

MATHIEU NEBRA  
MATTHIEU SCHALLER

# PROGRAMMEZ AVEC LE LANGAGE C++

TOUTE LA PUISSANCE DU LANGAGE C++  
EXPLIQUÉE AUX DÉBUTANTS



Issu du célèbre  
**Site du Zéro**  
[www.siteduzero.com](http://www.siteduzero.com)



[www.siteduzero.com](http://www.siteduzero.com)



Sauf mention contraire, le contenu de cet ouvrage est publié sous la licence :  
Creative Commons BY-NC-SA 2.0

La copie de cet ouvrage est autorisée sous réserve du respect des conditions de la licence  
Texte complet de la licence disponible sur : <http://creativecommons.org/licenses/by-nc-sa/2.0/fr/>

Simple IT 2011 - ISBN : 978-2-9535278-5-8

# Chapitre 37

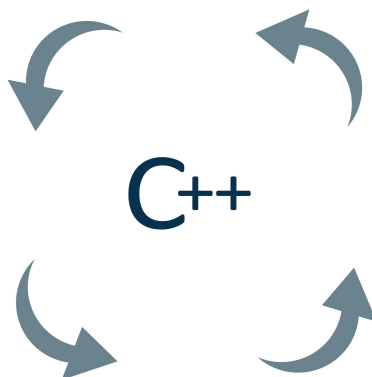
## Utiliser les itérateurs sur les flux

Difficulté : 

Si l'on retourne au tout début de votre apprentissage du C++, on découvre que le premier objet que vous avez manipulé (sans le savoir!) est l'objet `cout`. Avec son acolyte habituel `cin`, vous avez pu interagir avec les utilisateurs de la console. Mais que savez-vous réellement sur ces objets? Que peut-on faire d'autre qu'utiliser les chevrons et la fonction `getline()`? Il est enfin temps d'aller plus loin et de découvrir la vraie nature des flux.

Dans ce chapitre, nous allons apprendre à utiliser des itérateurs sur les flux. À nouveau, cela va nous ouvrir grand les portes du monde des algorithmes et nous allons pouvoir utiliser tout ce que nous savons déjà sur les flux, par exemple pour simplifier l'écriture d'un `vector` dans la console ou dans un fichier.

Finalement, nous verrons qu'il existe aussi des flux sur les `string`. Encore une nouvelle découverte sur ce type vraiment particulier!



## Les itérateurs de flux

Au chapitre sur les itérateurs, je vous avais présenté deux catégories d'itérateurs :

- les *random access iterators*, qui permettent d'accéder directement à n'importe quelle case d'un tableau;
- les *bidirectional iterators* qui, eux, ne peuvent avancer et reculer que d'une case à la fois sans pouvoir aller directement à une position donnée.

En réalité, j'avais simplifié les choses. Il existe encore deux autres catégories d'itérateurs. Et si je vous en parle, c'est que nous allons en avoir besoin dans ce chapitre.

Une des propriétés importantes des flux est qu'ils ne peuvent être lus et modifiés que dans un seul sens. On ne peut pas lire un fichier à l'envers ou écrire une phrase dans la console en sens inverse. Les itérateurs sur les flux ont donc la propriété de ne pouvoir qu'avancer. Par conséquent, ils ne possèdent que l'opérateur ++ et pas le -- comme ceux que nous avons rencontrés jusque-là.

En plus de cette importante restriction, les itérateurs sur les flux entrants (`cin`, les fichiers `ifstream`,...) ne peuvent pas modifier les éléments sur lesquels ils pointent. C'est normal : on ne peut que lire dans `cin`, pas y écrire. Les itérateurs respectent cette logique. De même, les itérateurs sur les flux sortants (`cout`, les fichiers `ofstream`,...) ne peuvent pas lire la valeur des éléments, seulement y écrire.

### Déclarer un itérateur sur un flux sortant

Comme toujours, la première question qui se pose est celle du fichier d'en-tête contenant ce que l'on cherche. Les itérateurs de flux (et plus généralement tous les itérateurs) sont déclarés dans l'en-tête `iterator` de la SL. Pour une fois, c'est un nom facile à retenir !

Déclarons pour commencer un itérateur sur le flux sortant `cout`.

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

int main()
{
    ostream_iterator<double> it(cout);

    return 0;
}
```

Ce code déclare un itérateur sur le flux sortant `cout`, permettant d'écrire des `double`. Vous remarquerez deux choses différentes de ce qu'on a vu jusqu'à maintenant :

- on n'utilise pas la syntaxe `conteneur::iterator`;
- il faut indiquer entre les chevrons le type des éléments envoyés dans le flux.

Mais, à part cela, tout fonctionne comme d'habitude. On peut utiliser l'itérateur *via* son opérateur \*, ce qui aura pour effet d'écrire dans la console :

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

int main()
{
    ostream_iterator<double> it(cout);
    *it = 3.14;
    *it = 2.71;

    return 0;
}
```

Testez ce code, vous devriez obtenir ceci :

```
3.142.71
```

Les deux nombres ont bien été écrits. Le seul problème, c'est que nous n'avons pas inséré d'espace entre eux. C'est là qu'intervient le deuxième argument du constructeur de l'itérateur. On peut spécifier ce qu'on appelle un délimiteur, c'est-à-dire le ou les symboles qui seront insérés entre chaque écriture faite *via* l'opérateur \*. Essayons de mettre une virgule et un espace pour voir.

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

int main()
{
    ostream_iterator<double> it(cout, ", ");
    *it = 3.14;
    *it = 2.71;

    return 0;
}
```

Ce qui donne :

```
3.14, 2.71,
```

Parfait ! Juste ce que l'on voulait.



Pour obtenir un retour à la ligne entre chaque écriture, il faut spécifier le délimiteur `"\n"`.

Je vous propose, comme exercice, de reprendre le tout premier code C++, le fameux « Hello World! ». Essayez de le réécrire en utilisant un itérateur de flux sur `cout`, permettant d'écrire des chaînes de caractères séparées par des espaces.

## Déclarer un itérateur sur un flux entrant

Les itérateurs sur les flux entrants s'utilisent exactement de la même manière. On déclare l'itérateur en spécifiant entre les chevrons le type d'objet et en passant en argument du constructeur le flux à lire. Pour lire depuis un fichier, on aurait ainsi la déclaration suivante :

```
ifstream fichier("C:/Nanoc/data.txt");
istream_iterator<double> it(fichier);    //Un itérateur
↪ lisant des doubles depuis le fichier
```

La différence avec les `ostream_iterator` est qu'il faut explicitement les faire avancer après chaque lecture. Et bien sûr, cela se fait grâce à l'opérateur `++`.

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

int main()
{
    ifstream fichier("C:/Nanoc/data.txt");
    istream_iterator<double> it(fichier);

    double a,b;
    a = *it;    //On lit le premier nombre du fichier
    ++it;      //On passe au suivant
    b = *it;    //On lit le deuxième nombre

    return 0;
}
```

Bref, ce n'est pas très complexe. Il faut cependant savoir s'arrêter à la fin du fichier. Heureusement, les concepteurs de la SL ont pensé à tout ! Pour les conteneurs, il y avait la méthode `end()` qui nous renvoyait un itérateur indiquant la fin du conteneur. Il existe un mécanisme similaire ici. Si l'on déclare un `istream_iterator` sans lui passer d'argument à la construction, alors il pointe directement vers ce qu'on appelle un *end-of-stream iterator*, une sorte de signal de fin de flux. On peut ainsi utiliser ce signal comme limite pour la lecture. Pour lire un fichier du début à la fin et l'afficher dans la console on procéder ainsi :

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

```
using namespace std;

int main()
{
    ifstream fichier("data.txt");
    istream_iterator<double> it(fichier); //Un itérateur sur le fichier
    istream_iterator<double> end;        //Le signal de fin

    while(it != end) //Tant qu'on a pas atteint la fin
    {
        cout << *it << endl; //On lit
        ++it;                //Et on avance
    }
    return 0;
}
```

Tiens, cela me donne une idée. Plutôt que d'utiliser directement `cout` pour afficher les valeurs lues, essayez de réécrire ce code avec un itérateur sur un flux sortant !

## Le retour des algorithmes

Bon, jusque là, utiliser ces nouveaux itérateurs n'a rien amené de vraiment intéressant. À part pour frimer dans les discussions de programmeurs, tout cela est un peu inutile. C'est parce que nous n'avons pas encore appris à utiliser les algorithmes ! Comme nous avons des itérateurs, il ne nous reste qu'à les utiliser à bon escient !



Tous les algorithmes ne sont pas utilisables. Par exemple, ceux qui nécessitent des accès aléatoires comme `sort` ne peuvent pas être employés.

### L'algorithme copy

Commençons avec l'algorithme qui est très certainement le plus utilisé dans ce contexte : `copy()`. Il arrive très souvent que l'on doive lire des valeurs depuis un fichier pour les stocker dans un `vector` par exemple. Il s'agit simplement de lire les éléments depuis le flux et de les insérer dans le tableau créé au préalable.

La fonction `copy()` reçoit trois arguments. Les deux premiers correspondent au début et à la fin de la zone à lire et le troisième est un itérateur sur le début de la zone à écrire.

Pour copier depuis un fichier vers un `vector`, on ferait donc ceci :

```
#include <algorithm>
#include <vector>
#include <iterator>
```

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

int main()
{
    vector<int> tab(100,0);
    ifstream fichier("C:/Nanoc/data.txt");
    istream_iterator<int> it(fichier);
    istream_iterator<int> fin;
    copy(it, fin, tab.begin()); //On copie le contenu du fichier
    ↪ du debut à la fin dans le vector

    return 0;
}
```



Il faut absolument que votre vector soit assez grand pour contenir tous les nombres lus.

On peut bien sûr utiliser `copy()` pour écrire dans un fichier ou dans la console. On peut donc reprendre les exemples des chapitres précédents et remplacer la boucle d'affichage des valeurs par un appel à `copy()`, comme ceci :

```
int main()
{
    srand(time(0));
    vector<int> tab(100,-1); //Un tableau de 100 cases

    //On génère les nombres aléatoires
    generate(tab.begin(), tab.end(), Generer());
    //On trie le tableau
    sort(tab.begin(), tab.end());
    //Et on l'affiche
    copy(tab.begin(), tab.end(), ostream_iterator<int>(cout, "\n"));
    return 0;
}
```

C'est simple et efficace. On ne s'embête plus avec des boucles. Tout est caché derrière des noms de fonctions qui décrivent bien ce qui se passe. Le code est ainsi devenu plus lisible et compréhensible, et il n'a bien sûr rien perdu en efficacité.

## Le problème de la taille

Lorsqu'on lit des données dans un fichier pour les insérer dans un tableau, il y a un problème qui survient assez souvent : celui de la taille à donner au tableau. On ne sait pas forcément, avant de lire le fichier, combien de valeurs il contient. Et ce serait dommage de le lire deux fois simplement pour obtenir cette information ! Il serait judicieux

d'avoir des itérateurs un peu plus évolués permettant de faire grandir le `vector`, la `list` ou la `deque` à chaque lecture. C'est ce qu'on appelle des `back_inserters`. Pour déclarer un de ces itérateurs sur un `vector`, on écrit ceci :

```
vector<string> tableau; //Un tableau vide de chaînes de caractères
back_inserter it(tableau); //Un itérateur capable de faire grandir le tableau
```

Cet itérateur s'utilise alors comme n'importe quel autre itérateur. La seule différence se ressent au moment de l'appel à l'opérateur `*`. Au lieu de modifier une case, l'itérateur en ajoute une nouvelle à la fin du tableau. Nous pouvons donc reprendre le code qui copiait un fichier dans un tableau pour l'améliorer :

```
#include <algorithm>
#include <vector>
#include <iterator>
#include <fstream>
using namespace std;

int main()
{
    vector<int> tab; //Un tableau vide
    ifstream fichier("C:/Nanoc/data.txt");
    istream_iterator<int> it(fichier);
    istream_iterator<int> fin;
    //L'algorithme ajoute les cases nécessaires au tableau
    copy(it, fin, back_inserter(tab));

    return 0;
}
```

Pour vous exercer, je vous propose d'essayer de refaire le tout premier TP : le tirage au sort d'un mot dans le dictionnaire devrait maintenant en être grandement simplifié!

Voyons rapidement quelques autres algorithmes utilisables avec des fichiers.

## D'autres algorithmes

Au chapitre précédent, nous avons vu l'algorithme `count()` qui permettait de compter les occurrences d'une valeur dans un conteneur. On peut aussi l'utiliser pour compter dans les fichiers, ou employer `min_element()` ou `max_element()` pour chercher la plus petite ou la plus grande des valeurs contenues. Cela ne devrait pas être trop difficile à utiliser. Voici par exemple les lignes permettant de trouver le minimum des valeurs dans un fichier :

```
ifstream fichier("C:/Nanoc/data.txt");
cout << *min_element(istream_iterator<int>(fichier), istream_iterator<int>())
<< << endl;
```

## Encore un retour sur les strings !

Les flux sont un concept tellement puissant que les créateurs de la SL ont décidé de l'appliquer également aux chaînes de caractères. Jusqu'à maintenant, vous avez appris à modifier les `string` *via* l'opérateur `[]`, mais vous n'avez jamais vu comment insérer un nombre dans une chaîne de caractères. Les flux sur les chaînes de caractères permettent d'écrire un `double` ou n'importe quel autre type dans un `string` sous forme de texte. Les flux sur les `string` s'appellent `ostream` et `istream` selon qu'on lit la chaîne ou qu'on y écrit. Pour créer de tels objets, rien de plus simple : il suffit de passer en argument au constructeur la chaîne sur laquelle le flux va travailler. On peut alors récupérer la chaîne de caractère en utilisant la méthode `str()`. Auparavant, il faut, comme toujours, inclure le bon fichier d'en-tête : `sstream`.

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

int main()
{
    ostream flux; //Un flux permettant d'écrire dans une chaîne

    flux << "Salut les"; //On écrit dans le flux grâce à l'opérateur <<
    flux << " zeros";
    flux << " !";

    string const chaine = flux.str(); //On récupère la chaîne

    cout << chaine << endl; //Affiche 'Salut les zeros !'
    return 0;
}
```

Une fois que le flux est déclaré, on utilise simplement les chevrons pour écrire dans la chaîne. Si vous souhaitez insérer un nombre dans un `string`, il n'y a aucune différence. Tout se passe comme si on utilisait `cout` :

```
string chaine("Le nombre pi vaut: ");
double const pi(3.1415);

ostream flux;
flux << chaine;
flux << pi;

cout << flux.str() << endl;
```

C'est à la fois très simple et très puissant. On combine la simplicité d'utilisation des `string` à la liberté sur l'écriture des types que donne l'utilisation des flux. C'est assez

magique, je trouve! C'est cette technique que l'on utilise à chaque fois l'on cherche à convertir un nombre en une chaîne de caractères. Souvenez-vous de cela.

On peut bien sûr faire cela dans l'autre sens, c'est-à-dire extraire des nombres depuis une chaîne de caractères. Il faudra alors utiliser les flux de lecture `istream`. Mais vous êtes doués maintenant, vous pouvez essayer tous seuls. ;-)

Enfin, sachez que l'on peut tout à fait utiliser les itérateurs sur les `ostream` et `istream` comme sur n'importe quel autre flux. Vous pouvez ainsi coupler la puissance des itérateurs et algorithmes à tout ce que vous savez sur les `string`. Mais personne ne procède ainsi : la solution correcte est présentée au prochain chapitre!

## En résumé

- Il existe des itérateurs sur les flux.
- Ces itérateurs ne peuvent qu'avancer. Ils ne possèdent donc que l'opérateur `++` et l'opérateur `*`.
- On peut utiliser ces itérateurs avec les algorithmes pour simplifier nos programmes.
- On peut écrire et lire dans les chaînes de caractères grâce aux `istream` et `ostream`. Cela permet de combiner la puissance des flux à la simplicité des `string`.
- On utilise les `stringstream` pour convertir des nombres en chaîne et vice-versa.