

Information, Calcul et Communication (partie programmation) : Cours de programmation (C++) Entrées/Sorties

Jamila Sam

Laboratoire d'Intelligence Artificielle
Faculté I&C

Interagir avec le monde : les entrées/sorties

Jusqu'à présent, les possibilités d'interaction de vos programmes sont restées très limitées :

quelques exemples rudimentaires de saisie au clavier (`cin`) et d'affichage à l'écran (`cout`).

On peut évidemment faire beaucoup plus !

Les interactions d'un programme avec « l'extérieur » sont gérées par des **instructions d'entrée/sortie** et ce qu'on appelle des « **flots** ».

Un flot correspond à un canal d'échange de données entre le programme et l'extérieur.

`cin` est le nom de la variable associée par défaut au flot d'entrée. On l'appelle « **entrée standard** ».

`cout` est le nom de la variable associée par défaut au flot de sortie.

On l'appelle « **sortie standard** ».

Objectifs du cours d'aujourd'hui

- ▶ Présenter les bases des entrées/sorties en C++ :
 - ▶ les entrées/sorties clavier/écran (`cin/cout`)
 - ▶ les fichiers
 - ▶ formatage des entrées/sorties

Buts des entrées/sorties

- ▶ **intégrer**, au sein du programme, des *données issues de l'extérieur*, par exemple saisies au clavier ou lues dans un fichier
 - ☞ nécessité de **fonctions d'entrée**, permettant d'associer des informations externes aux variables du programme
- ▶ **communiquer des données internes**
 - ☞ nécessité de **fonctions de sortie**, permettant de visualiser (en spécifiant éventuellement le format) les valeurs des variables du programme.
- ▶ **sauvegarder les résultats produits** par les programmes, par exemple pour des utilisations ultérieures
 - ☞ nécessité de **fonctions de lecture/écriture**, permettant
 1. de stocker dans des **fichiers** les données produites par les programmes
 2. de récupérer dans le programme les données contenues dans de tels fichiers

Entrées-sorties clavier/écran

À l'heure actuelle, les interactions les plus simples (« naturelles » ?) avec un programme se font via l'**écran** pour la sortie et le **clavier** pour les entrées.

En C++, ces « **flots** » sont représentés respectivement par `cout` et `cin`.

`cin` et `cout` sont définis dans le fichier de définitions `iostream`. Pour les utiliser il faut donc faire

```
#include <iostream>
```

au début du programme.

Lecture au clavier

```
cin >> var1 >> var2 >> ... ;
```

permet à l'utilisateur de saisir au clavier une liste de valeurs `val1`, `val2`, ... qui seront stockées dans les variables `var1`, `var2`, ...

Remarque : Lorsque plusieurs valeurs sont lues à la suite, le **caractère séparateur** de ces valeurs est le **retour à la ligne** ou l'**espace**.



Attention ! On ne peut donc pas lire en une fois une ligne entière contenant des blancs (par exemple saisie d'un nom composé) avec `cin`

il faut utiliser `getline` pour lire une ligne entière (contenant des espaces) Conseil : faire précéder les « `getline(cin, ...)` » de :

```
cin >> ws;
```

Exemple :

```
string phrase_a_lire;  
...  
cin >> ws;  
getline(cin, phrase_a_lire);
```

Affichage à l'écran

```
cout << expr1 << expr2 << ... ;
```

affiche à l'écran les valeurs des **expressions** `expr1`, `expr2`,

Remarque : par défaut, les valeurs affichées ne sont pas séparées par des espaces

ajoutez des ' ' (1 espace), si nécessaire.

Exemple d'affichage :

```
int a(1);  
cout << "intervalle : [" << -a << ", " << a  
    << "]" << endl;
```

```
intervalle : [-1, 1]
```

Remarque : `endl` permet à l'affichage d'aller à la ligne.

Exemple simple d'entrée-sortie

```
#include <iostream>  
main () {  
    int i ;  
    double x;  
    cout << "Valeurs pour i et x : " << flush;  
    cin >> i >> x ;  
    cout << "lus : " << i << ", " << x << endl;  
}
```



Remarque : `flush` est un manipulateur du flot qui impose la « vidange » de la mémoire tampon associée.

En clair, dans l'exemple ci-dessus cela **force** l'écriture du texte (avant de continuer).



Vous avez peut être remarqué que la lecture sur `cin` est parfois « *capricieuse* » (e.g. quand on ne lui donne pas ce qu'elle attend).

Exemple tiré des séries d'exercices :

```
int lu;
do {
    cout << "entrez un nombre entre 1 et 10 : "
        << flush;
    cin >> lu;
} while ((lu < 1) || (lu > 10));
```

Si vous tapez 'a' (ou n'importe quoi qui ne soit pas un nombre) ➡ boucle infinie !!

Comment éviter cela ?

- ➡ en contrôlant l'état du flot `cin` (et en « jetant à la poubelle » ce qui ne convient pas)



En plus de `cin` et `cout`, il existe une **sortie d'erreur standard**, `cerr`.

Par défaut `cerr` est envoyée sur le terminal, comme `cout`. Mais il s'agit bien d'un flot séparé !

On peut en effet sous plusieurs systèmes d'exploitation séparer la sortie standard de la sortie d'erreur standard : (commandes Unix)

```
ls > bidon
more bidon # bidon contient le resultat de la commande ls
rm bidon

ls nonononono > bidon
more bidon # bidon est vide : le flot d'erreur n'est pas envoyé
# dans le fichier, uniquement le flot de sortie
rm bidon
```

De plus `cerr` n'a pas de mémoire tampon. L'écriture sur `cerr` se fait donc directement (on n'a pas besoin de `flush`).

- ➡ Conseil : Pour afficher des messages d'erreur depuis votre programme, préférez `cerr` plutôt que `cout`.



```
int lu(0);
do {
    cout << "entrez un nombre entre 1 et 10 : " << flush;
    cin >> lu;

    if (cin.fail()) { // teste si cin "ne va pas bien"
        cout << "Je vous ai demandé d'entrer un nombre, "
            << "pas du charabia !" << endl;
        // remet cin dans un état lisible
        cin.clear();
        // "jette" tout le reste de la ligne
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
    }
} while ((lu < 1) || (lu > 10));
```

Remarque : Une autre solution consiste à ne lire l'entrée que **caractère par caractère** avec la fonction `get (c = cin.get())` et à traiter ensuite la séquence lue, de façon totalement contrôlée par notre programme. Mais cette bonne solution est trop avancée dans sa mise en œuvre (« **tokenization** ») pour être détaillée dans ce cours d'introduction.

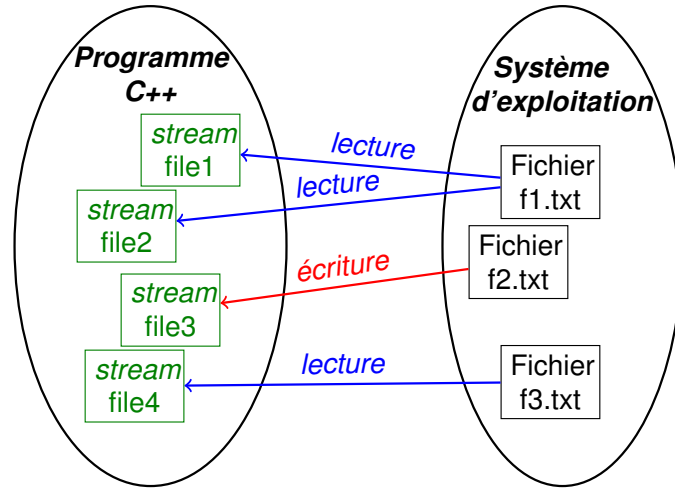
Pour **sauvegarder** de façon permanente (*i.e.* rendre disponible après la fin de l'exécution du programme) les données produites par un programme, une solution est d'utiliser le **système de fichiers** fourni par le système d'exploitation (revoir cours 2).

Il est donc nécessaire d'avoir un objet informatique permettant de gérer les entrées et les sorties du programme vers le système de fichiers.

Cet objet informatique (appelé un **flot**, « stream » en anglais) **représente** en fait *un lien à un fichier physique* stocké dans le disque et géré par le système de fichiers.

L'idée de base des flots est de **séparer l'aspect logique** des entrées-sorties (*i.e.* leur intégration dans des programmes) de leur **aspect physique** (*i.e.* leur réalisation par le biais de périphériques particuliers).

Les entrées/sorties avec des fichiers (2)



Utilisation des flots

Mécanisme général pour la mise en œuvre d'entrées-sorties :

- ▶ **création d'un flot** (d'entrée ou de sortie) par la **déclaration** d'une variable du type correspondant (`ifstream` ou `ofstream`).
- ▶ **lien** de la variable déclarée **avec un dispositif** d'entrée-sortie **physique** (fichier)
- ▶ **utilisation** de la variable déclarée et liée pour effectivement réaliser les entrées-sorties
- ▶ **fermeture** du flot

La plupart de ces actions se font en appliquant des fonctions spécifiques définies pour les types `stream`.

Ces fonctions sont appelées avec la syntaxe suivante :

```
nom_de_stream.nom_de_fonction(arg1, ...);
```

Les types « stream »

Comme pour de nombreuses autres notions C++, pour pouvoir utiliser les flots, il faut tout d'abord inclure les fichiers de définitions correspondant, `iostream` et `fstream` :

```
#include <iostream>
#include <fstream>
```

Deux nouveaux types sont alors disponibles :

- ▶ `ifstream` (pour *input file stream*) qui définit un flot d'**entrée** (similaire à `cin`)
- ▶ `ofstream` (pour *output file stream*) qui définit un flot de **sortie**. (similaire à `cout`)



Attention ! On ne peut jamais copier de flot (`ifstream`, `ofstream`, ...), ni les passer par valeur. Il faut impérativement les passer par référence !

Création d'un flot

Pour créer un flot, il suffit de déclarer une variable du type correspondant.

Exemple :

```
ifstream entree;
ofstream sortie;
```

déclare deux variables `entree` et `sortie`, respectivement de type `ifstream` et `ofstream`.

Lien avec un fichier

Dans le cas des **fichiers textes** (fichiers lisibles par les humains), l'association d'un flot d'entrée-sortie avec le fichier se fait par le biais de la fonction spécifique `open`.

Exemple :

```
ifstream entree;  
entree.open("test.txt");
```

associe le stream `entree` avec le fichier physique `test.txt`.

Lien avec un fichier (3)

Par défaut, un « `ofstream` » ouvre le fichier en mode « **écrasement** » (c'est-à-dire détruit le contenu du fichier si il existe déjà)

On souhaite parfois pouvoir ouvrir le fichier en mode « **ajout** » (« append ») (c'est-à-dire écrire en fin de fichier)

Cela se fait aussi en ajoutant un argument supplémentaire à `open` :

```
ofstream sortie;  
sortie.open("a_poursuire.txt", ios::out|ios::app);
```

Dans le cas de fichiers en binaire :

```
ofstream sortie("a_completer.data",  
ios::out|ios::binary|ios::app);
```

Lien avec un fichier (2)

Dans le cas des **fichiers binaires**, il faut ajouter un argument supplémentaire :

`ios::in|ios::binary` pour la lecture
`ios::out|ios::binary` pour l'écriture

Exemple :

```
ifstream entree;  
entree.open("a_lire.zip", ios::in|ios::binary);  
  
ofstream sortie;  
sortie.open("a_ecrire.exe", ios::out|ios::binary);
```

Lien avec un fichier (4)

L'association du stream d'entrée ou de sortie avec un fichier particulier peut aussi se faire directement par une **initialisation** lors de la déclaration du stream :

Exemple :

```
ifstream entree("test.txt") ;
```

déclare la variable `entree` et l'associe au fichier texte `test.txt`.

Dans le cas de fichiers en binaire :

```
ifstream entree("a_lire.zip",  
ios::in|ios::binary);  
ofstream sortie("a_ecrire.exe",  
ios::out|ios::binary);
```

Dans le cas de fichiers en écriture en mode « ajout » :

```
ofstream sortie("a_poursuivre.txt",  
ios::out|ios::app);
```

Lien avec un fichier (remarque C++98)

C++11 : **OUF!** Ce n'est plus vrai... **oubliez!**



C++98 :

Attention! L'utilisation directe d'une variable de type `string` pour donner un nom de fichier n'est malheureusement **pas** possible.

Le nom du fichier doit être une chaîne de caractères littérale ("`...`").

Les variables de type `string` doivent être utilisées par le biais de la fonction `c_str()` spécifique aux variables de type `string`.

Exemple :

```
string chaine1("un_fichier");

// ofstream sortie1(chaine1); // NE COMPILE PAS !!
// Bonne solution :
ofstream sortie1(chaine1.c_str());
```

Utilisation des flots (2)

Une fonction utile pour tester si la lecture d'un fichier (associé à un `ifstream`) est terminée est la fonction `eof`.

Exemple :

```
while ( ! entree.eof() )
    entree >> mot;
```

Utilisation des flots

L'utilisation des variables de type `ifstream` ou `ofstream` dans les programmes pour réaliser les entrées-sorties se fait de la même façon que pour les flots particuliers `cin` et `cout`, à l'aide des opérateurs `<<` et `>>`.

Exemple :

```
ifstream entree("fichier_entree");
ofstream sortie("fichier_sortie");
string mot;
...
entree >> mot; // lit un mot dans le fichier
                //"fichier_entree"
sortie << mot; // écrit ce mot dans le fichier
                //"fichier_sortie"
```

Utilisation des flots – Remarques

Dans un contexte de test, les opérateurs `<<` et `>>`, ainsi que `getline`, retournent en fait « `not flot.fail()` »,

Exemples :

```
if (cin >> x) { // on a pu lire x \}

while (entree >> nom >> age) {
    // on a eut un nom (1 seul mot) et un age
}

while (getline(entree, ligne)) {
    // on peut traiter la ligne
}
```

Utilisation des flots – Remarques

En cas d'erreur, une fonction utile pour tester si l'erreur est liée à la fin du fichier (ou à autre chose) est encore la fonction `eof`.

Exemple :

```
while (entree >> age) { ... }  
if (not entree.eof()) {  
    cerr << "je voulais un entier" << endl;  
    //etc...  
}
```

Robustesse des accès fichier

Il est **important** de **vérifier** que le lien du flot avec le fichier **s'est bien passé** (e.g le fichier existe, il est bien lisible, ...).

Ceci est fait en **testant l'état du flot**, par exemple après l'appel à la fonction `open()`.

En cas de problème, la fonction `fail()` du flot retourne la valeur `true` si le flot n'est pas dans un état correct pour la prochaine opération (lecture ou écriture). Elle renvoie `false` si le flot est prêt.

(voir exemple suivant)

Fermeture des flots

La fermeture du stream se fait par la fonction

`close`

Exemple :

```
ifstream entree("fichier_entree");  
...  
entree.close();
```



Attention ! NE PAS oublier de fermer tout fichier ouvert !
En particulier en écriture : vous risqueriez sinon d'avoir des surprises...

Exemple de lecture à partir d'un fichier

Exemple de programme de lecture d'un fichier texte de nom « `test` » :

```
int main() {  
    string nom_fichier("test");  
    ifstream entree(nom_fichier.c_str());  
  
    if (entree.fail()) {  
        cerr << "Erreur : impossible de lire le fichier "  
            << nom_fichier << endl;  
    } else {  
        string mot;  
  
        while (!entree.eof()) {  
            entree >> mot ;  
            cout << mot << endl;  
        }  
        entree.close() ;  
    }  
}
```

Écriture dans un fichier

Exemple de programme d'écriture dans un fichier texte :

```
main() {
    string nom_fichier;

    cout << "Dans quel fichier voulez vous ecrire ? "
         << flush;
    cin >> ws;
    getline(cin, nom_fichier);

    ofstream sortie(nom_fichier.c_str());
    if (sortie.fail()) {
        cerr << "Erreur : impossible d'ecrire"
             << " dans le fichier " << nom_fichier
             << endl;
    } else {
        string phrase;
        cout << "Entrez une phrase : " << flush;
        cin >> ws;
        getline(cin, phrase);
        sortie << phrase << endl;
        sortie.close();
    }
}
```

Plan

- ▶ les entrées/sorties :
 - ▶ les entrées/sorties clavier/écran
 - ▶ les fichiers
 - ▶ **formatage des entrées/sorties**

Sorties formatées

Un certain nombre de paramètres pour le format des sorties peuvent être explicitement spécifiés.

Il peuvent être de deux formes :

- ▶ soit des **manipulateurs** appliqués à l'instruction `<<`
- ▶ soit des **options de configurations** pour une variable de type `ofstream` (dont `cout` en particulier)

(Note : certains formatages peuvent s'obtenir indifféremment des deux façons)

Pour pouvoir utiliser **les manipulateurs**, il faut utiliser le fichier de définitions `iomanip` :


```
#include<iomanip>
```

Les **manipulateurs** s'utilisent de la façon suivante :

```
cout << manipulateur << expression << ...
```

Manipulateurs

Exemple de quelques manipulateurs :

- ①  `dec`, `hex`, `oct` : choix de la base numérique : décimale, hexadécimale, octale...

```
cout << 16 << ", " << hex << 16 << ", "
     << oct << 16 << endl;
```

```
16, 10, 20
```

- ② `setprecision` : précision désirée dans l'affichage des nombres

```
for (int i(2); i <= 5; ++i)
    cout << setprecision(i) << i << " " << 123.46
        << endl;
```

```
2 1.2e+02
3 123
4 123.5
5 123.46
```

Manipulateurs (2)

- ③ `setw` : longueur d'une chaîne de caractères
la donnée qui suit ce manipulateur est affichée sur (au moins)
le nombre de caractères spécifiés avec un cadrage à droite.
Ceci est pratique pour représenter des nombre en colonnes :

```
int const col1(14);  
int const col2(5);  
cout << setw(col1) << "un :"  
    << setw(col2) << 1 << endl;  
cout << setw(col1) << "cent deux :"  
    << setw(col2) << 102 << endl;  
cout << setw(col1) << "tres froid :"  
    << setw(col2) << -18 << endl;
```

```
un :      1  
cent deux : 102  
tres froid : -18
```

Manipulateur setw (2)

`setw` peut aussi s'utiliser en **lecture**, pour lire au plus un nombre
donné de caractères.

```
string code;  
cout << "Donnez moi un code de 5 caracteres : "  
    << flush;  
cin >> setw(5) >> code;  
cout << "lu : " << code << endl;
```

```
Donnez moi un code de 5 caractères : abcdefgh  
lu : abcde
```

```
Donnez moi un code de 5 caractères : ab  
lu : ab
```

Manipulateur setw

Autre exemple :

```
for (int i(1); i<5; ++i)  
    cout << i << " " << setw(i) << 8 << endl;
```

```
1 8  
2  8  
3   8  
4    8
```

Manipulateurs (3)

- ④ `setfill` : change le caractère utilisé pour remplir les
alignements (par défaut c'est l'espace)

Exemple :

```
int const col1(14);  
int const col2(5);  
cout << setw(col1) << "un :"  
    << setw(col2) << 1 << endl  
cout << setw(col1) << "cent deux :"  
    << setw(col2) << 102 << endl  
cout << setw(col1) << "tres froid :"  
    << setw(col2) << -18 << endl;
```

```
un :....1  
cent deux :..102  
tres froid :..-18
```

Manipulateurs (4)

Les flots d'**entrée** ont aussi le manipulateur suivant, qui peut s'avérer utile :

- ⑤ `ws` : saute tous les caractères blancs

Exemple :

```
cin >> ws;
getline(cin, phrase);
```

option de flot (2)

- ③ `fixed/scientific` : affichage fixe ou scientifique (en puissance de 10) des nombres

```
cout.setf(ios::fixed);
cout << sqrt(200) << endl;
```

14.14214

```
cout.setf(ios::scientific);
cout << sqrt(200) << endl;
```

1.41421e+01

Les options des flots

Les **options** de flots sont configurées avec la fonction `setf`, de la façon suivante :

```
flot.setf(ios::option) ;
```

Quelques options :

- ① `showbase` : affiche la base numérique du nombre ;

```
cout.setf(ios::showbase);
cout << 16 << ", " << hex << 16 << ", "
    << oct << 16 << dec << endl;
```

16, 0x10, 020

- ② `showpoint` : affiche explicitement la partie fractionnaire des nombres réels ;

```
cout.setf(ios::showpoint);
cout << 16.0 << ", " << 16.1 << ", " << 16
    << endl;
```

16.000, 16.100, 16

option de flot (3)

- ④ `left` : affichage à gauche

```
cout << setw(15) << "abcde" << "x" << endl;
cout << setw(15) << "ab" << "x" << endl;
cout.setf(ios::left);
cout << setw(15) << "abcde" << "x" << endl;
cout << setw(15) << "ab" << "x" << endl;
```

```
          abcde
          abx
abcde      x
ab         x
```

Pour annuler une option faire :

```
flot.unsetf(ios::option) ;
```



Manipulateur ou option ?



```

out << hex
out << oct
out << dec
out << setprecision(p)
out << setw(w)
out << setfill(c)
in >> ws
out << showbase
out << showpoint
out << fixed
out << scientific
out << left
out << right

out.setf(ios::hex, ios::basefield)
out.setf(ios::oct, ios::basefield)
out.setf(ios::dec, ios::basefield)
out.precision(p)
out.width(p)
out.fill(c)
—
out.setf(ios::showbase)
out.setf(ios::showpoint)
out.setf(ios::fixed, ios::floatfield)
out.setf(ios::scientific, ios::floatfield)
out.setf(ios::left, ios::adjustfield)
out.setf(ios::right, ios::adjustfield)

```



Les entrées/sorties (2)



Fichiers :

Fichier de définitions : `#include <fstream>`

Flot d'**entrée** (similaire à `cin`) : `ifstream`

Flot de **sortie** (similaire à `cout`) : `ofstream`

Création : `type_flot nom_de_flot;`

Lien (ouverture) : `flot.open("fichier");`

ouverture en binaire :

```

pour lecture : ifstream flot("fichier", ios::in|ios::binary);
pour écriture : ofstream flot("fichier", ios::out|ios::binary);

```

Utilisation : comme `cin` et `cout` :

```

flot << expression << ... ;
flot >> variable_lue >> ... ;

```

Test de fin de fichier : `flot.eof()`

Fermeture du fichier : `flot.close()`

Test d'échec de lecture/écriture/ouverture sur le flot : `flot.fail()`



Les entrées/sorties



Clavier / Terminal : `cin` / `cout` et `cerr`

Fichier de définitions : `#include <iostream>`

Utilisation :

écriture : `cout << expr1 << expr2 << ... ;`

lecture : `cin >> var1 >> var2 >> ... ;`

Saut à la ligne : `endl`

Lecture d'une ligne entière : `getline(cin, string);`

Formatage :

Manipulateurs		Options	
<code>#include <iomanip></code>		<code>setf(ios::option)</code>	
<code>cout << manip << expr << ...</code>		<code>unsetf(ios::option)</code>	
<code>dec, oct, hex</code> <code>setprecision(int)</code>	changement de base nombre de chiffres à afficher	<code>ios::left</code>	alignement à gauche
<code>setw(int)</code>	largeur de colonne	<code>ios::showbase</code>	afficher la base
<code>setfill(char)</code>	nombre de caractères à lire caractère utilisé dans l'alignement	<code>ios::showpoint</code>	afficher toujours la virgule
<code>cin >> ws</code>	saute les blancs	<code>ios::fixed</code>	notation fixe
		<code>ios::scientific</code>	notation scientifique



Flots dans des chaînes



Supposons que l'on écrive un programme avec une **interface graphique** que nous gérons nous-même dans le programme. On ne voudra évidemment pas envoyer nos messages sur `cout` mais dans notre interface.

Comment faire pour avoir accès aux messages que l'on envoie sur `cout` ?

☞ avec des **flots dans des chaînes de caractères** :

les `stringstream`

Ils se manipulent comme les flots précédemment vus sauf qu'ils sont associés à des `string`

Pour pouvoir les utiliser il faut inclure le fichier de définition

`sstream` :

```
#include <sstream>
```

Pour **écrire** dans un tel flot, utiliser `ostringstream`.

Par contre, pour **lire** depuis une chaîne (c.-à-d. simuler avec une chaîne de caractères l'entrée standard), utiliser `istringstream`.



Exemple :

```
#include <sstream>

void mon_interface(string a_afficher);

ostringstream message;
int nb_carottes;

message << "Vous avez " << nb_carottes << " legumes.";

mon_interface(message.str());
```

Notez l'emploi de la fonction spécifique `str()` pour effectivement transformer le contenu du **flot** en **string**.

Ce que j'ai appris aujourd'hui

- ▶ Entrées/sorties :
 - ▶ donner de la persistance aux données manipulées par un programme (utilisation des fichiers) ;
 - ▶ à formater les résultats en sortie



Un exemple avec `istringstream` : convertir une chaîne de caractères en entiers :

```
int str2int(string const& str)
{
    int extracted(0);
    istringstream is(str);
    is >> extracted;
    return extracted;
}
```