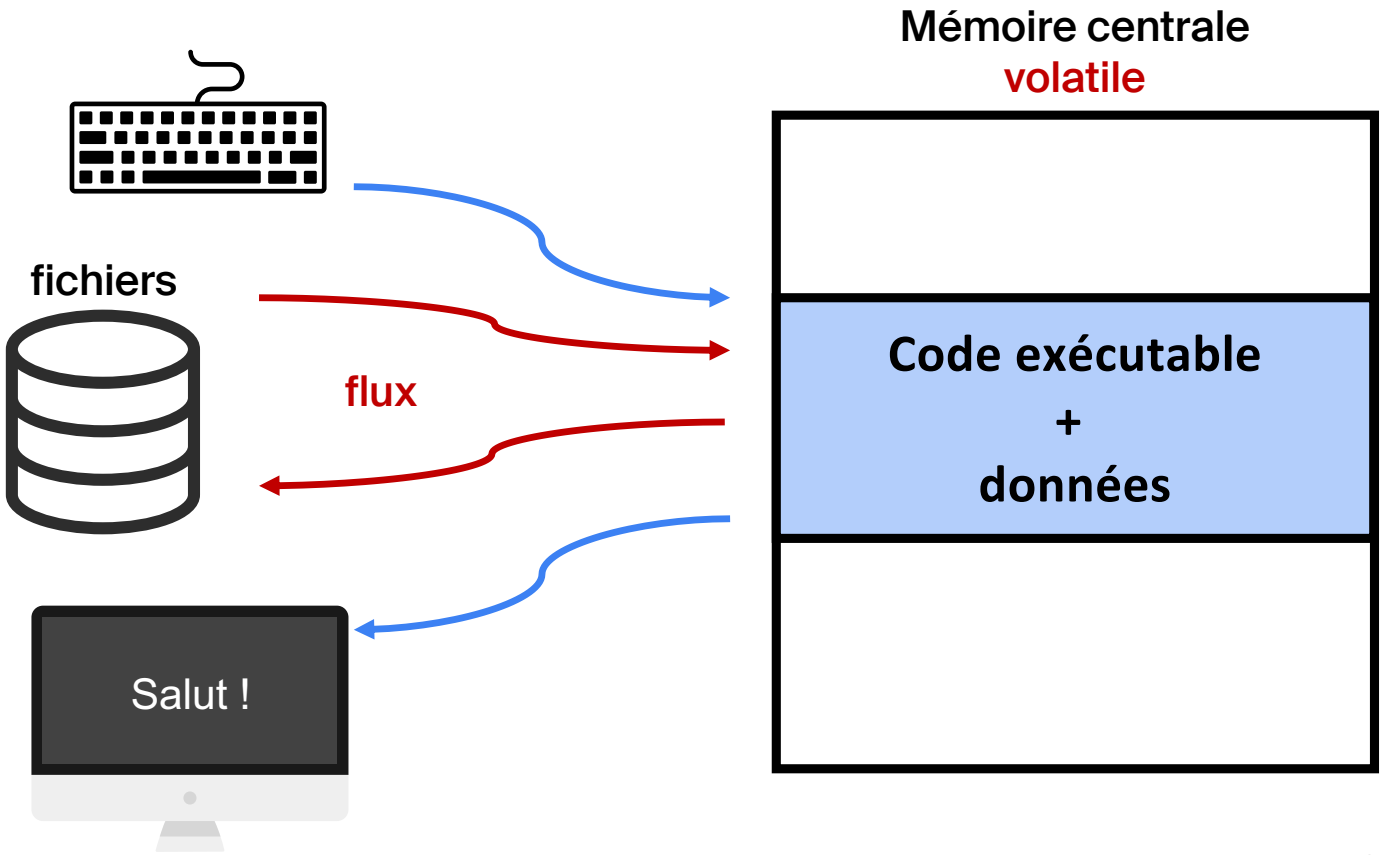
Les entrées-sorties sur fichiers



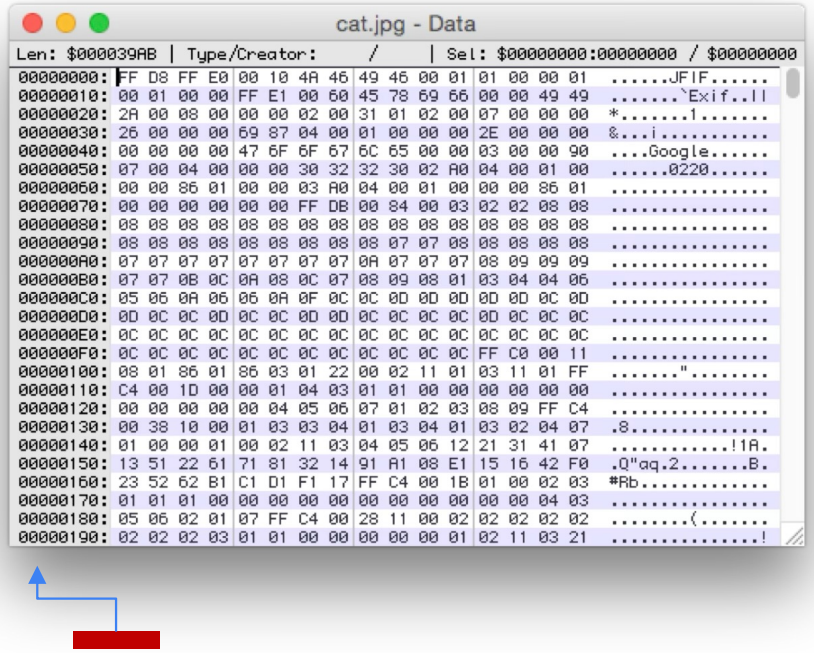
- Objectifs
  - Manipuler les fichiers en lecture/écriture
  - Introduire le concept d'automate pour la lecture
- Plan :
  - Mémoire volatile et permanente
  - Différence entre binaire et format
  - Automate de lecture

Les entrées-sorties sur fichiers



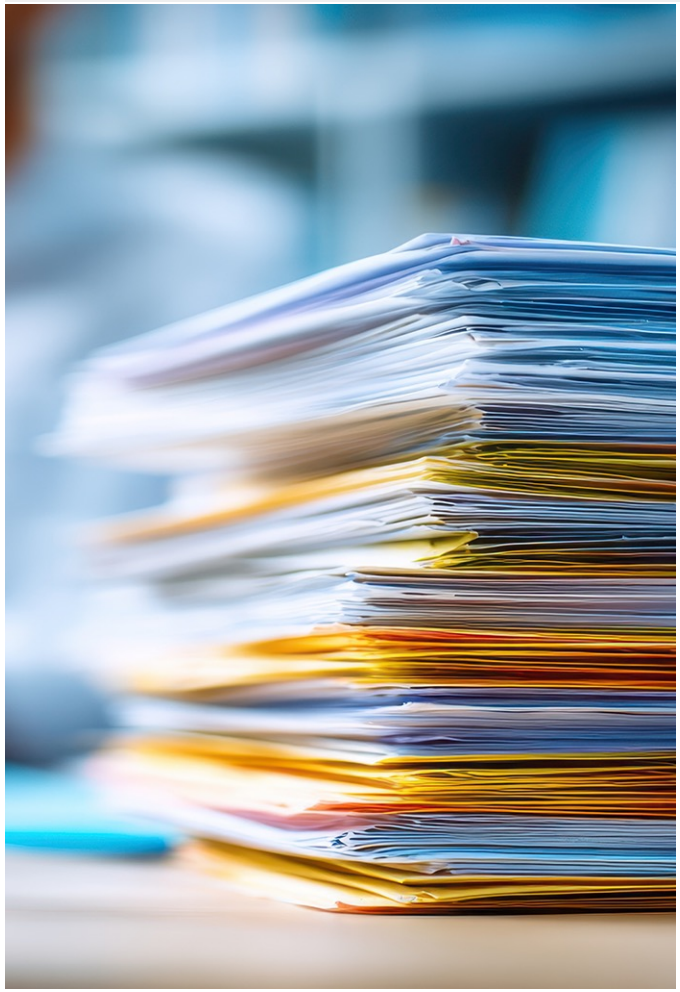


Fichiers mode binaire



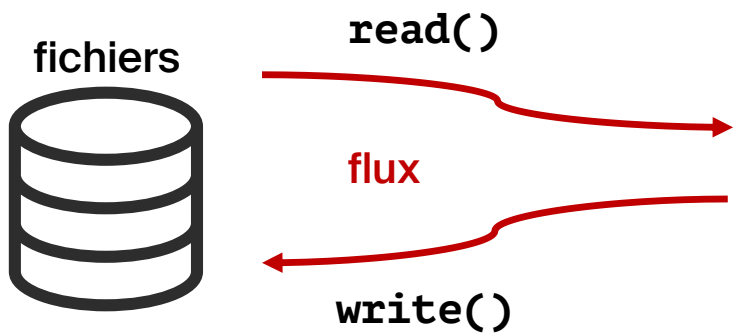
Un fichier est une séquence de bytes qui, interprétés correctement, ont du sens.

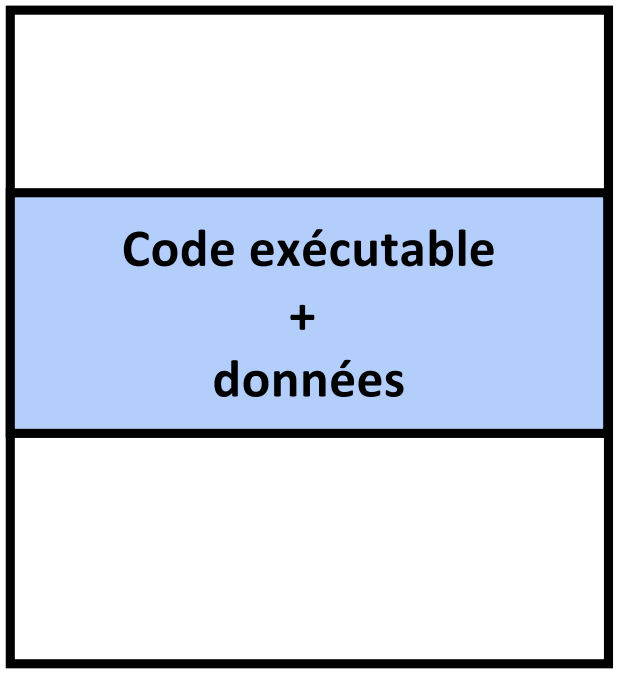
Transfert du motif binaire brut (mode `binary`)



Avantage : Aucune altération des données
Inconvénient : Pas pratique pour éditer

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






```
std::ifstream file("mon_fichier.bin", std::ios::binary);
```

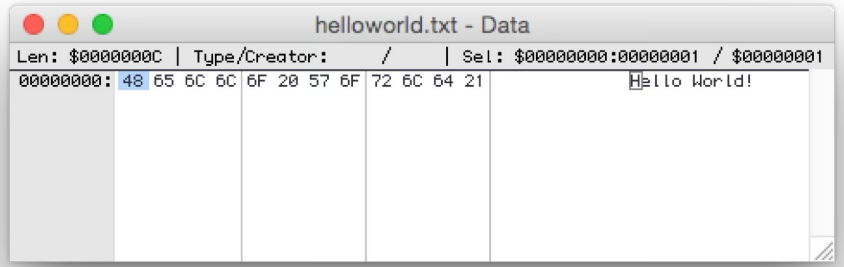
Fichiers mode texte

- American Standard Code for Information Interchange (ASCII, 1963)
- Unicode
- UTF-8

1	Start of header
...	
9	Tabulator (“\t”)
10	Line feed (“\n”)
...	
32	Space
...	
48	“0”
...	...
57	“9”

66	“B”
67	“C”
...	...
90	“Z”
...	
97	“a”
...	...
122	“z”
...	
127	Delete

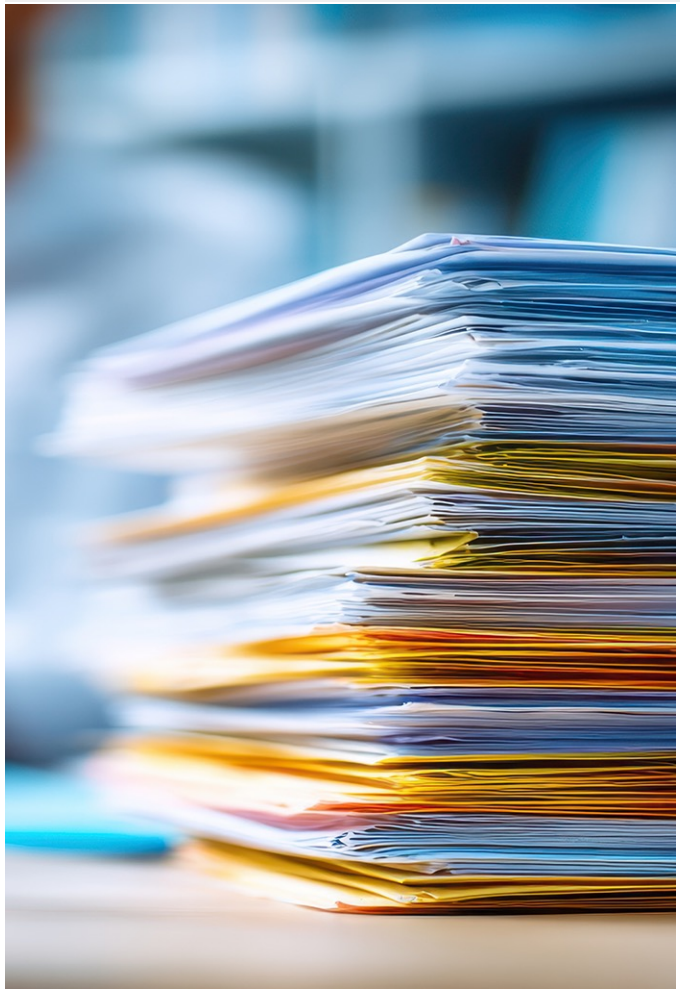
Fichiers mode texte



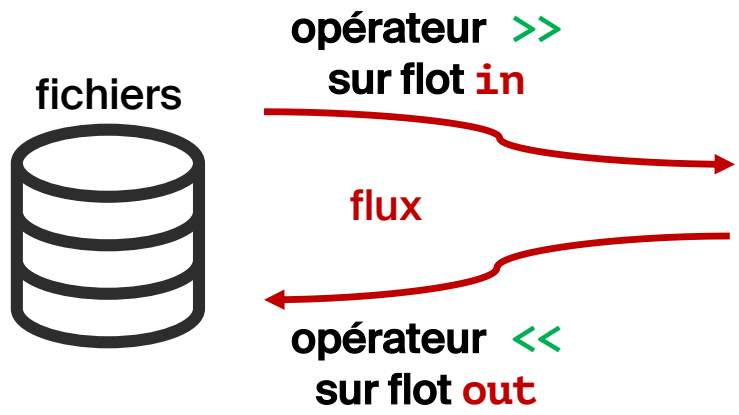
0	1	2	3	4	5	6	7	8	9	10	11
01001000	01100101	01101100	01101100	01101111	00100000	01010111	1101111	01110010	01101100	01100100	00100001
48	65	6C	6C	6F	20	57	6F	72	6C	64	21
H	**e**	**l**	**l**	**o**		**W**	**o**	**r**	**l**	**d**	**!**

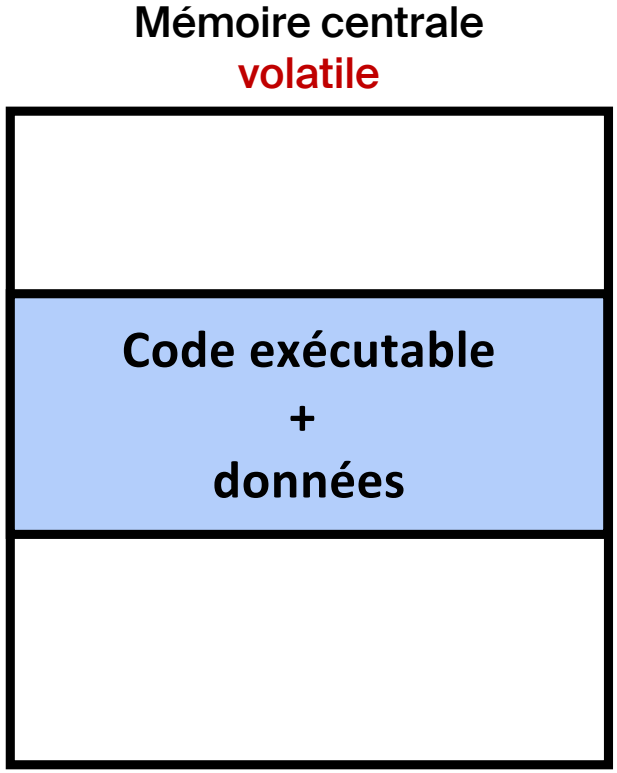
- Indice de l'octet
- binaire
- hexadécimal
- Interprétation texte

Entrées-sorties formatées (projet)



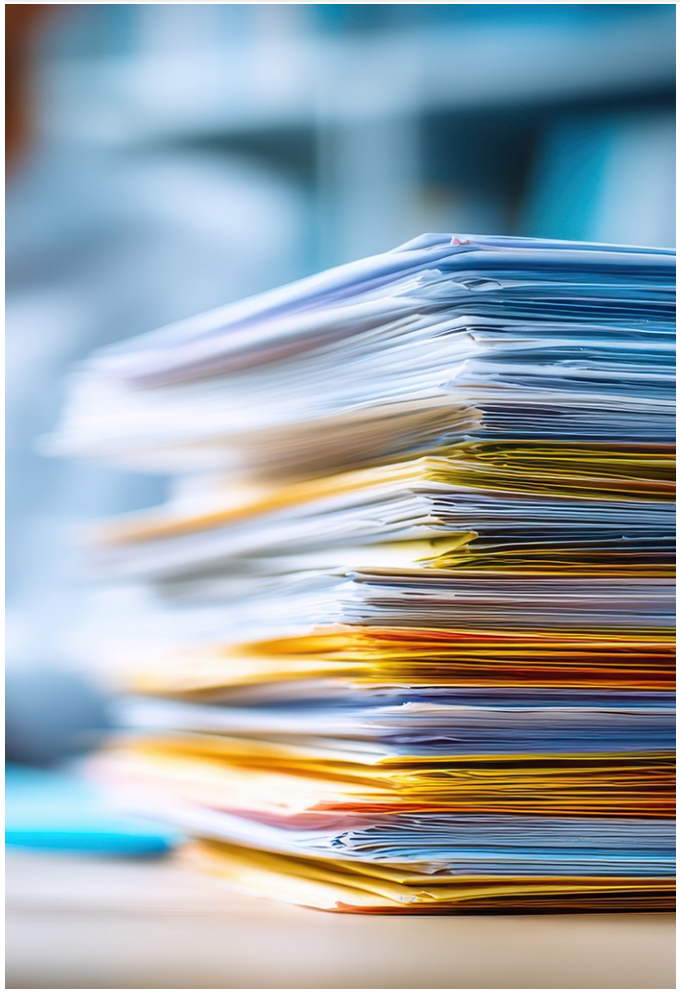
Les entrées/sorties formatées sont celles que nous connaissons déjà avec `cin` et `cout`.
Les données sont converties en une suite de caractères alphanumériques.





Avantage : Consultation/modification avec un éditeur de texte.
Inconvénient : Perte de précision potentielle

Lecture d'un fichier en accès séquentiel



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

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

Étapes :

- **Extraire une ligne complète dans un `string`**

```
string line;
getline(fichier, line);
```

- **Initialiser un input string stream `istringstream` avec cette ligne lue**

```
istringstream data(line);
```

- **Lire depuis cet `istringstream` ce qui nous intéresse**

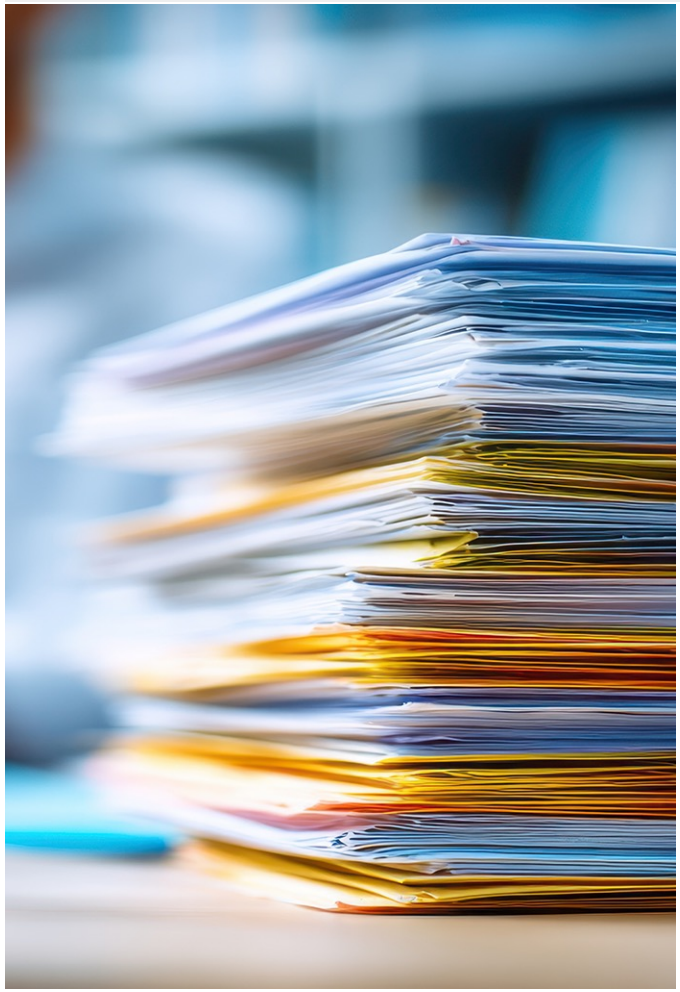
```
int valeur;
data >> valeur;
```

Recommandation :

- **Filtrer les séparateurs avec la syntaxe :**

```
getline(fichier >> ws, ligne)
```

Lecture d'un fichier en accès séquentiel



1. Ouverture du fichier

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

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

2. Une ou plusieurs opération(s) de lecture

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

Exemple de fichier texte pouvant être traité avec ce programme

```
33
1  44
   -2

100


99 2
```

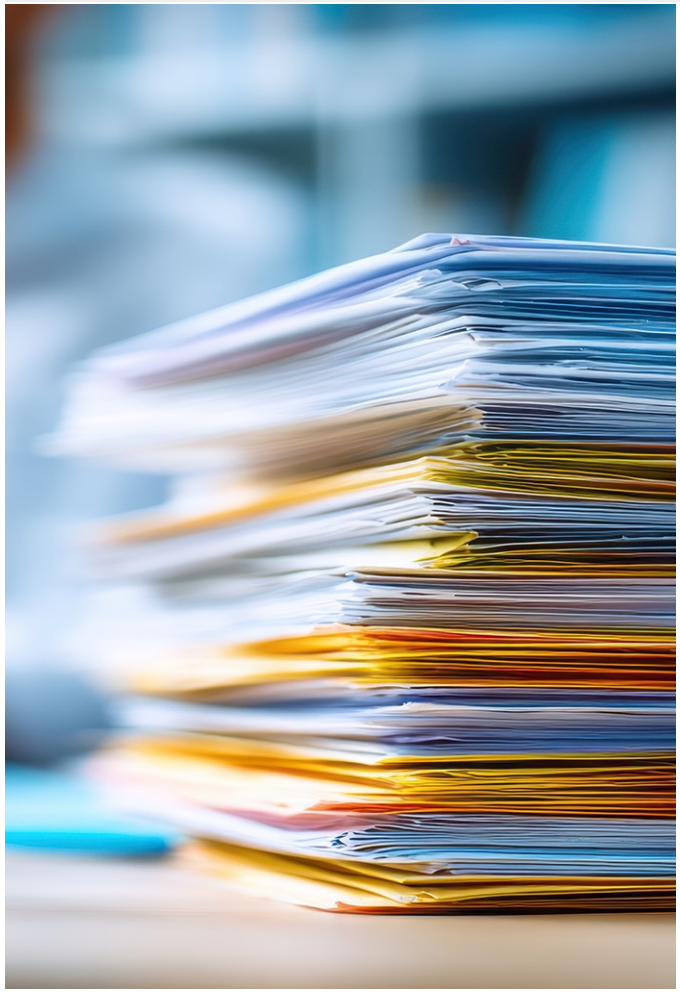
Affichage

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

3. Fermeture du fichier

```
fichier.close();
```

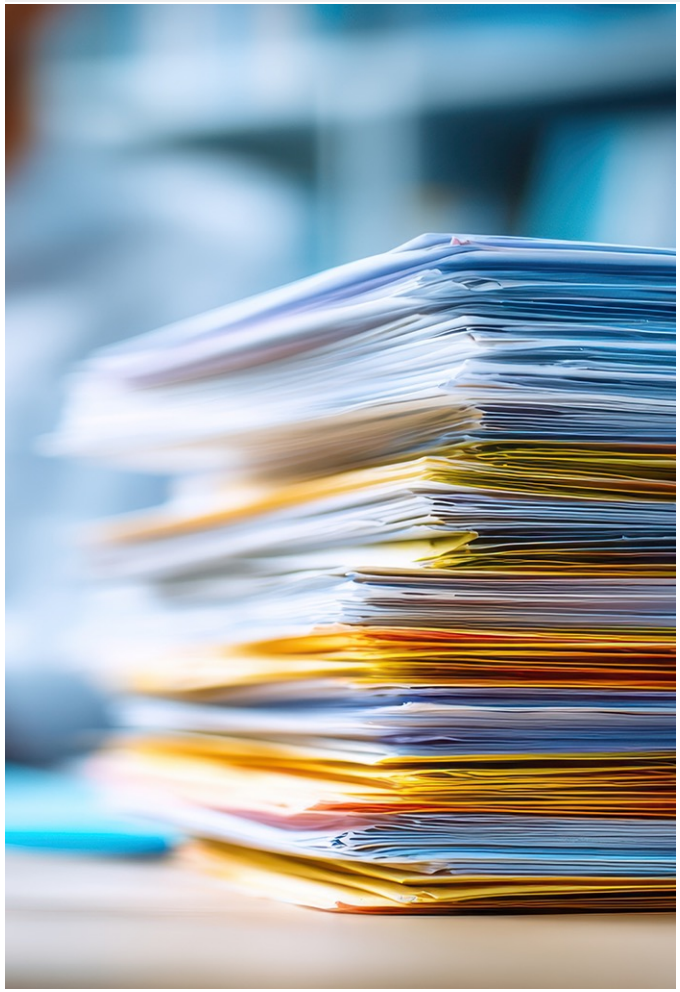
Exemple : fichier de configuration



- 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.

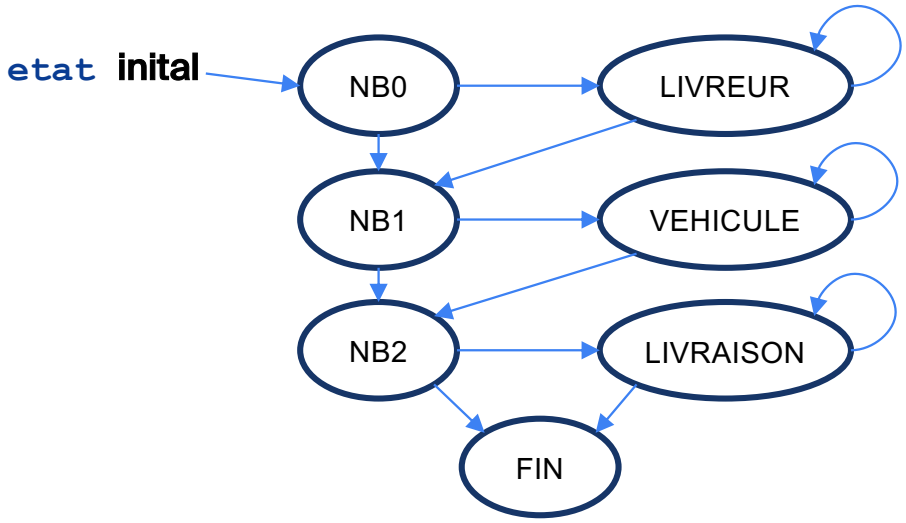
```
Nombre de LIVREURs (NB0)
Nom_livreur  disponibilité(1/0)
...
Nombre de VEHICULEs (NB1)
Numéro_vehicule disponibilité(1/0)
...
Nombre de LIVRAISONs (NB2)
Nom_livreur Numéro_vehicule
...
```

Exemple : fichier de configuration (automate de lecture)



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 istringstream
                  selon l'etat courant
          mise à jour éventuelle de etat
```



Résumé Cours 3

1. Constructeurs et destructeurs
   - Faire attention quand on effectue l'allocation dynamique de memoire.

2. Préprocesseur
   - Le préprocesseur prepare le code source expansé pour la compilation.
   - Les directives permettent de gérer plusieurs versions du code dans un même fichier à l'aide de la **compilation conditionnelle**.
   - `assert()` est un outil de mise au point recommandé (version debug) ; il est désactivé si `NDEBUG` est défini avant l'include `<cassert>`.

3. Fichiers
   - toutes les entrées-sorties sont traitées avec des flots (streams).
   - les sorties formatées ont l'avantage de pouvoir être éditées avec un éditeur de texte
   - **`getline()`** et l'usage d'un **`istringstream`** permettent de lire séquentiellement un fichier, du début à la fin, une ligne à la fois.

rafael.pires@epfl.ch

Merci