



Chapitre 5: Les flux

Plan du chapitre

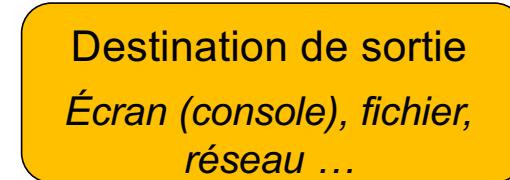
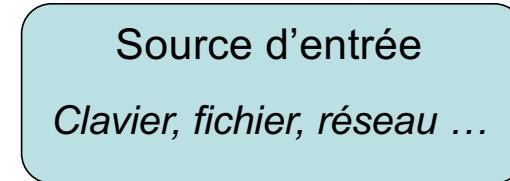
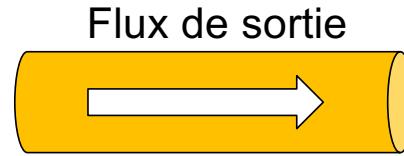
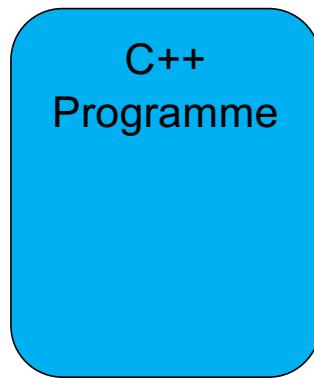


- 1.** Introduction aux flux d'entrée / sortie (I/O streams)
- 2.** Sortie et entrée standard
- 3.** Lire et écrire des fichiers
- 4.** Lire et écrire des chaînes de caractères
- 5.** Sortie formatée
- 6.** États des flux et validation des entrées
- 7.** Fonctions utiles

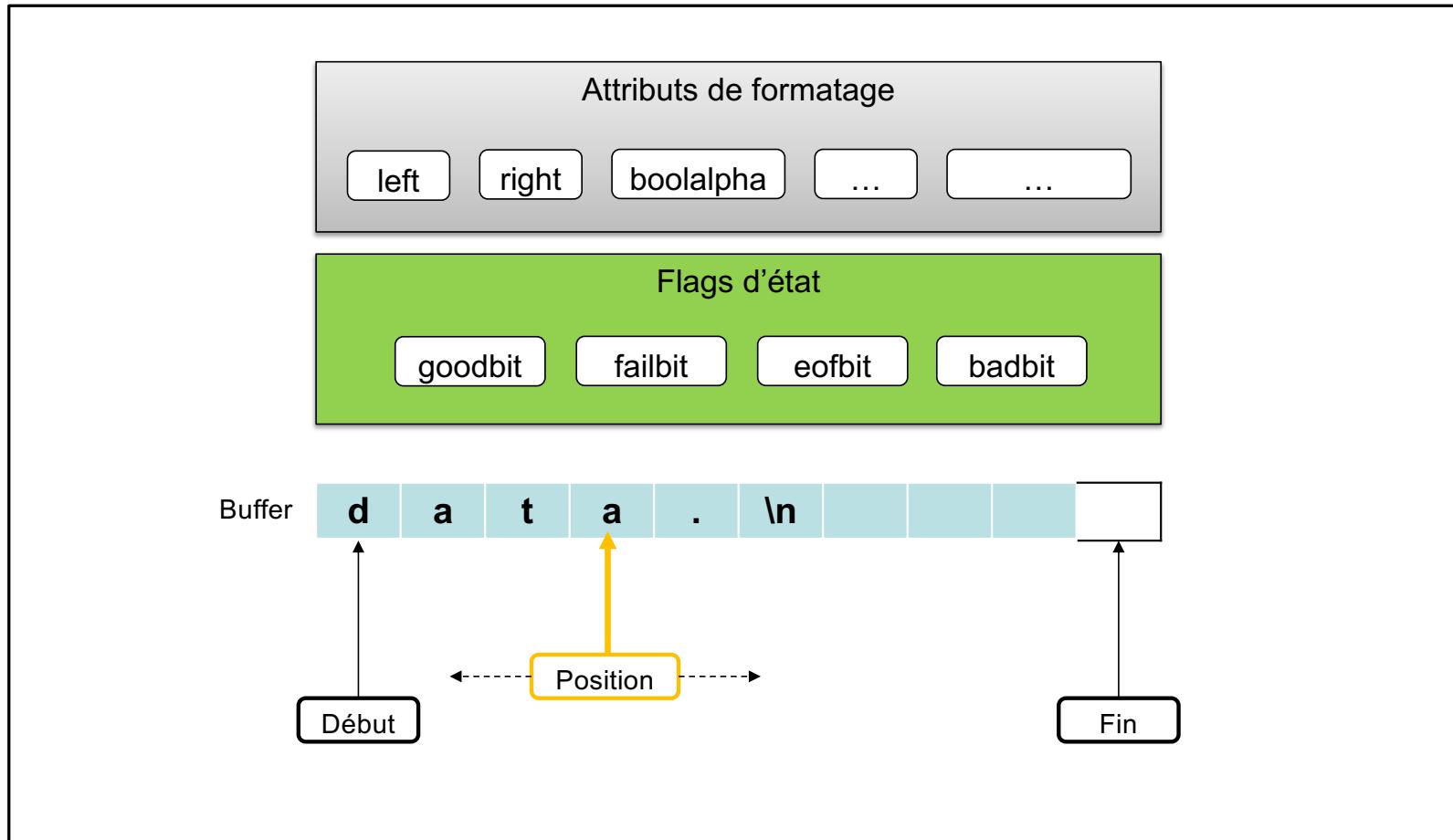


1. Introduction aux flux d'entrée / sortie (I/O streams)





- Un **flux** est un **canal** permettant au programme de **recevoir ou envoyer des données**.
- On distingue alors les **flux d'entrée** et les **flux de sortie**.
- Un flux peut être connecté à un périphérique ou un fichier.

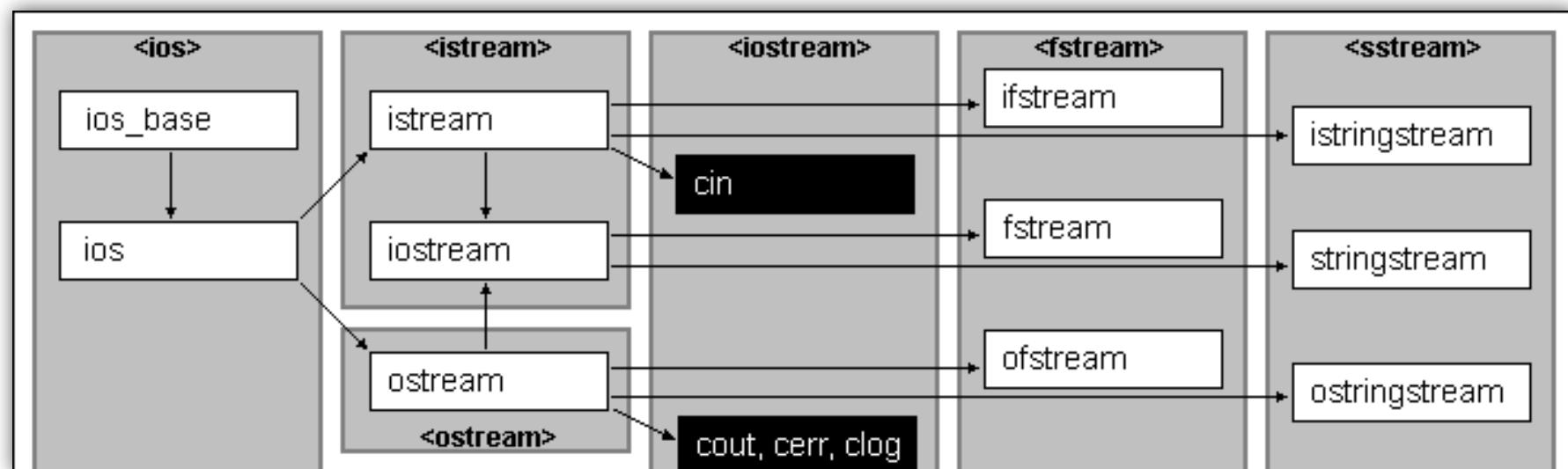




- Certain flux «standard» sont prédéfinis (connectés) :
 - `cin` – un flux **d'entrée** lié à l'entrée standard (typiquement le clavier)
 - `cout` – un flux de **sortie** lié à la sortie standard (typiquement la console)
 - `cerr` – un flux de **sortie d'erreurs** sans buffer (**immédiate**) lié à la sortie standard (typiquement la console).
 - `clog` – un flux de **sortie d'erreurs** avec buffer (**pas immédiate**) lié à la sortie standard (typiquement la console).



Introduction aux flux



<https://cplusplus.com/reference/ios/>

Un organigramme simplifié de l'organisation des flux en C++

Les opérateurs << et >>



- L'opérateur d'**insertion (<<)** est utilisé pour mettre des informations dans un **flux de sortie**.
- L'opérateur d'**extraction (>>)** pour **lire** les informations d'un **flux d'entrée**.
- Les **deux** opérateurs **retournent une référence** au **flux** concerné.
- Cela **permet** de l'**appliquer plusieurs fois** de suite.

```
cout << "Hello";
```

```
cin >> nbCanette;
```

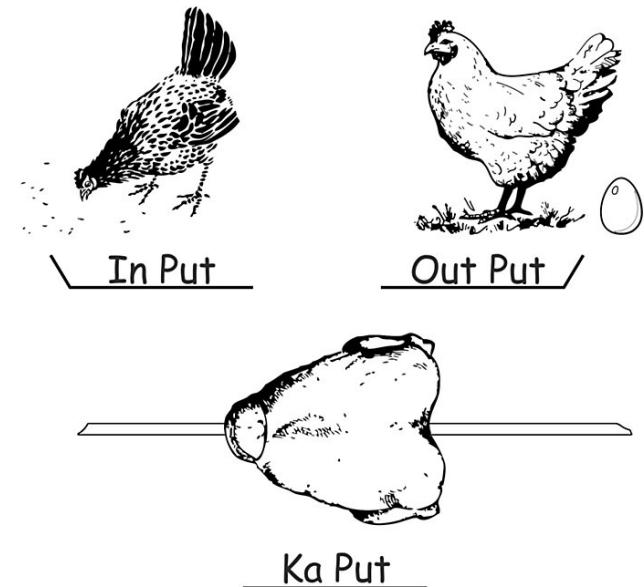
```
cout << "Valeur = " << valeur;
```

```
cin >> valeurL >> valeurH;
```



Chicken Lifestyle

2. <iostream> Entrée et sortie standard





- La librairie standard `<iostream>` définit les objets **permettant** l'utilisation de la **sortie et l'entrée** standard.

```
extern istream cin;
extern ostream cout;
extern ostream cerr;
extern ostream clog;
```

- `<iostream>` inclut `<ostream>` (flux de sortie) et `<istream>` (flux d'entrée)
- Elle est à **inclure dans votre programme C++** si vous utilisez ses objets.
- Par défaut, **l'entrée** standard est liée au clavier et la **sortie** standard est liée à l'écran (console).
- La **sortie** standard peut être **redirigée** selon le besoin, vers un fichier par exemple (c.f. slide 21).



- `std::cout` est utilisé pour afficher des **informations** à la sortie standard, typiquement l'écran.
- `std::cerr` est utilisé pour afficher les **erreurs immédiatement** (sans buffer) à la sortie standard, typiquement l'écran.
- `std::clog` est utilisé pour **afficher les logs** à la sortie standard, typiquement l'écran.



std::cout, cerr, et clog

```
int valeur;
int nbErreurs = 0;

cout << "Entrez un entier < 100: ";
cin >> valeur;

while (valeur >= 100) {
    nbErreurs++;
    cerr << "Erreur : saisie incorrecte, réessayez." << endl;
    cin >> valeur;
}

if (nbErreurs > 0) {
    clog << "Log : Erreurs de saisie (" << nbErreurs << ")" << endl;
}

cout << "Valeur = " << valeur << endl;
```

```
Entrez un entier < 100: 200
Erreur : saisie incorrecte, réessayez.
150
Erreur : saisie incorrecte, réessayez.
88
Log : Erreurs de saisie (2)
Valeur = 88
```



- `std::cin` est utilisé pour lire des **données** depuis l'entrée standard, typiquement le clavier.
- `cin` a besoin d'une **variable** pour **stocker** les données lues.
- La saisie de l'utilisateur est **terminée** par le bouton «**Entrée**».
- On peut saisir plusieurs variables sur une même ligne, elles peuvent être séparées par un espace ou en appuyant sur le bouton «**Entrée**».

```
int x = 0, y = 0;  
cin >> x >> y;  
cout << x << " -- " << y;
```

```
42 50  
42 -- 50
```

```
42  
50  
42 -- 50
```



3. <fstream> lire et écrire des fichiers

IT FEELS WEIRD THAT IT'S 2019 AND YET I STILL SOMETIMES FIND THAT THE EASIEST WAY TO MOVE A FILE AROUND IS TO EMAIL IT TO MYSELF.



IF ONLY THERE WERE A BETTER WAY...



MY NEW BOOK **HOW TO** IS OUT NEXT WEEK! IF YOU WANT TO LEARN HOW TO SEND DATA, YOU CAN VISIT BLOG.XKCD.COM FOR A SNEAK PREVIEW OF CHAPTER 19: HOW TO SEND A FILE.

EXCLUSIVE ADVICE FROM **HOW TO**:
WHEN SENDING A FILE, IT HELPS TO KNOW WHICH PART OF YOUR DEVICE THE FILE IS STORED IN.

FILES ARE
USUALLY IN
THIS PART





- Les opérations I/O depuis/vers des fichiers sont similaires aux I/O standards.
- Il faut inclure `<fstream>` dans votre programme.
- Contrairement à `cout`, `cin`, `cerr`, et `clog`, qui sont prêts à l'emploi, les flux de fichiers doivent être explicitement configurés.
 1. *Ouvrez* le fichier avant de toute autre opération.
 2. *Ecrivez / lisez* des données avec les opérateurs `<<` ou `>>`
 3. *Fermez* le fichier une fois vos opérations terminées.
- Nous allons considérer deux objets de la librairie `<fstream>` :
 - `std::ifstream` pour la *lecture*
 - `std::ofstream` pour l'*écriture*



Dans le mode «out» (par défaut), le fichier sample.txt

- est **créé** s'il n'existe pas.
- est **écrasé** s'il existait.

sample.txt

Ceci est la ligne 1
Ceci est la ligne 2

```
#include <iostream> // std::cerr
#include <fstream> // std::ofstream

int main() {
    std::ofstream file_out;

    // ouvrir le fichier en mode écriture
    file_out.open("sample.txt");

    if (!file_out) { // l'ouverture a échoué
        std::cerr << "Erreur d'ouverture du fichier\n";
        return 1;
    }

    // écrire dans le fichier
    file_out << "Ceci est la ligne 1\n";
    file_out << "Ceci est la ligne 2\n";

    // fermer le fichier
    file_out.close();
}
```



- Si nous souhaitons **ajouter** au fichier **sans l'écraser**, nous pouvons utiliser le mode **ios::app** pour ouvrir le fichier.

sample.txt

Ceci est la ligne 1
Ceci est la ligne 2
Ceci est la ligne 3

```
#include <iostream> // std::cerr
#include <fstream> // std::ofstream

int main() {
    std::ofstream file_out;

    //ouvrir le fichier en mode ajout
    file_out.open("sample.txt", ios::app);

    if (!file_out) { // l'ouverture a échoué
        std::cerr << "Erreur d'ouverture du fichier\n";
        return 1;
    }

    // écrire dans le fichier
    file_out << "Ceci est la ligne 3\n";

    // fermer le fichier
    file_out.close();
}
```



- Pour lire un fichier, il suffit de l'ouvrir dans un `std::ifstream`
- Ci-contre, on peut utiliser `file_in` exactement comme on utiliserait `cin`

```
#include <iostream> // std::cout
#include <fstream> // std::ifstream
#include <string> // std::string, std::getline

int main() {
    std::ifstream file_in;

    // ouvrir le fichier en lecture
    file_in.open("sample.txt");

    // Tant qu'il reste encore des choses à lire
    while (file_in) {
        std::string une_ligne;
        // Lire une ligne à la fois
        std::getline(file_in, une_ligne);
        std::cout << une_ligne << '\n';
    }

    // fermer le fichier
    file_in.close();
}
```

Ceci est la ligne 1
Ceci est la ligne 2
Ceci est la ligne 3



Ouvrir et fermer implicitement les fichiers

- Il n'est pas indispensable d'utiliser explicitement les méthodes `open` et `close`
 - le fichier est ouvert automatiquement si l'on donne le nom de fichier (et le mode d'ouverture) en paramètre d'initialisation
 - Le fichier est fermé automatiquement quand l'objet `ifstream` ou `ofstream` sort de scope
- Les 2 codes suivants ont le même effet

```
std::ofstream file_out;
file_out.open("sample.txt");
file_out << "contenu du fichier\n";
file_out.close();
```

```
{
    std::ofstream file_out("sample.txt");
    // ouverture par initialisation
    file_out << "contenu du fichier\n";
} // fermeture par sortie de scope
```

- Par contre, avec l'approche explicite, on peut réutiliser `file_out` pour ré-ouvrir le même fichier ou en ouvrir un autre

HE^{VD} IG Exemple



```
// ouvrir implicitement le fichier
std::ofstream file_out("sample.txt", std::ios::out);

// y écrire quelques lignes
file_out << "Ceci est la ligne 1\n";
file_out << "Ceci est la ligne 2\n";

// fermer explicitement le fichier
file_out.close();

// Oups, nous avons oublié quelque chose
// réouvrir explicitement le fichier en mode append
file_out.open("sample.txt", std::ios::app);
file_out << "Ceci est la ligne 3\n";

// refermer explicitement le fichier
file_out.close();
```



Redirection des flux

- On peut rediriger les flux cin, cout, cerr, et clog d'un programme vers des fichiers en le lançant **en ligne de commande** avec les options <, 1> et 2>. Par exemple,

```
prog.exe < input.txt 1> out.txt 2> error.txt
```

- < redirige le flux d'entrée cin pour qu'il lise depuis le fichier input.txt
- 1> redirige la sortie cout vers le fichier out.txt
- 2> écrit les sorties de cerr et clog dans le fichier error.txt
- Pour rediriger la sortie en mode **append** (ajout sans écraser le fichier s'il existait), on utilise un double > :

```
prog.exe < input.txt 1>> out.txt 2>> error.txt
```



- On peut aussi les rediriger directement depuis le code C++ en **changeant le buffer** utilisé par le flux
- La fonction **rdbuf sans argument** retourne un pointeur vers le buffer du flux
- La fonction **rdbuf avec un argument** permet de spécifier un nouveau buffer, par exemple celui d'un autre flux.
- Cette méthode est valable pour tout type de flux.

```
std::ofstream file_out("sortie.txt", std::ios::out);

// backup du buffer de cout
auto backup = std::cout.rdbuf();

// rediriger cout vers le fichier
// en remplaçant son buffer par celui de file_out
std::cout.rdbuf(file_out.rdbuf());
std::cout << "Ligne écrite dans le fichier \n";

// remettre son buffer d'origine
std::cout.rdbuf(backup);
std::cout << "Ligne affichée sur la console \n";

file_out.close();
```



4. <sstream> Lire et écrire des chaines de caractères

```
stringstream out;  
  
out << "C++; "  
    << char(47) << '\x2F'  
    << " makes C bigger,"  
    << " returns old value";  
  
cout << out.str();
```

```
C++; // makes C bigger, returns old value
```

HE^{vd} IG **stringstream**



- Le type **stringstream**, défini dans la **<sstream>**, permet de créer un **flux** qui lit ou écrit dans une **string** plutôt que dans la console
- Il peut être initialisé vide ou par constructeur avec une **string**
- L'accès à cette chaîne en lecture ou écriture s'effectue via la méthode **str()**
- Toutes les opérations vues sur **cin** et **cout** sont applicables,
 - ce qui permet par exemple de **convertir** un nombre en chaîne ou inversement
 - sortie tampon pour un affichage ultérieur
 - ou pour traiter l'entrée ligne par ligne

HE^{vd} IG stringstream



- Conversion de nombre en chaîne

```
int nombre = 123;                      // nombre à convertir
stringstream convert;                   // flux de conversion
convert << nombre;                     // affichage comme pour cout
string chaine = convert.str();          // chaine contient "123"
```

- Conversion de chaîne en nombre

```
string text = "456";                  // chaîne à convertir
stringstream convert(text);           // flux de conversion
int nombre;                          // nombre contient la valeur 456
convert >> nombre;
```



C++11 a introduit des fonctions simplifiant grandement ces **conversions**

- **string → int** avec **stoi** (stol, stoul, stoll, stoull vers d'autres types entiers)
- **string → double à** avec **stod** (stof, stold vers d'autres types réels)
- **nombre → string** avec **to_string**

```
int    entier = stoi("123");
double reel   = stod("3.14");
string chaine = to_string(entier) + " " + to_string(reel);

// chaine contient "123 3.140000"
// to_string ne permet pas de choisir le format
```



5. <iomanip> Sortie formatée



HE^{VD} IG Les manipulateurs



- La sortie peut être formatée via l'utilisation des manipulateurs.
- Ces manipulateurs se trouvent dans deux librairies :
 - `<iostream>` pour les manipulateurs non paramétriques, déjà inclue par `<iostream>`
 - `<iomanip>` qui est à inclure pour les manipulateurs paramétriques.

En-tête	Manipulateur	Explication	Persistante
<code><iostream></code>	<code>internal</code>	Justifie à gauche le signe du nombre et à droite la valeur	Oui
	<code>left</code>	Justifie à gauche le signe et la valeur	Oui
	<code>right</code>	Justifie à droite le signe et la valeur	Oui
<code><iomanip></code>	<code>setfill(char)</code>	Définit le paramètre comme caractère de remplissage	Oui
	<code>setw(int)</code>	Définit la largeur du champ	Non

HE^{VD} IG Les manipulateurs



int a = -12345; int wcol = 10; cout << "0123456789" << endl;	0123456789
cout << a << endl; // formatage par défaut	-12345
cout << setw(wcol) << a << endl; // colonne de 10 char	-12345
cout << left << setw(wcol) << a << endl; // justifié à gauche	-12345
cout << right << setw(wcol) << a << endl; // justifié à droite (choix par défaut)	-12345
cout << internal << setw(wcol) << a << endl; // justifié interne	- 12345
cout << setw(wcol) << a << endl; // left, right, internal persistent	- 12345
cout << a << endl; // setw ne persiste pas. s'applique à 1 seul affichage	-12345
cout << setfill('*') << setw(wcol) << a << endl; // char de remplissage autre que ' '	-****12345
cout << setw(wcol) << a << endl; // setfill persiste	-****12345
cout << "Rempli avec " << cout.fill() << endl; // .fill() retourne le char de remplissage	Rempli avec *



- Les opérations d'écriture ne sont pas transférées immédiatement à la sortie mais conservées temporairement dans un buffer, puis envoyées en bloc, ce qui est plus efficace
 - `std::flush` force à vider ce buffer vers la sortie
- Pour passer à la ligne suivante, il suffit d'écrire le caractère '`\n`' dans le flux.
 - `std::endl` est équivalent à écrire '`\n`' puis `std::flush` dans le flux
 - S'il n'est pas nécessaire de «flusher», utiliser '`\n`' seul est plus efficace

```
// Les 4 lignes suivantes produisent la même sortie
cout << "Hello" << endl << "World !" << endl;
cout << "Hello\n" << flush << "World!\n" << flush;
cout << "Hello\n" "World!\n" << flush;
cout << "Hello\n" "World!" << endl;
```

Hello
World!

HE^{VD} IG std::boolalpha



- Par défaut, afficher un booléen avec cout affiche 0 pour `false` et 1 pour `true`
- Ce comportement peut être modifié avec les modificateurs de flux `std::boolalpha` et `std::noboolalpha`

```
bool vrai = true, faux = false;

cout << vrai << " " << faux << endl;
cout << boolalpha;
cout << vrai << " " << faux << endl;
cout << noboolalpha;
cout << vrai << " " << faux << endl;
```

1 0
true false
1 0



6. États des flux et validation des entrées





- Le code suivant gère les entrées utilisateur via une boucle `do...while` qui répète le message original en cas d'erreur de l'utilisateur

```
int valeur;
do {
    cout << "Entrez un entier < 100: ";
    cin >> valeur;
} while (valeur >= 100);

cout << "Valeur = " << valeur << endl;
```

```
Entrez un entier < 100: 2015
Entrez un entier < 100: 421
Entrez un entier < 100: 89
Valeur = 89
```

- Que se passe-t-il si l'utilisateur entre une lettre plutôt qu'un nombre ?



```
int valeur;
do {
    cout << "Entrez un entier > 0 : ";
    cin >> valeur;
} while (valeur > 1);

cout << "Valeur = " << valeur << endl;
```

- La valeur saisie « a » ne permet pas de construire un entier
- Le caractère saisi n'est pas consommé et reste dans le flux d'entrée cin
- La lecture n'ayant pas abouti, la variable est mise à zéro (C++11)
- La prochaine lecture essaie à nouveau de lire le caractère 'a' ... et on entre dans une boucle infinie

```
Entrez un entier > 1 : -1
Entrez un entier > 1 : a
Entrez un entier > 1 :
...
...
```



Pour gérer une erreur de ce type, il est nécessaire de

1. **Déetecter l'erreur** de lecture via les fonctions qui renseignent sur l'état du flux
 - `cin.good()` – pas d'erreur, le flux est fonctionnel
 - `cin.bad()` – une erreur irrémédiable a eu lieu sur le flux
 - `cin.eof()` – la fin du fichier a été dépassée
 - `cin.fail()` – une erreur a eu lieu : bad, ou juste un mauvais formatage
2. **Effacer l'indication** d'erreur avec `cin.clear()`
3. **Vider le tampon** du flux avec `cin.ignore(streamsize n, int delim = EOF)` qui retire jusqu'à n caractères du flux, ou jusqu'à ce qu'on rencontre le caractère `delim` (par défaut la fin du fichier)



- Le code suivant détecte une erreur de lecture, remet le statut du flux `cin` à `good`, et vide le buffer d'entrée jusqu'au prochain saut de ligne

```
cin >> valeur;
if (cin.fail()) {
    cin.clear();
    cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

- Le test de détection de l'erreur peut également être réécrit

```
if (not cin.good()) ...

if (not cin)
// car la conversion de cin en bool équivaut à appeler .good()

if (not (cin >> valeur))
// car l'expression (cin >> valeur) retourne cin
```



7. Fonctions utiles



"I spoke to my old math teacher today.
He says don't give up hope, someday
algebra WILL be useful to me."
I know! I'm still waiting for my first math problem!



Lecture d'une std::string avec >>

- Observons l'effet de l'opérateur >> utilisé pour lire une std::string

```
stringstream in(" \n \t Hello, World ! ");
int i{};
string s;
while (in >> s) {
    cout << ++i << " : " << s << endl;
}
```

```
1 : Hello,
2 : World
3 : !
```

- il passe les blancs (espace, tab, retour à la ligne) en début de lecture
- il lit dès le premier caractère non blanc et s'arrête dès le blanc suivant
- Ce comportement standard pour tout type de données n'est **pas toujours approprié**

```
string str;
cin >> str;           // l'utilisateur entre "James Bond"
cout << str << endl;
```

```
James
```

HE^{VD} IG std::getline



- La fonction `getline(flux, str)` résout ce problème en lisant toute une ligne depuis `flux` et en stockant le résultat dans la chaîne `str`

```
string str;
getline(cin,str);    // l'utilisateur entre "James Bond"
cout << str << endl;
```

James Bond

- Un 3^{ème} paramètre permet de spécifier un caractère de terminaison autre que '`\n`'

```
stringstream in("James Bond/John Doe/Jack Smith");
int i{};
while (not in.eof()) {
    string name;
    getline(in, name, '/');
    cout << ++i << " : " << name << endl;
}
```

1 : James Bond
2 : John Doe
3 : Jack Smith

- Notons que ce caractère n'est pas inclus dans `str`, mais est supprimé du flux



istream::get(char)

- La méthode `get` d'un `istream` permet de lire le flux caractère par caractère. Elle écrit dans le `char` passé par référence et retourne le flux lui-même

```
stringstream in("Une ligne avec des blancs\nUne autre ligne");
char c;
while (in.get(c)) { cout << c; }
```

Une ligne avec des blancs
Une autre ligne

- Son effet est à contraster avec l'utilisation de l'opérateur `>>` qui, même pour lire un `char`, passe les caractères blancs

```
stringstream in("Une ligne avec des blancs\nUne autre ligne");
char c;
while (in >> c) { cout << c; }
```

UneligneavecdesblancsUneautreligne

HE^{VD} IG istream::get()



- La fonction `get()` existe aussi sans paramètre. Elle retourne le caractère lu.
- S'il est possible que cette lecture échoue, cette syntaxe peut rendre plus compliquée la gestion d'erreur

```
stringstream in("Une ligne avec des blancs\nUne autre ligne");
while (true) {
    char c = in.get();
    if(in.eof())
        break;
    cout << c;
}
```

Une ligne avec des blancs
Une autre ligne

- Attention, la méthode retourne un `int`, qu'il faut probablement convertir en `char`



istream::unget(), istream::putback(char)

- La méthode unget() remet le dernier caractère lu dans le flux afin qu'il puisse être relu lors du prochain appel.

```
stringstream in("abcdefg");
char c; in.get(c); cout << c << endl;
in.unget();
while(in.get(c)) cout << c;
```

```
a
abcdefg
```

- La méthode putback(char) remet le caractère en paramètre dans le flux à la place de celui qui vient d'être lu

```
stringstream in("abcdefg");
char c; in.get(c); cout << c << endl;
in.putback('Z');
while(in.get(c)) cout << c;
```

```
a
Zbcdefg
```



istream::peek()

- La méthode peek() retourne le dernier caractère dans le flux sans le consommer.

```
stringstream in("abcdefg");
char c = in.peek();
cout << "caractère " << c;
if (c >= 'a' and c <= 'z')
    cout << " minuscule";
```

caractère a minuscule



Résumé



"Skip the blow by blow. Just condense it into a couple of barks."



<iostream>

- Variables globales `cin`, `cout`, `cerr` et `clog`
- Lecture et écriture depuis l'entrée et vers la `sortie standard`.

<fstream>

- Types `ifstream`, `ofstream`, et `fstream`
- Lecture et écriture depuis et vers des `fichiers`

<stringstream>

- Types `istringstream`, `ostringstream` et `sstream`
- Lecture et écriture depuis et vers des `chaines de caractère`



<ostream>

- Type `ostream` et manipulateurs de flux `endl`, `flush`
- Inclus par les 3 précédents headers. Nous l'utiliserons directement quand nous écriront nos propres opérateurs <<

<istream>

- Types `istream` et `iosteam`
- Inclus par les 3 précédents headers. Nous l'utiliserons pour nos propres opérateurs >>



<ios>

- Classe mère de toutes les autres: `ios`
- Manipulateurs de flux sans paramètres : `left`, `right`, `internal`, `boolalpha`, et d'autres pour le formatage des nombres entiers et réels (chapitre 6)
- Ce header est inclus par tous les précédents

<iomanip>

- Manipulateurs de flux avec paramètres : `setw`, `setfill`, et d'autres pour le formatage des nombres entiers et réels (chapitre 6), des valeurs monétaires, et des dates (pas vus en PRG1)
- Ce header doit être inclus explicitement pour utiliser ses manipulateurs