> **Ce cours est une mise à jour du cours C++ de OpenClassRoom pour le mettre à jour pour le C++11/14. Le cours d'origine est consultable sur la page suivante : Programmez avec le langage C++, par Mathieu Nebra et Matthieu Schaller. Ce cours est sous licence CC-BY-NC-SA.**

Retourner au sommaire principal

Aller plus loin avec la SL

Dès le premier chapitre sur la SL, je vous ai prévenus que le sujet était très vaste et qu'il serait difficile d'en faire le tour. Nous avons étudié les principaux éléments repris du langage C puis nous nous sommes concentrés sur la STL et sur les flux. Il faut quand même que je vous présente les autres possibilités de la bibliothèque standard.

Ce chapitre présente trois domaines différents où la SL va nous aider à améliorer nos programmes. Pour commencer, nous allons reparler des chaînes de caractères et voir comment, là aussi, utiliser des itérateurs. Puis, nous reviendrons sur les tableaux statiques. Nous les avions un peu abandonnés au profit des autres conteneurs mais il est temps d'en reparler et d'utiliser nos nouveaux meilleurs amis : les itérateurs bien sûr ! Enfin, la troisième partie sera assez différente puisque nous y découvrirons quelque chose de complètement nouveau : les outils dédiés au calcul scientifique. Le C++ est en effet très utilisé par les chercheurs en tous genres pour faire des simulations, que ce soit sur un ordinateur classique ou sur un super-calculateur.

Plus loin avec les strings

Bon, les string, on commence à connaître depuis le temps ! Cependant, vous êtes encore loin de tout savoir. Et puisque nous parlons de la STL depuis quelques chapitres, vous vous doutez probablement que nous allons avoir affaire à des itérateurs. Nous avions vu que les string se comportaient comme des tableaux grâce à la surcharge de l'opérateur [].

Mais ce n'est pas leur seul point commun avec les vector, ils possèdent aussi les méthodes begin() et end() renvoyant un itérateur sur le début et la fin de la chaîne.

```
string chaine("Salut les zeros!"); // Une chaîne

string::iterator it = chaine.begin(); // Un itérateur sur le début
```

Bref, que des choses déjà bien connues. Je vous avais présenté, dans le chapitre d'introduction à la SL, les fonctions toupper() et tolower() qui permettent de convertir une minuscule en majuscule et vice-versa. Il est possible d'utiliser ces fonctions dans les algorithmes. Ici, c'est bien sûr l'algorithme transform() qu'il faut utiliser. Il parcourt la chaîne, applique la fonction sur chaque élément et écrit le résultat au même endroit.

```
#include <iostream>
#include <string>
#include <algorithm>
#include <cctype>
using namespace std;

class Convertir
{
public:
    char operator()(char c) const
    {
        return toupper(c);
    }
};

int main()
{
    string chaine("Salut les zeros !");
    transform(chaine.begin(), chaine.end(), chaine.begin(),
Convertir());
    cout << chaine << endl;
    return 0;
}
```

Ce code affiche donc le résultat suivant :

```
SALUT LES ZEROS !
```

Il n'y pas grand chose de plus à dire sur le sujet. En fait, vous savez déjà presque tout. Sachez seulement que les string possèdent aussi des méthodes insert() et erase() qui fonctionnent de manière similaire à celles de `vector`. Vous pouvez, grâce à elles, insérer et supprimer des lettres au milieu d'une chaîne.

Passons maintenant à une autre vieille connaissance : le tableau statique.

Manipuler les tableaux statiques

Commençons par un bref rappel. Un tableau statique est un tableau dont la taille ne peut pas varier. Il se déclare en utilisant les crochets [] entre lesquels on spécifie le nombre de cases désirées. Par exemple, pour un tableau de 10 entiers nommé tab, on aurait la déclaration suivante :

```
int tab[10];
```

Et on peut bien sûr créer des tableaux de n'importe quel type : double, string ou même Personnage (la fameuse classe que nous avions créée lorsque nous avions découvert la POO). Il y a une seule obligation : les objets doivent posséder un constructeur par défaut.

Comme ces tableaux ne sont pas des objets (comme vector ou deque), ils ne possèdent aucune méthode. Il n'est donc pas possible, par exemple, de connaître leur taille. Il faut toujours stocker la taille du tableau dans une variable supplémentaire. Mais cela, vous le saviez déjà.

Les itérateurs

Ces tableaux ont beau ne pas être des objets, on aimerait quand même bien pouvoir utiliser des itérateurs puisque cela nous ouvrirait la porte des algorithmes. Cependant, il n'existe pas d'itérateur spécifique et bien sûr pas de méthode begin() ou end(). Je vous avais dit que les itérateurs étaient la « version objet » des pointeurs, tout comme vector est la «

version objet » des tableaux statiques. Et là, je vous sens frémir. Effectivement, nous allons utiliser des pointeurs comme itérateurs sur ces tableaux. Que demande-t-on à un itérateur ? Principalement de pouvoir avancer, reculer et de nous renvoyer la valeur pointée grâce à l'opérateur *. Ce sont justement des opérations qui existent pour les pointeurs. Il n'y a donc plus qu'à se jeter à l'eau.

Dans la plupart des cas, on a besoin d'un itérateur sur le premier élément. Dans notre nouveau langage, on dirait qu'on a besoin de l'adresse de la première case. On pourrait donc écrire ceci pour notre itérateur :

```
int tab[10];  //Un tableau de 10 entiers

int* it(&tab[0]); //On récupère l'adresse de la première
case
```

Ah, je vois que cela vous fait peur. Rappelez-vous que l'esperluette (&) renvoie l'adresse d'une variable, ici la première case du tableau. On initialise ensuite notre pointeur d'entiers à cette valeur. Nous avons donc un itérateur.

Heureusement, il existe une manière plus simple d'écrire cela. Il faut savoir que tab est lui aussi, en réalité, un pointeur (on n'avait jamais eu besoin de cette information jusqu'ici et j'espère que vous ne m'en voudrez pas de ne pas l'avoir dit plus tôt) ! Ce pointeur pointe sur la première case, justement ce qu'il nous faut. On écrit donc généralement plutôt ceci :

```
int tab[10];  //Un tableau de 10 entiers

int* it(tab); //Un itérateur sur ce tableau
```

L'itérateur de fin

Comme toujours, on a besoin de spécifier la fin du tableau via un deuxième itérateur. La solution est de réfléchir aux cases qui sont accessibles. Dans l'exemple précédent, it pointe sur la première case.

Donc, it+1 pointera sur la deuxième, it+2 sur la troisième, etc. Un itérateur pointant sur la première case en dehors du tableau sera, en suivant cette logique, it+10. Si on itère de it à it+10 exclu, on aura parcouru toutes les cases du tableau. En règle générale, on stocke la taille du tableau dans une variable et on écrirait le code suivant pour obtenir le début et la fin d'un tableau :

```
int const taille(10);
int tab[taille]; //Un tableau de 10 entiers

int* debut(tab); //Un itérateur sur le début
int* fin(tab+taille); //Un itérateur sur la fin
```

Nous avons ainsi un équivalent de begin() et un équivalent de end(). Il ne nous reste plus qu'à utiliser les algorithmes. Mais cela, vous savez déjà le faire. En tout cas je l'espère... Bon, je vous donne quand même un exemple. Pour trier un tableau de nombres, on peut écrire ceci :

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

int main()
{
    int const taille(1000);
    double tableau[taille];     //On déclare un tableau

    //Remplissage du tableau…

    double* debut(tableau);     //Les deux itérateurs
    double* fin(tableau+taille);

    sort(debut, fin);           //Et on trie

    return 0;
}
```

> Il est possible d'accéder directement à n'importe quel élément du tableau grâce à cette technique. Les pointeurs se comportent donc comme des random access iterators.

Je crois que vous auriez trouvé par vous-mêmes. Vous êtes devenu des pros de la STL depuis le temps. 

Faire du calcul scientifique

Dans tous les programmes scientifiques, il y a, vous vous en doutez, beaucoup de calculs. Ce sont des programmes qui manipulent énormément de nombres en tous genres. Vous connaissez déjà les int et les double ainsi que les fractions mais, dans certains projets, on utilise également des nombres complexes (si vous ne savez pas ce que c'est, ce n'est pas grave, vous pouvez simplement sauter cette section pour attaquer celle parlant des valarray).

Les nombres complexes

Comme c'est une brique de base, la SL se devait de fournir un moyen de manipuler ces nombres. C'est pour cela qu'il existe l'en-tête complex, dans lequel se trouve la définition de la classe du même nom. Pour déclarer un nombre complexe 2+3i et l'afficher, on utilise le code suivant :

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

int main()
{
    complex<double> c(2,3);
    cout << c << endl;
    return 0;
}
```

Ce code produit le résultat suivant :

```
(2., 3.)
```

Il faut spécifier le type des nombres à utiliser pour représenter la partie réelle et la partie imaginaire des nombres complexes. Il est très rare d'utiliser pour cela autre chose que des double, mais on ne sait jamais... À partir de là, on peut utiliser les opérateurs usuels pour faire des additions, multiplications, divisions, etc. avec ces nombres. La force de la surcharge des opérateurs est à nouveau visible. En plus des opérations arithmétiques de base, il existe aussi quelques fonctions mathématiques bien pratiques comme la racine carrée ou les fonctions trigonométriques.

```
complex<double> a(1., 2.), b(-2, 4), c;
c = sqrt(a+b);

a = cos(c/b) + sin(b/c);
```

Bref, tout ce qui est nécessaire pour faire des maths un peu poussées. Enfin, il existe des fonctions spécifiques aux nombres complexes comme la norme ou le conjugué. Vous trouverez une liste complète des possibilités dans votre documentation préférée.

```
complex<double> a(3,4);
cout << norm(conj(a)) << endl; //Affiche '5'
```

Toutes les fonctions ont leur nom habituel en maths, il n'y a donc aucune difficulté. Il faut juste savoir qu'elles existent, ce qui est chose faite maintenant. 

Les valarray

L'autre élément que l'on retrouve dans beaucoup de programmes de simulation est bien sûr le tableau de nombres. Vous en connaissez déjà beaucoup mais il y a une forme particulièrement bien adaptée aux calculs : les valarray. Ils sont plus restrictifs que les vector dans le sens où l'on ne peut pas facilement ajouter des cases à la fin mais, comme ce n'est pas une opération très courante, ce n'est pas un problème. La grande force des valarray est la possibilité d'effectuer des opérations mathématiques directement avec l'ensemble du tableau. On peut par

exemple calculer la somme de deux tableaux élément par élément simplement en utilisant l'opérateur +.

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

int main()
{
    valarray<int> a(10, 5);  //5 éléments valant 10
    valarray<int> b(8, 5);   //5 éléments valant 8

    valarray<int> c = a + b;  //Chaque élément de c vaut 18
    return 0;
}
```

On n'a ainsi pas besoin d'écrire des boucles pour effectuer ces opérations de base. Remarquez au passage que le constructeur des valarray prend ses arguments dans l'ordre inverse des vector. Il faut d'abord indiquer la valeur que l'on souhaite puis le nombre de cases. Faites attention, on se trompe souvent !

Tous les opérateurs usuels sont surchargés de sorte qu'ils travaillent sur tous les éléments séparément. Par exemple, l'opérateur == compare un par un tous les éléments du tableau et renvoie un tableau de bool. On peut alors savoir quels sont les éléments identiques et ceux qui sont différents en lisant la case correspondante de ce tableau.

Enfin, on peut aussi utiliser la méthode apply() pour appliquer un foncteur aux éléments du tableau. On s'économise ainsi l'utilisation d'un algorithme et des itérateurs. C'est un confort de notation supplémentaire. Pour calculer le cosinus de tous les éléments d'un valarray, on écrirait ceci :

```
#include<valarray>
#include<cmath>
using namespace std;

class Cosinus   //Un foncteur pour le calcul du cosinus
{
public:
```

```
    double operator()(double x) const
    {
        return cos(x);
    }
};

int main()
{
    valarray<double> a(10);  //10 éléments

    //Remplissage du tableau…

    a.apply(Cosinus);

    //Chaque case contient maintenant le cosinus de son
ancienne valeur

    return 0;
}
```

À nouveau, faites un tour dans votre documentation favorite pour découvrir toutes les fonctionnalités de ces tableaux. Ils sont vraiment pratiques.

> La SL ne propose pas de fonctionnalités pour faire du calcul matriciel, même si c'est très courant. On doit alors se tourner vers des bibliothèques externes comme lapack, MTL ou blas.

En résumé

- Les string proposent eux aussi des itérateurs. On peut donc utiliser les algorithmes également sur les chaînes de caractères.
- Les tableaux statiques ne possèdent pas d'itérateurs mais on utilise pour les remplacer les pointeurs.
- La SL propose quelques outils pour le calcul scientifique, notamment une classe de nombres complexes et des tableaux optimisés pour effectuer des opérations mathématiques.

Retourner au sommaire principal

Cours, C++