Itérateurs divers et *inserters* (M.M. mars 1998, rév. avril 1999, fév. 2000)

Voir aussi Stroustrup : 555-558, Prata : 772-778 (il manque `back_inserter, front_inserter et inserter`), Lippman : p.595-601

Essayons d'écrire la fonction générique `copy` :

```
template <typename In, typename Out>
Out copy(In p_debut, In p_fin, Out p_dest)
    {
    while (p_debut != p_fin)
        *p_dest++= *p_debut++;

    return p_dest;
    }
```

On aurait pu aussi écrire la version suivante qui est probablement plus performante avec des itérateurs :

```
template <typename In, typename Out>
Out copy(In p_debut, In p_fin, Out p_dest)
    {
    while (p_debut != p_fin)
        {
        *p_dest= *p_debut;
        ++p_dest;
        ++p_debut;
        }

    return p_dest;
    }
```

Rappelons qu'on utilise cette fonction ainsi :

```
list<int> liste;
int vec[NB_ELEMENTS];
...
copy(liste.begin(), liste.end(), vec); // Il faut que le vecteur soit assez grand
...
copy(vec, vec+nb, liste.begin());   // La liste doit avoir au moins nb éléments
```

On a un danger de débordement dans les deux cas, car `copy` ne teste pas si la fin de la destination est atteinte avant de copier ou de faire avancer l'itérateur. Il faut que des éléments existent dans la collection de destination et leur valeur sera remplacée par les données à copier.

Serait-il possible d'écrire plutôt une fonction similaire mais qui évite le débordement ? Non, pas en général. Par exemple, il n'y a aucune façon de savoir si on déborde d'un vecteur de base; il n'y a aucune façon non plus de changer la taille de ceux-ci. Comme les fonctions de `<algorithm>` doivent fonctionner avec les vecteurs de base, il est normal qu'elles requièrent d'allouer l'espace nécessaire et de faire les vérifications de débordement avant l'appel.

Par contre, lorsqu'on utilise les collections de la librairie, on sait qu'il y a toujours possibilité d'un `insert` ou d'un `push_back` qui éviterait tout débordement et toute pré-allocation d'espace. Faudrait-il des fonctions de `<algorithm>` spéciales pour ces cas-là ? Non, car on peut utiliser C++ pour adapter les fonctions en passant comme itérateur de sortie un objet qui agira comme un itérateur mais qui fera des `insert`s ou des `push_back`. Par exemple, supposons cette fonction :

```
template <typename C>
ClAjouterALaFin<C>& AjouterALaFin(C& p_collection)
    {
    return ClAjouterALaFin<C>(p_collection);
    }
```

Et la classe suivante dont on crée facilement des instances grâce à la fonction :

```
template <typename C>
class ClAjouterALaFin
    {
    private :
    typedef typename C::value_type T;
    C& m_collection;

    public :
    ClAjouterALaFin(C& p_collection)
        : m_collection(p_collection)
        {}

    T operator=(const T& p_val)
        {
        m_collection.push_back(p_val);
        return p_val;
        }

    ClAjouterALaFin& operator*()          // Ces trois fonctions
        { return *this; }
    ClAjouterALaFin& operator++()         // ne font rien sur
        { return *this; }
    ClAjouterALaFin& operator++(int)      // l'objet
        { return *this; }
    };
```

Par la suite, on fera par exemple :

```
copy(vec, vec+nb, AjouterALaFin(liste)); // Ou copy(..., ClAjouterALaFin<list<int> >());
```

Que se passe-t-il dans la fonction `copy` (deuxième version) ?

- `p_dest` est un objet de type `ClAjouterALaFin` conservant la collection `liste` dans `m_collection`;
- la ligne `*p_dest= *p_debut;` équivaut à `p_dest= *p_debut`, car `operator*` ne renvoie que l'objet de base;
- `p_dest= *debut` devient un `m_collection.push_back(*p_debut)` donc `liste.push_back(*p_debut);`
- le `++p_dest;` ne fait rien…

Donc c'est ce qu'on voulait ! Vous vérifierez que la première version de `copy` fonctionne aussi…

Il y a évidemment l'équivalent de `ClAjouterALaFin/AjouterALaFin` dans la librairie. Mais il y a aussi ce qu'il faut pour ajouter au début, ou à une position spécifique. Les fonctions sont `back_inserter, front_inserter` et `inserter`. Par exemple :

```
copy(vec, vec+nb, back_inserter(liste));
copy(vec, vec+nb, front_inserter(liste));
liste<int>::iterator it= liste.begin();
copy(vec, vec+nb, inserter(liste, ++it));  // Ajoute après le premier
```

Évidemment on peut utiliser ces «inserters» avec toutes les fonctions de la librairie qui font des copies de données dans une collection, pas seulement avec `copy`…

Itérateurs dans les streams

Dans le même esprit, ne pourrait-on pas utiliser `copy` (et les autres fonctions) pour ajouter ou extraire des données dans les streams ? Il faut d'abord remarquer que, pour les fichiers séquentiels ordinaires de texte, il n'est pas possible de reculer simplement. Par conséquent, une opération de style -- n'est pas possible facilement. Par contre, le ++ est quasi automatique puisque la position dans un fichier change après une lecture ou une écriture…

Les fonctions de `<algorithm>` n'utilisant généralement pas `--`, on peut facilement faire ceci :

```
template <typename T>  // T est le type des données dans le stream
class ClIterateurSortie
    {
    private :
    ostream& m_os;

    public :
    ClIterateurSortie(ostream& p_os)
        : m_os(p_os)
        {}

    ClIterateurSortie& operator=(const T& p_val)
        {
        m_os << p_val << ' ';
        return *this;
        }

    ClIterateurSortie& operator*()
        {return *this;}
    ClIterateurSortie& operator++()
        {return *this;}
    ClIterateurSortie& operator++(int)
        {return *this;}
    };
```

Ce qui permettra :

```
copy(vec, vec+nb, ClIterateurSortie<int>(cout));
```

Encore une fois, on obtient ce qu'on voudrait, car le `*dest` et le `++dest` de la fonction `copy` sont «ignorés» et, donc le `dest= *debut` correspondant devient une écriture avec <<…

La vraie classe de la librairie, `ostream_iterator`, est plus complète de façon à permettre le choix du séparateur d'écriture (ici on séparait toujours par un espace). De même, la classe pour les itérateurs en lecture, `istream_iterator`, doit permettre le test de fin de fichiers, etc. Voici des exemples d'utilisation :

```
ifstream ficNoms("nomsProf.txt");

vector<string> tabNoms(istream_iterator<string>(ficNoms), istream_iterator<string>());
// Au lieu du constructeur, on aurait pu utiliser copy avec un back_inserter...

sort(tabNoms.begin(), tabNoms.end());

copy(tabNoms.begin(), tabNoms.end(), ostream_iterator<string>(cout, "...\n"));
```

Cette portion de programme charge d'abord les noms dans un vector. Les noms sont lus avec >> dans le fichier *ficNoms*. Le `istream_iterator<string>()` génère une valeur sentinelle pour le test de fins de fichiers, il y a donc des opérateurs `==` et `!=` en conséquence dans la classe `istream_iterator`. Une fois triés, les éléments du vector sont écrits à l'écran (séparés par ... et un changement de ligne).

Évidemment tout ça devient plus intéressant si on a des fichiers de structures (d'enregistrements) pour lesquels on a défini les opérateurs <<, >>, etc. Par exemple (un programme presque complet) :

```
ifstream ficClients("clients.données");
vector<TypeClient> tabMauvaisPayeurs;

remove_copy_if(istream_iterator<TypeClient>(ficClients), istream_iterator<TypeClient>(),
               back_inserter(tabMauvaisPayeurs),
               not1(ClSoldeSuperieurA(SOLDE_MIN_MAUVAIS_PAYEUR)));

for_each(tabMauvaisPayeurs.begin(), tabMauvaisPayeurs.end(), AfficherClient);
```

La classe de fonction objet `ClSoldeSuperieurA` et la fonction `AfficherClient` sont laissées en exercice...