Notion de portée locale (1)

Les notions de *portée* ***locale*** v.s. ***globale*** définies pour les blocs sont également valides dans le cadre des fonctions[1].

Comme dans le cas des blocs, la *portée* permet de résoudre les problèmes de conflits de référençage entre entités ***externes*** (i.e. définies à l'extérieur du corps de la fonction) et entités ***internes*** (i.e. définies dans le corps-même de la fonction).

Exemple:

```
int i(2);
void f(int a);
void main {
   f(i);
}
void f(int a) {
   int i(10);
   cout << a*i << endl;
}
```

Quelle variable est référencée par le `i` de l'instruction `cout << a*i << endl` ?

1. Cela est bien naturel, puisque le corps d'une fonction n'est autre qu'un bloc.

Notion de portée locale (2)

Les règles utilisées en C++ sont:

- En cas de conflit de référence entre une entité interne et une entité externe, **l'entité interne est systématiquement choisie**; on dit que les **références internes sont de portée locale**, i.e. limitées aux entités internes de la fonction. On dit également que les entités internes ***masquent*** les entités externes.

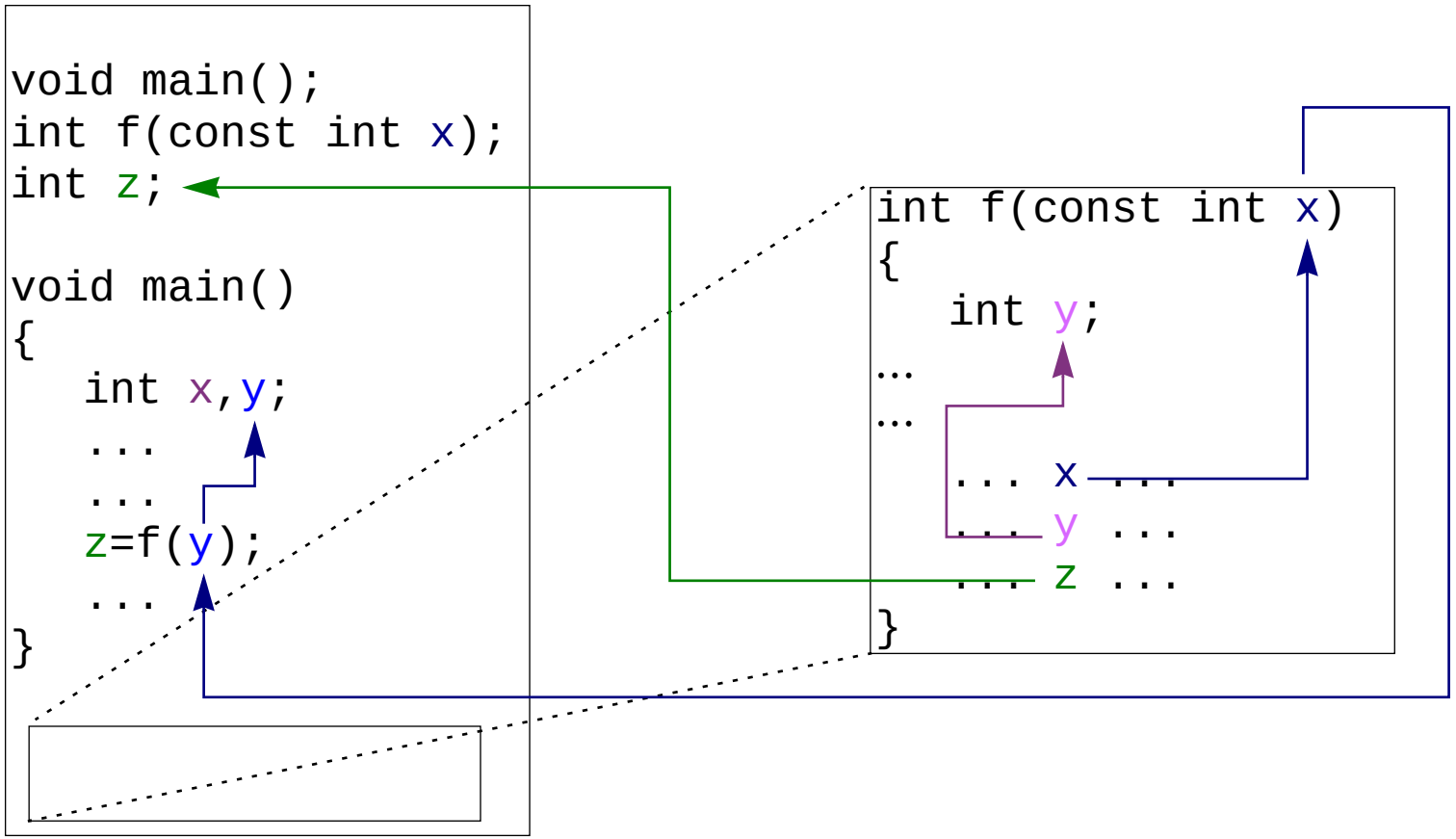
- Les **arguments formels** d'une fonction constituent des **entités internes** à sa définition.

- Les références (non ambiguës) à des entités externes sont possibles[2]. Il est néanmoins préférables de les éviter, en **explicitant systématiquement** toute référence à une entité externe **par le biais d'un argument formel**.

2. Une fonction modifiant des variables externes sera dite «*à effet de bord*», ce qui est fortement déconseillé.

Notion de portée locale (3)

Schématiquement, ces règles peuvent se représenter par:



Le passage des arguments

En C++, on distingue pour les fonctions
2 types de **passage d'arguments**:

☞ le *passage par **valeur***:

dans lequel la fonction travaille sur
des **copies locales** des arguments transmis

☞ et le *passage par **référence***:

dans lequel la fonction travaille effectivement
l'argument **transmis** lui-même**.**

Passage par valeur (1)

La variable locale associée à un argument formel
passé par valeur correspond à une ***copie locale*** de l'argument utilisé lors de l'appel de la fonction.

C'est donc une entité distincte,
mais de même valeur littérale, et en conséquence,
les modifications éventuelles apportées à l'argument formel
à l'intérieur de la fonction n'affectent pas la variable originale:

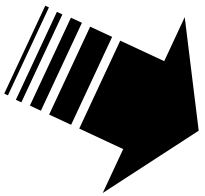
> Les modifications effectuées à l'intérieur de la fonction
> **ne sont pas répercutées à l'extérieur**.

Passage par valeur (2)

Exemple

```
void f(int x)
{
   x = x + 2;
   cout << "'x' vaut: " << x << endl;
}

void main()
{
   int val(0);
   f(val);
   cout << "'val' vaut: " << val << endl;
}
```



```
'x' vaut: 2
'val' vaut: 0
```

La modification de l'argument formel `x` effectuée dans la fonction `f()` laisse inchangée la valeur de la variable `val` utilisée comme argument (associé à `x`) lors de l'appel de la fonction..

De ce fait, il est tout à fait possible d'utiliser des valeurs littérales comme argument lors des appels aux fonctions [pour les arguments passés par valeur].

Passage par référence (1)

La variable locale associée à un argument formel
passé par référence ne correspond qu'à un autre nom
(synonyme local) pour l'argument utilisé lors de l'appel de la fonction[3].

Aucune nouvelle variable n'est crée,
et en conséquence, les modifications éventuelles apportées
à l'argument formel à l'intérieur de la fonction affectent également
l'argument original:

> Les modifications effectuées à l'intérieur de la fonction
> **sont répercutées à l'extérieur**.

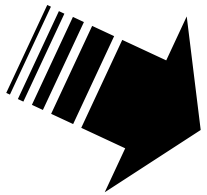
Le passage par référence doit être explicitement indiqué. Pour ce faire, on ajoute le symbole «&» au type des arguments formels concernés (par exemple: `int&`, `bool&`, ...).

3. Ce type de «variable» est appelé ***référence***, d'où le nom de *passage par référence*.

Passage par référence (2)

Exemple

```
void f(int& x) {
   x = x + 2;
   cout << "'x' vaut: " << x << endl;
}
void main() {
   int val(0);
   f(val);
   cout << "'val' vaut: " << val << endl;
}
```



```
'x' vaut: 2
'val' vaut: 2
```

Remarquons que dans le cas d'arguments passés par référence, les valeurs littérales ne peuvent être utilisés comme arguments lors de l'appel à la fonction que pour des arguments formels déclarés constants.

Entités de type «référence à»

De manière générale, les entités de type «*référence à*» (appelée plus simplement des ***références***) permettent de définir de nouveaux noms (**synonymes** ou ***alias***) pour des variables ou constantes existantes.

Leur déclaration se fait de la même manière qu'en paramètre des fonctions; la syntaxe est:

*[***const***]* *<type>***&** *<alias>***(***<entité>***);**

Où *entité* est une variable, une constante ou une référence de type *type*.[4]

Exemple:

```
char c('a');
const int i(2);
...
char& c_alias(c);                  // un simple synonyme de «c»
const int& i_alias(i);             // un synonyme (obligatoirement) constant de la constante «i»
const char& c_alias2(c_alias);     // un synonyme constant de la variable «c»
```

Une référence ne peut pas être ré-assignée. Une fois déclarée, elle restera donc toujours associée au même élément.

4. Il est possible de déclarer des références *constantes* à des variables, mais il n'est pas permis de déclarer des références *simples* à des constantes.

Arguments par défaut (1)

Lors de sa définition,
une fonction peut être munie d'**arguments avec valeur par défaut**,
pour lesquels il n'est alors pas obligatoire de fournir de valeur
lors de l'appel de la fonction.

La syntaxe pour définir des arguments
avec **valeur par défaut** est:

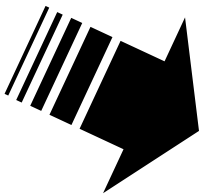
```
<type> <argument> = <valeur>
```

Arguments par défaut (2)

Exemple

```
void afficheLigne(const char c, const int n = 5)
{
   for (int i(0); i<n; cout << c, ++i);
   cout << endl;
}

void main()
{
   afficheLigne('*');
   afficheLigne('+',8);
}
```



```
*****
++++++++
```

Dans le cas du premier appel («`afficherLigne('*')`»),
la valeur du second argument n'étant pas spécifiée,
celui-ci prend la valeur par défaut «5».

Par contre, il prend la valeur explicitement spécifiée («8»)
dans le cas du second appel («`afficherLigne('+',8)`»).

Arguments par défaut (3)

Attention

- Dans le cas d'une fonction avec *prototype* et *définition* distincts, la spécification des arguments avec valeur par défaut **ne doit être réalisée que lors du *prototypage***.
- Les arguments avec valeur par défaut doivent apparaître **en dernier** dans la liste des arguments de la fonction.
- Lors de l'appel d'une telle fonction, avec plusiqurs arguments avec valeur par défaut, les arguments omis doivent être **les derniers** de la liste des arguments optionnels.

Exemple:

```
void f(int x, int y=2, int z=3); // prototype
void f(int x, int y, int z) { ... } // définition
...
f(1) <=> f(1,2,3)
f(0,1) <=> f(0,1,3)
```

Surcharge des fonctions (1)

On appel «***signature*** d'une fonction» le couple constitué de l'identificateur de la fonction et de la liste de types de ses arguments formels.

(le type de retour de la fonction ne fait pas partie de la signature !)

Comme c'est par le biais de leur signature que le compilateur identifie les fonctions utilisées par le programmeur[5], il est de ce fait tout à fait possible de définir **plusieurs fonctions avec un même identificateur** (nom), **mais de listes d'arguments différentes** (soit par le nombre d'arguments, soit par leur type).

Un tel mécanisme, appelé ***surcharge*** [des fonctions], est très utile pour l'écriture de fonctions correspondant à des traitements de même nature, mais s'appliquant à des entités de types différents.

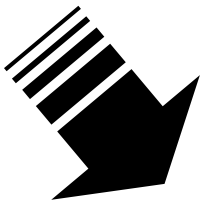
5. La ***signature*** d'une fonction est donc similaire à la notion d'***attribut identifiant*** dans le domaine des bases de données.

Surcharge des fonctions (2)

Exemple:

```
void affiche(const int x) {
   cout << "entier: " << x << endl;
}
void affiche(const float x) {
   cout << "réel: " << x << endl;
}
void affiche(const int x1, const int x2) {
   cout  << "couple: ("
         << x1 << ',' << x2 << ')' << endl;
}

void main() {
   affiche(1.0);
   affiche(1);
   affiche(5,2);
}
```



```
réel: 1.0
entier: 1
couple: (5,2)
```

Les arguments avec valeur par défaut **ne font pas** partie de la signature.

La classe «string»

Comme dans le cas des tableaux,
le type natif (*built-in*) fourni par le langage pour la représentation
des chaînes de caractères (c'est-à-dire un tableau de taille fixe, avec le
caractère `'\0'` comme marqueur de fin de chaîne) n'est pas toujours très pratique.

La librairie standard de C++ met donc à disposition du programmeur
un nouveau type (en réalité une *classe*) dénommé ***string***, permettant une représentation
efficace des chaînes, et offrant une large panoplie de fonctions nécessaires à la
manipulation et au travail avec de telles chaînes:

- concaténation de chaînes,
- ajout/retrait d'un ou plusieurs caractères,
- recherche de sous-chaînes,
- substitution de sous-chaînes,
- ...

Pour pouvoir utiliser des *string* dans un programme, il faut importer les prototypes et
définitions contenus dans la librairie au moyen de la directive d'inclusion:

```
#include <string>
```

Déclaration d'une chaîne

Comme pour tout les autres types,
les variables correspondant à des chaînes de caractères
`string` doivent être déclarées; la syntaxe est:

`string` «*`identificateur`*»`;`

Exemple:

```
#include <string>
...
string chaine;
```

Les variables déclarées suivant la syntaxe indiquée ci-dessus sont automatiquement initialisées à une valeur correspondant à la chaîne vide.

Déclaration+initialisation d'une chaîne

La syntaxe d'une déclaration avec initialisation est:

```
string «identificateur»(«valeur»);
```

où *valeur* est:

- ⇨ soit une chaîne de caractères de la forme `"..."`
- ⇨ soit une variable, constante ou référence de type `string`

Exemple:

```
string str("une chaîne");
string chaine(str);
```

La valeur littérale à utiliser pour une initialisation explicite à une chaîne vide est: «""».

Affectation d'une chaîne

Toute variable de type `string` peut être modifiée par affectation:

«*identificateur*» **=** «*valeur*»**;**

où *valeur* est:

- ⇨ soit un caractère (par exemple: «'i'»)
- ⇨ soit une chaîne de la forme «"..."»
- ⇨ soit une référence à une autre entité de type `string`

Exemple:

```
string chaine;                          // chaine  vaut «»
const string chaine2("test");           // chaine2 vaut «test»
chaine = 'c';                           // chaine  vaut «c»
chaine = string("str-temporaire");      // chaine  vaut «str-temporaire»
chaine = "built-in str";                // chaine  vaut «built-in str»
chaine = chaine2;                       // chaine  vaut «test»
```

Dans le cas de l'affectation par un caractère, la valeur affectée à la chaîne est une **chaîne de caractère** réduite à ce caractère.

Opérateurs relationnels entre chaînes

Les **opérateurs relationnels** suivant sont définis pour les chaînes de type `string`:

Opérateur	*Opération*	
< <= > >=	strictement inférieur inférieur ou égal strictement supérieur supérieur ou égal	La relation d'ordre utilisée pour ces opérateurs est l'**ordre alphabétique**. (plus exactement, ordre **lexicographique**)
== !=	égalité différence (non-égalité)	

L'ordre alphabétique n'est pas respecté dans le cas des caractères accentués, et les majuscules sont considérées comme précédant les minuscules.

Accès aux caractères

Comme dans le cas des tableaux, il est possible d'accéder individuellement à chaque caractère constitutif d'une chaîne:

Si `str` est une chaîne de type `string`, l'expression `str[i]` fait référence au $(i+1)^{ème}$ caractère de `str`.

Chacun des éléments `str[i]` d'une chaîne `str` de type `string` est de type `char`.

Toujours comme dans le cas des tableaux, les éléments d'une chaîne sont indicés de 0 à (*taille* - 1), où *taille* est la longueur de la chaîne (c'est-à-dire son nombre de caractères).

Concaténation de chaînes

La ***concaténation*** de chaînes de type `string`
est réalisée par l'opérateur « **+** »

L'expression « `chaine1 + chaine2` » correspond donc à une **nouvelle chaîne**, dont la valeur littérale est constituée par la concaténation des valeurs littérales de `chaine1` et de `chaine2`.

Les combinaisons suivantes sont possibles:

- *string* + *string*,
- *string* + *char*,
- *string* + `"..."`,
- *char* + *string*,
- `"..."` + *string*,

Méthodes de `string`: prédicats

Parmi les fonctions (*méthodes*)[1] disponibles, on trouve[2]:

Prédicats:

- `int` **`size`**`()`
  `int` **`length`**`()`

  Toutes deux renvoient la longueur de la chaîne (i.e. le nombre de caractères).

  Une manière usuelle pour parcourir un à un les caractères d'une chaîne est donc l'itération `for` suivante:

```
for (int i(0); i<str.size(); ++i) {
   // traitements avec «str[i]»
}
```

- `bool` **`empty`**`()`: indique si la chaîne est vide (ou non).

  On a l'équivalence suivante: «`str.empty()`» <=> «`(str.size() == 0)`»

1. Pour plus d'informations sur l'appel de tels fonctions, se référer au chapitre présentant les vecteurs.
2. Par convention, «`str`» désignera la variable de type *`string`* pour laquelle la méthode est invoquée

Méthodes de `string`: insertions

- `string& insert(int pos, const char[] s)`
  `string& insert(int pos, const string& s)`

  Insère, à partir de la position indiçée par `pos`, la chaîne `s`, et renvoie une référence à la chaîne modifiée.

  Exemple: «`string("1234").insert(2,"-xx-")`» utilise le premier des prototypes spécifiés ci-dessus, et renvoie une référence à la chaîne de type `string` et de valeur «`12-xx-34`».

- `string& insert(int pos, const char[] s, int len)`
  `string& insert(int pos, const string& s, int start=0, int len=npos)`[3]

  Insère, à partir de la position indiçée par `pos`, une sous-chaîne de `s` débutant à la position `start` (ou 0 dans le cas du premier prototype), et de longueur `len`. Une référence à la chaîne modifiée est retournée.

  Exemple: «`string("1234").insert(2,string("#$%-!"),3,1)`» utilise le second prototype[4], et renvoie une référence à la chaîne de type `string` et de valeur «`12-34`».

3. Une constante particulière, batisée «`string::npos`» permet en effet de représenter la notion «*jusqu'à la fin de la chaîne*».
4. Remarquons que grâce à la convertion automatique (casting) des chaînes de forme «"..."» vers le type `string`, la commande peut également s'écrire: «`string("1234").insert(2,"#$%-!",3,1)`».

Méthodes de `string`: remplacements

- `string& replace(int pos, int long, const char[] s)`
  `string& replace(int pos, int long, const string& s)`

  Substitue `s` au `long` caractères de la chaîne, à partir de la position indiçée par `pos`. Renvoie une référence à la chaîne modifiée.

  Exemple: «`cout << string("1234").replace(2,1,"-trois-")`» produira comme résultat l'affichage de la chaîne «`12-trois-4`» : les caractères de l'intervalle [*pos*, *pos*+*long*-1], soit le caractère [2,2], sont remplacés par la chaîne `s`.

- `string& replace(int pos, int long, const char[] s, int len)`
  `string& replace(int pos, int long const string& s, int start, int len)`[5]

  Substitue la sous-chaîne de `long` caractères et débutant à la position `pos`, par la sous-chaîne de `s` de `len` caractères et débutant à la position `start` (ou 0 dans le cas du premier prototype). Une référence à la chaîne modifiée est retournée.

  Exemple: «`cout << string("1234").replace(2,1,"-trois-",1,5)`» produira comme résultat l'affichage de la chaîne «`12trois4`» : les caractères de l'intervalle [*pos*, *pos*+*long*-1], soit le caractère [2,2], sont remplacés par la sous-chaîne de s [*start*, *start*+*len*-1].

5. Avec les mêmes valeurs par défaut pour `start` et `l` que dans le cas de la méthode `insert`.

Méthodes de `string`: recherches

- `int find(char s, int pos = 0)`
  `int find(const char[] s, int pos = 0)`
  `int find(const string& s, int pos = 0)`

  Retourne l'indice du premier caractère de l'occurrence de `s` **la plus à gauche** dans la chaîne, éventuellement privée de ses `pos` premiers éléments (la recherche débute à partir de la position `pos`). Renvoie «`string::npos`» dans le cas où aucune occurrence n'est trouvée.

  Exemple: «`string("baabbaabbaab").find("ba",2)`» renverra 4

- `int rfind(char s, int pos = npos)`
  `int rfind(const char[] s, int pos = npos)`
  `int rfind(const string& s, int pos = npos)`

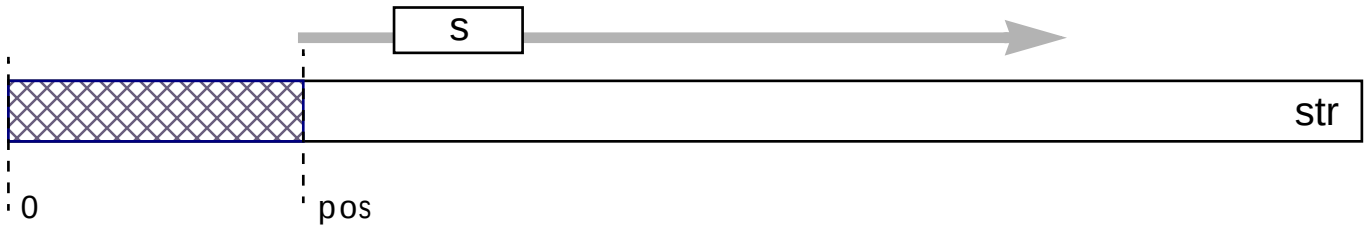
  Retourne l'indice du premier caractère de l'occurrence de `s` **la plus à droite** dans la chaîne, éventuellement privée de ses (*taille*-`pos`) derniers éléments (la recherche débute à partir de la position `pos`). Renvoie «`string::npos`» dans le cas où aucune occurrence n'est trouvée.

  Exemple: «`string("baabbaabbaab").rfind("ba")`» renverra 8

Fonctionnement de `find` et `rfind`

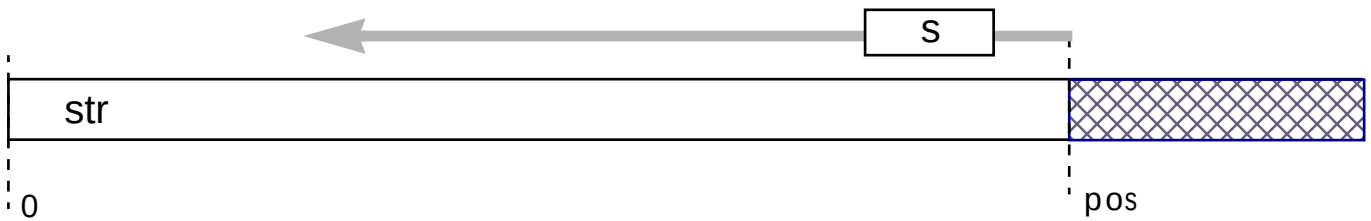
- La méthode «`find`» effectue une recherche du début vers la fin, en prenant comme début la position éventuellement spécifiée comme position de départ.

  Ainsi, «`str.find(s, pos)`» consiste à faire une recherche de `s` dans `str`, privée de ses `pos` premiers éléments:



- La méthode «`rfind`» effectue une recherche de la fin vers le début, en prenant comme fin la position éventuellement spécifiée comme position de départ.

  Ainsi, «`str.rfind(s, pos)`» consiste à faire une recherche «arrière» de `s` dans la sous-chaîne préfixe de `str` et de longueur `pos` + 1:



Exemple d'utilisation des strings (1)

On désire réaliser une extension automatique des abbréviations dans un texte:

les CFF évoluent, c-à-d progressent.

les Chemin de Fer Fédéraux évoluent, c'est-à-dire progressent.

On utilise pour cela une table de correspondance entre les abbréviations (acronymes) et leur signification:

Acronyme	***Signification***
CFF	Chemin de Fer Fédéraux
c-à-d	c'est-à-dire
...	...

Exemple d'utilisation des strings (2)

```
#include <string>
#include <vector>

// Définition des composants de la table associative
typedef vector<string> Strings;

// Table associative, globale (externe)
Strings acronymes;
Strings significations;

// Fonction de remplacement:
// utilise les variables externes définissant les acronymes,
// mais uniquement en lecture (pas d'effet de bord).
void expliciteAcronymes(string& str)
{
   int debutAbbrev;
   for (int i(0); i<acronymes.size(); ++i)
      while ((debutAbbrev=str.find(acronymes[i])) != string::npos)
         str.replace(debutAbbrev,acronymes[i].size(),significations[i]);
}
...
```

Entrées-sorties en C++

En C++, les Entrées-Sorties sont gérées par le biais d'un ensembles de classes spécialisées, les ***streams*** (*flots*).

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).

Ainsi, un *flot* (stream) représente le **flux de données** entre le programme et un dispositif d'entrée-sorties externe (écran, imprimante, ...) ou un fichier. Notons que, pour obtenir de meilleurs performances, est par défaut associé à chaque flot un **tampon mémoire**, par lequel les données transitent.

Comme nous l'avons déjà vu, les opérations de sorties sont réalisées à l'aide de l'*opérateur* ***d'insertion*** « << », tandis que celle d'entrées le sont à l'aide de l'*opérateur* ***d'extraction***, « >> ».

Il existe un certain nombre de variables prédéfinies de type `stream`:

- **`cin`** : flot lié à l'*entrée standard*, qui par défaut est le clavier
- **`cout`** : flot lié à la *sortie standard*, qui par défaut est la console
- **`cerr`** : flot lié à la *sortie d'erreur standard*, qui par défaut est la console.
  Remarquons que `cerr` n'a pas de mémoire tampon (pas besoin de «`flush`»).

Strings et Streams

Il peut parfois être utile, en vue d'effectuer des traitements particuliers, de disposer d'une **représentation sous forme de chaîne de caractères** des différentes variables d'un programme.

Une telle convertion de représentation est automatiquement réalisée lorsque l'on insère ces éléments dans un *stream* de sortie, tel que `cout`.

Mais peut-on récupérer ce qui est transmis dans le flot ?

A défaut de récupérer ce qui est effectivement transmis, on peut du moins en obtenir l'équivalent, en utilisant des strings hybrides, associés à des streams.

Ces entités hybrides sont désignées «**`stringstream`**».

Pour pouvoir utiliser les *stringstream* dans un programme, il faut importer les prototypes et définitions contenus dans la librairie, au moyen de la directive d'inclusion:

```
#include <sstream>
```[6]

6. La version de *gcc* utilisée dans le cadre du cours utilise une ancienne représentation de ces entités, pas tout à fait compatible avec la nouvelle norme. Toutefois, une adaptation a été écrite pour le besoin du cours, et placée dans le répertoire de la librairie standard.

stringstream de sortie

Pour obtenir la représentation alphanumérique d'une variable, il faut utiliser un `stringstream` de sortie, dans lequel on insérera les données désirée, comme dans le cas de `cout`, avant d'en extraire, l'équivalent sous forme de chaîne de caractère.

Le type (la classe) matérialisant de tels éléments est désigné: **`ostringstream`**, et l'extraction de la représentation chaîne s'obtient au moyen de la méthode:

```
string str()
```

Exemple:

```
#include <sstream>
...
string composeMessage(const int errno, const string& description)
{
   ostringstream ost;
   ost << "Erreur (n°" << errno << "): " << description;
   return ost.str();
}
```

stringstream d'entrée

Il est également possible de réaliser des entrées depuis un `string` (d'entrée) préformaté: il suffit de définir le contenu du `stringstream` d'entrée lors de sa déclaration, et d'utiliser l'opérateur d'extraction sur le *stream* ainsi obtenu.

Le type (la classe) matérialisant de tels éléments est désigné: **`istringstream`**.

Exemple:

```
#include <sstream>
...
// affiche 1 mot par ligne
void wordByWord(const string& str)
{
   istringstream ist(str);
   string s;
   while (ist >> s) cout << s << endl;
}
...
   wordByWord(composeMessage, 5, "Fichier non trouvé");
}
```

```
Erreur
(n°5):
Fichier
non
trouvé
```