Table des matières

Table des matières 1

Remarques sur les exercices de C++ 5

Organisation des TDs 5
Environnement de développement en C++ 5
Environnement par défaut 5
Environnements intégrés (IDE) 5
Compatibilité 6
Autres ressources 6

TD 1
Après une pile, une file générique 7

1.1 Objectif 7
1.2 Énoncé du problème 7
1.3 Note sur l'instanciation des classes génériques (templates) 7

TD 2
Programmation procédurale et introduction à la *Standard Template Library* (STL) : chaînes de caractères, flux d'E/S et vecteurs 9

2.1 Objectif 9
2.2 Description rapide de quelques éléments de la STL 9
2.2.1 Entrées-sorties dirigées par les types (`iostream`) 9
2.2.2 Chaînes de caractères : la classe `string` 11
2.2.3 Tableaux dynamiques : la classe `vector` 11
2.2.4 Une dernière remarque pour utiliser `iostream`, `string` et `vector` 12
2.3 Exercices 12
2.3.1 Palindrome 12
2.3.2 Utilisation de `vector` 12
2.3.3 Indexation et exceptions 13
2.3.4 Passage de paramètres par référence ; fonctions génériques 13
2.3.5 Définition de `operator<<` et `operator>>` 13
2.3.6 Allocation dynamique de mémoire avec l'opérateur `new` 14
2.3.7 Opérateur `new` ou vecteur ? 14

TD 3
Programmation procédurale et introduction à la *Standard Template Library* (STL): strings, containers, itérateurs et algorithmes 15

Objectif 15
Code fourni 15
3.1 Quelques fonctions complémentaires sur les strings 15

3.2 Introduction aux containers, itérateurs et algorithmes de la STL 16
3.2.1 Collections (*containers*) de la STL 16
3.2.2 Itérateurs 17
3.2.3 Algorithmes 20
3.3 Encore des compléments sur les strings 22
3.4 Constitution de l'index d'un texte 22

TD 4
Une classe `Date` 24
4.1 Objectif 24
4.2 Description de la classe `Date` 24
4.3 Note sur le code C fourni 25

TD 5
Vecteurs et matrices 26
5.1 Objectif 26
5.2 Vecteurs mathématiques : classe `MVector` 26
5.2.1 Spécification 26
5.2.2 Réalisation 26
5.3 Matrices rectangulaires : classe `Matrix` 27
5.3.1 Spécification 27
5.3.2 Réalisation 27
5.3.3 Note sur les opérations d'indexation 28
5.4 Remarques et conseils 28
5.4.1 Remarque sur le code fourni 28
5.4.2 Conseil 28

TD 6
Variations sur les matrices 30
6.1 Objectif 30
6.2 Diverses sortes de matrices 30
6.2.1 Matrices carrées 30
6.2.2 Matrices diagonales 30
6.2.3 Matrices scalaires 31
6.3 Question pour les meilleurs : métamorphoses des matrices 31
6.4 Remarques et conseils 32
6.4.1 Code fourni 32
6.4.2 Conseils 32
6.4.3 Limites de l'exercice 32

TD 7
Opérations de copie 34
7.1 Objectif 34
7.2 Synthèse automatique des opérations de copie par C++ 34

7.3 Importance des opérations de copie 34
7.3.1 Expérimentation avec les destructeurs et les copies 35
7.3.2 Pour ne pas perdre la main... 35

TD 8
Variations sur les listes, les files et les piles 36
8.1 Objectif 36
8.2 Une simple classe liste générique 36
8.3 Les listes comme implémentation des files et des piles 37
8.3.1 Les classes `Queue` et `Stack` 37
8.3.2 Dérivation privée 38
8.3.3 La classe `Priority_Queue` 38
8.3.4 Typage dynamique 39

TD 9
Fonctions virtuelles. 40
Objectif 40
9.1 Extension de la classe `Expr` 40
9.1.1 Deux extensions simples 40
9.1.2 Une extension un peu plus compliquée 40
9.2 Menus en cascade 42

TD 10
Algorithmes de la STL 44
Objectif 44
10.1 Définition de collections polymorphes 44
10.2 Manipulation de collections polymorphes 44
10.2.1 Utilisation des fonctions-membres de `Figure` 44
10.2.2 Tri d'une collection 44
10.3 Identification du type des figures d'une collection 45
10.4 Destruction des figures d'une collection 45

Remarques sur les exercices de C++

ORGANISATION DES TDS

Les informations sur le déroulement du cours et des TDs se trouvent sur mon site Web local à Polytech'Nice Sophia,

```
http://www-local/~jpr
```

où se trouve également le présent document.

Lisez avec soin et complètement le sujet de chaque TD avant de vous lancer à corps perdu dans la programmation.

Pour certains TDs un code de départ est fourni. De plus les solutions des TDs sont mises en ligne régulièrement sur le site Web. Pour compiler le code fourni et les solutions, il ne suffit pas de copier chez vous le répertoire contenant le TD courant. Encore faut-il respecter la hiérarchie proposée (en particulier le répertoire `include`, la `Makefile` et le fichier `default.mk` sont indispensables). Le plus simple est de suivre les indications du site Web pré-mentionné en copiant et décompressant chez vous le fichier `Basic_C++.zip` fourni sur le site et qui contient les premiers sujets de TDs et les exemples de programmes du cours. Il vous suffira ensuite de copier au fur et à mesure les solutions et sujets suivants en respectant cette hiérarchie.

ENVIRONNEMENT DE DÉVELOPPEMENT EN C++

Environnement par défaut

Les TDs ayant lieu sous Unix (Linux), la solution par défaut est l'utilisation de **g++**, avec Emacs/XEmacs comme éditeur et **gdb** pour la mise au point. Apprenez

- à rédiger des *makefiles* simples (en général, les exemples viennent avec leur propre `Makefile` qu'il suffit d'éditer pour l'adapter à vos besoins) ;
- à compiler *dans* l'éditeur (Emacs ou XEmacs) : c'est facile, cela évite les erreurs et les pertes de temps ;
- à utiliser un *debugger* comme **gdb** ou, plus agréablement, l'une de ses interfaces graphiques (**xxgdb** ou **ddd**) ; compilez toujours avec l'option **-g** — du moins pour ces exercices.

Certes il s'agit là d'un environnement rustique par bien des côtés comparé aux dorures des environnement intégrés (voir juste après). Mais, bien utilisé, c'est ce que je connais de plus efficace et rapide pour le cycle de développement édition-compilation-mise au point.

Environnements intégrés (IDE)

Bien entendu vous pouvez adopter un autre environnement, à condition d'y être à l'aise. Tant qu'à faire, il vaut mieux privilégier les environnements qui intègrent la compilation et la mise au point (*Integrated Development Environment* ou IDE). Notez cependant qu'ils nécessitent un apprentissage non négligeable pour en tirer pleinement partie et sont souvent (beaucoup) plus lents que la solution par défaut évoquée plus haut.

Un grand nombre d'IDE sont disponibles sous Linux : **kdevelop** (à la VisualC++, utilisé par KDE), **eclipse** (personnellement, cet environnement m'est indispensable en Java, mais je

trouve son mode C++ un peu limité et assez pénible), **anjuta**, CodeBlocks, CodeForge (celui-là est payant, mais pas cher !), etc.

Sous Windows, il y a bien sûr l'incontournable Microsoft Visual C++. C'est un environnement très agréable à utiliser ; évitez absolument la version 6.0 dont le compilateur est abominablement mauvais dans son support de la norme C++ ; la version 7.x (version dite **.net**) est un peu meilleure mais n'est pas exempte de bugs ; les versions 8 (Visual Studio 2005 pour l'environnement complet, ou Visual C++ Express 2005 pour la partie C++ uniquement) et 9 (Visual Studio 2008 et Visual C++ Express 2008) fournissent (enfin !) un compilateur C++ digne de ce nom.

Sous Windows encore, Cygwin vous permet de développer du C++ « comme sous Linux », avec la compatibilité des outils et des compilateurs. Enfin à noter l'excellent Dev-C++, un environnement intégré « à la Visual », mais simple, léger et rapide, supportant les compilateurs GNU. Cet outil n'évolue malheureusement plus depuis quelques années, mais il reste malgé tout utilisable et bien adapté à des projets de taille pas trop énorme.

Noter que les deux IDEs CodeBlocks et **eclipse** sont disponibles (et identiques) à la fois sous Unix/Linux et sous MW Windows.

COMPATIBILITÉ

Toutes les solutions des exercices sont compilables et exécutables à la fois sous Unix/Linux avec les outils GNU et sous Windows avec Cygwin ou Dev-C++. GNU MAKE est indispensable ; la version du compilateur utilisée actuellement est **g++-3.4.x** sous Windows et **g++-4.x** sous Unix (g++-4.3.2 en septembre 2008).

AUTRES RESSOURCES

Voir mon site Web local pour des compléments sur C++ :

```
http://www-local/~jpr
```

Jean-Paul RIGAULT
jpr@polytech.unice.fr
Septembre 2008

TD 1
Après une pile, une file générique

1.1 OBJECTIF

L'objectif de cet exercice est de montrer que l'on peut réaliser du code utile par simple imitation. C'est aussi une occasion de mettre en œuvre certains des mécanismes principaux de C++ qui ont été survolés dans l'introduction et de prendre en main l'environnement de développement en C++ (éditeur, compilateur, *makefiles*, etc.).

Peut-être une autre classe un peu moins proche de Stack ?

1.2 ÉNONCÉ DU PROBLÈME

Vous trouverez dans le répertoire `Fifo` le source de la classe générique `Stack` vue en cours accompagné d'un programme de test, ainsi que d'une `Makefile`. Après avoir copié le répertoire[1] chez vous, compilez (il suffit de faire `make`) et exécutez le programme de test (`main_Stack.exe`) dans une fenêtre **shell**. Le programme attend que vous entriez des caractères et que vous terminiez par une fin de fichier (au terminal, `^D` seul sur une ligne).

En prenant cette classe `Stack` comme modèle, on vous demande d'écrire une classe `Fifo`, également générique, qui réalise une file avec la stratégie « premier entré, premier sortie » (*first in, first out*). Une telle classe modélise une file d'attente.

Votre classe `Fifo` devra être dotée des propriétés suivantes :

- une taille maximale `N` (une constante),
- un constructeur par défaut (c'est-à-dire sans paramètre) qui construit une file de taille `N` mais qui ne contient aucun élément utile,
- deux opérations principales : `put()` qui place un élément dans la file, et `get()` qui retire l'élément le plus ancien de la file et retourne sa valeur,
- deux prédicats `is_full()` et `is_empty()`, comme dans `Stack`.

Bien entendu vous devrez aussi lever des exceptions en cas d'opérations impossibles. Enfin votre file devra être gérée comme un *tampon circulaire* afin que la place libérée par `get()` puisse être réutilisée par `put()`.

Dans votre programme de test, essayez d'instancier votre classe `Fifo` avec plusieurs types très différents (`char`, `double`, pointeurs...).

1.3 NOTE SUR L'INSTANCIATION DES CLASSES GÉNÉRIQUES (*TEMPLATES*)

Lorsque le compilateur instancie une classe générique comme `Stack` ou `Fifo`, il produit le code source des classes correspondant aux divers paramètres d'instanciation[2]. À cause du mécanisme de compilation séparée de C/C++, le compilateur a besoin non seulement de la définition de la classe, mais encore de la définition (du corps) de toutes ses fonctions-membres. Ceci explique que, contrairement aux (bonnes) habitudes, la distinction entre fichier d'entête (`.h`) et fichier de corps (`.cpp`) est, *dans le cas des templates*, sans grande signification

1. et aussi quelques éléments du répertoire supérieur (voir la remarque préliminaire à tous ces sujets de TD)
2. L'instantciation des *templates* est un mécanisme analogue à la macro-génération, mais il s'agit d'une macro-génération *dirigée par les types*.

puisque le compilateur a besoin du contenu des deux. Si l'on veut cependant conserver cette séparation on peut alors inclure le fichier `.cpp` dans le fichier `.h` comme il est fait dans l'exemple de la classe `Stack`. Bien entendu, dans ce cas, le fichier `Stack.cpp` ne doit pas être compilé séparément puisqu'il le sera lors de son inclusion (voir la `Makefile`).

Noter également qu'en C++, il n'y a jamais *aucune* excuse pour éviter de protéger les fichiers d'entête contre une inclusion multiple. Vous *devez* donc encadrer vos fichiers `.h` par la séquence magique

```
#ifndef <un identificateur unique qui rappelle le nom du fichier>
#define <le même identificateur>

<le contenu du fichier>

#endif
```

TD 2
Programmation procédurale et introduction à la *Standard Template Library* (STL) : chaînes de caractères, flux d'E/S et vecteurs

2.1 OBJECTIF

Ces exercices concernent la programmation procédurale en C++, au sens qu'ils ne requièrent du programmeur aucune *définition* de classe. Cependant on ne se privera pas d'utiliser des classes existantes, notamment certaines des classes prédéfinies de la STL qui représentent les structures de données fondamentales de la programmation.

Parmi les richesses de la STL trois (ensembles de) classes sont plus particulièrement précieuses car elles se substituent avantageusement à des constructions douteuses et pas très sûres de C :

- la classe `string` représente des chaînes de caractères ; contrairement aux chaînes de C (de simples pointeurs `char *`, bornées par le caractère nul), la classe `string` gère automatiquement la mémoire et permet des manipulations globales naturelles et sûres (affectation, concaténation...) .
- les classes de la bibliothèque `iostream` permettent également de rendre sures les fonctions d'entrées-sorties de C, en particulier les tristement célèbres `printf` et surtout `scanf` ;
- la classe `vector` fournit un type de tableau mono-dimensionnel ; contrairement aux tableaux de C, les `vectors` ne se confondent pas avec les pointeurs, s'occupent eux-mêmes de leur allocation mémoire, peuvent grossir (ou maigrir) automatiquement et supportent des opérations globales comme l'affectation ou l'initialisation entre tableaux.

Bien entendu la STL fournit de nombreux autres types représentant des collections homogènes d'objets (on parle de types « conteneurs », *containers*), comme des listes, des piles, des files, des tableaux associatifs (*maps*), des ensembles avec ou sans répétition, etc. Toutes ces collections peuvent être parcourues par un ensemble d'*itérateurs* présentant une interface commune quelle que soit la nature du conteneur. On peut aussi exécuter des opérations globales sur la collection grâce à un ensemble de fonctions génériques appelées *algorithmes*. Nous n'utiliserons pas ces dernières possibilités dans cet série d'exercices.

2.2 DESCRIPTION RAPIDE DE QUELQUES ÉLÉMENTS DE LA STL

2.2.1 Entrées-sorties dirigées par les types (`iostream`)

Les fonctions de la bibliothèque `stdio` de C comme `printf` et `scanf` sont peu sures : en effet le compilateur n'a, en général, aucun moyen de vérifier la cohérence entre les spécifications données dans le format et le nombre et type des paramètres effectifs fournis ; en particulier, dans le cas de `scanf`, il n'a même pas la possibilité de vérifier que ces paramètres sont bien des pointeurs. Les incohérences ne pourront donc être détectées que lors de l'exécution et (au mieux !) seront sanctionnées par un *crash* brutal.

C++ introduit une nouvelle bibliothèque, `iostream`, qui se substitue à `stdio`. En particulier deux nouvelles classes, `ostream` et `istream`, viennent remplacer les traditionnels `FILE *` de `stdio`. Les instances de `ostream` (resp. `istream`) correspondent à des flux d'écriture (resp. de lecture) ; comme dans `stdio`, ces flux sont tamponnés (*bufferisés*) dans l'espace utilisateur. Tout programme C++ s'exécute avec trois instances prédéfinies de ces classes : `cin` est une `istream` correspondant à l'entrée standard, `cout` et `cerr` sont des `ostreams` correspondant respectivement à la sortie et à l'erreur standard.

Utilisation des `ostreams` C++ surcharge l'opérateur << (décalage gauche) pour qu'il prenne une `ostream` comme premier opérande et un objet de n'importe quel type de base comme second opérande :

```
int i = 12;
double x = 3.14;
cout << "i = ";      // afficher une chaîne de C
cout << i;           // afficher la valeur de i
cout << '\n';        // afficher une fin de ligne
cout << "x = ";      // afficher une chaîne de C
cout << x;           // afficher la valeur du réel x
cout << endl;        // afficher une fin de ligne et vider le buffer
```

L'exécution des lignes précédentes produit ce qui suit sur la sortie standard (`cout`) :

```
i = 12
x = 3.14
```

Comme on le voit, le format de sortie est déterminé par le type de l'opérande droit. Noter aussi l'utilisation de la fonction spéciale `endl` qui provoque une fin de ligne et force le vidage du buffer interne (ce qui est différent de la simple sortie d'un caractère `'\n'`).

Il est même possible d'utiliser l'opérateur << en cascade, de sorte que les lignes précédentes peuvent s'écrire sous la forme plus compacte

```
cout << "i = " << i << '\n' << "x = " << x << endl;
```

Utilisation des `istreams` De manière similaire à `operator<<`, C++ définit `operator>>` (décalage droit) qui reçoit un premier argument de type `istream` et un second qui peut être une *variable* de n'importe quel type de base:

```
int i;
double x;
cin >> i >> x;
```

Ces instructions lisent (depuis l'entrée standard `cin`) un entier suivi d'un nombre réel. Pour l'opérateur >>, les espaces[1] servent de séparateurs et sont pour le reste ignorés. Ainsi si on tape sur l'entrée standard

```
··13··5.75
```

(ou les · représentent des blancs) alors les instructions précédentes font que la valeur de `i` sera 13 et celle de `x` sera 5.75.

1. Un *espace* ici doit être pris au sens « unixien » : cela peut être un blanc, un TAB, une fin de ligne, un saut de page, etc.

En fin de fichier[1], `cin.eof()` est vrai ; en cas d'erreur (e.g., mauvais format) `cin.err()` est vrai. Comme toutes les `istreams`, `cin` est convertible implicitement en un booléen qui est vrai si ni `cin.eof()` ni `cin.err()` ne sont vrais. Ainsi on peut écrire

```
if (cin >> i >> j) ...  // si on a pu lire 2 entiers...

// lire un fichiers contenant des entiers jusqu'à la fin de fichier ou une erreur
while (cin >> i) ...    // tant que l'on peut lire un entier...
```

2.2.2 Chaînes de caractères : la classe `string`

La classe `string` propose une considérable amélioration sur les traditionnelles chaînes terminées par le caractère nul de C :

- la mémoire est gérée automatiquement,
- il n'y a pas besoin de s'occuper de la terminaison de la chaîne (le caractère nul),
- les opérations globales sur les `strings` sont possibles et naturelles (copie, comparaison, concaténation, etc.),
- les opérateurs << et >> sont définies et permettent d'afficher et de lire des `strings`,
- etc.

Bien sûr on peut aussi utiliser l'indexation (`operator[]`) sur les `strings`.

Voici un extrait de programme utilisant des `strings` :

```
string greeting;     // initialisation à la chaine vide " "
string answer = "mumble, mumble";
cin >> greeting;
if (greeting == "hi,_there!")
    answer = "hi";
else if (greeting == "how_do_you_do?")
    answer = greeting;
answer[0] = toupper(answer[0]);// premier caractère en majuscule
cout << answer << endl;
unsigned int n = answer.size();// longueur (utile) de la chaîne
```

Rappelez vous que quand on utilise l'opérateur >>, les espaces servent de séparateur mais sont autrement ignorés : il n'est donc possible de lire que des `strings` ne contenant pas d'espace (d'où les caractères `'_'` dans l'exemple ci-dessus). En sortie, cette limitation n'existe évidemment pas.

2.2.3 Tableaux dynamiques : la classe `vector`

Il s'agit d'une classe générique (*template*), c'est donc `vector<T>` où `T` est le type des éléments : un `vector<T>` est un tableau mono-dimensionnel de `T`. Contrairement aux tableaux de C, un vecteur connaît sa taille (fonction-membre `size()`) et on peut effectuer des opérations globales de copie. Bien sûr, on peut aussi appliquer l'opérateur d'indexation à un vecteur (`operator[]`).

Voici un extrait de programme utilisant des vecteurs :

```
vector<int> vi(10); // un vecteur d'entiers à 10 éléments. Attention : ce sont des
                    // parenthèses et pas des crochets ! Il s'agit d'un appel de
                    // constructeur ; tous les éléments sont initialisés à 0
```

1. Rappelons que, sous Unix, on produit une fin de fichier au terminal en tapant `^D` seul sur une ligne ; sous Windows, if faut entrer `^Z` suivi d'un retour-chariot, seuls sur une ligne.

```
vector<int> v;          // un vecteur vide (taille 0)
v = vi;                 // copier vi dans v ; v a maintenant une taille de 10

// operator<< n'est pas défini pour les vector<T> ! Donc une boucle est nécessaire
for (int i = 0; i < v.size; ++i)
    cout << v[i] << ' ';
cout << endl;           // fin de ligne et vidage du buffer
```

De plus les vecteurs peuvent grossir automatiquement sous l'effet de certaine opérations comme `push_back()` qui ajoute un élément à la fin du vecteur :

```
v.push_back(99);    // maintenant v.size() == 11 et v[10] == 99
```

2.2.4 Une dernière remarque pour utiliser `iostream`, `string` et `vector`

Pour utiliser ces trois classes vous ne devez pas oublier d'inclure le fichier d'entête correspondant et d'utiliser une déclaration d'usage de l'espace de noms `std` :

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

2.3 EXERCICES

2.3.1 Palindrome

Un palindrome est, comme chacun sait, une chaîne de caractères qui se lit de la même manière dans les deux sens, comme `"otto"` ou `"madam"`. Écrivez un programme qui lit des chaînes de caractères sur l'entrée standard (jusqu'à la fin de fichier), vérifie s'il s'agit d'un palindrome et imprime le résultat.

2.3.2 Utilisation de `vector`

Un *bookmaker* enregistre dans un fichier chacune de ses transactions sous forme d'une ligne contenant le nom du client et une somme positive si le client lui doit de l'argent ou un somme négative dans le cas contraire (voir le fichier `Strings_and_Vectors/records.txt` pour un exemple).

Écrivez un programme qui effectue les actions suivantes :

- lire chaque ligne d'un fichier comme `records.txt` et la ranger dans un `vector<Bookrecord>`, où `Bookrecord` est la structure suivante :
  ```
  struct Bookecord {
      string _name;
      double _amount;
  };
  ```
- afficher le contenu de ce vecteur sur la sortie standard ;
- entrer dans une boucle où l'on demande de fournir, sur l'entrée standard, un nom de client, où l'on consolide toutes les sommes concernant ce client (c'est-à-dire qu'on fait le total algébrique des sommes dues) et on affiche le résultat sur la sortie standard. ;
- terminer la boucle (et le programme) à la fin de fichier sur l'entrée standard.

Suggestion Vous aurez besoin d'ouvrir un fichier (comme `records.txt`) et de lui associer une `istream`. En fait, vous allez créer une `ifstream` (une classe dérivée de `istream`) de la manière suivante :

```
#include <fstream>
...
ifstream is("records.txt");// constructeur : ouvre le fichier
```

Vous pouvez alors utiliser `is` comme une `istream` (et en particulier profiter de son opérateur >>).

2.3.3 Indexation et exceptions

En fait, les classes `string` et `vector` ont deux opérations d'indexation : l'opérateur usuel de C avec des crochets carrés (`v[i]`) et la fonction-membre `at()` (`v.at(i)`). Ces opérateurs retournent tous les deux une référence sur l'élément. Cependant `at()` vérifie les bornes de l'indice et lève une exception en cas d'échec alors que `operator[]` n'effectue aucune vérification. L'exception levée par `at()` s'appelle `out_of_range` et pour obtenir sa définition vous devez inclure le fichier d'entête `<stdexcept>`.

Vérifiez ce comportement des opérations d'indexation de `string` et `vector`. Vous pouvez utiliser les deux exercices précédents pour héberger vos expériences.

2.3.4 Passage de paramètres par référence ; fonctions génériques

Écrire une fonction `Swap()` (noter la majuscule[1]) qui échange les valeurs de deux variables entières :

```
int a = 1;
int b = 2;
Swap(a, b);
cout << a << ' ' << b << endl; // affiche 2 1
```

Pouvez vous imaginer comment rendre cette fonction générique, c'est-a-dire faire en sorte qu'elle échange les valeurs de ses deux paramètres quel que soit leur type, pourvu que ce soit le même ?

Essayer d'échanger des entiers, des doubles, des `strings`, des `vectors`... avec votre fonction générique.

2.3.5 Définition de `operator<<` et `operator>>`

Modifiez le programme de l'exercice 2.3.2 de la manière suivante :

- définissez les opérateurs de décalage pour la structure `Bookrecord` de façon à pouvoir afficher et lire une telle structure d'un seul coup grâce aux `iostreams` ; les prototypes de ces fonctions doivent être les suivants :
  ```
  ostream& operator<<(ostream&, const Bookrecord&);
  istream& operator>>(istream&, Bookrecord&);
  ```
  Notez le passage par référence sur une variable pour >> (pourquoi ?) ; ces deux opérateurs doivent retourner leur premier paramètre ;
- réécrivez le programme de 2.3.2 afin qu'il utilise ces deux opérateurs.

1. C'est pour éviter de la confondre avec la fonction `swap()` prédéfinie dans la STL, et qui fait évidemment la même chose..

2.3.6 Allocation dynamique de mémoire avec l'opérateur `new`

Reprenez la classe `Fifo` de l'exercice 1.2 du 1. La taille maximale de la `Fifo` est un paramètre *template* `N`,

```
template <typename ELEM, int N> class Fifo;
```

Donc sa valeur doit être connue à la compilation.

Nous souhaiterions changer cela et déterminer la taille maximale de la `Fifo` à l'exécution. Nous retirons donc le paramètre template `N` et dotons notre classe d'un constructeur prenant un entier en paramètre :

```
template <typename ELEM>
class Fifo
{
public:
    Fifo(int n);
    // ... détails omis
};
```

Ainsi, quand nous créons une `Fifo`, nous devons maintenant donner sa taille maximale :

```
int nElems;
Fifo<double> aFifo(nElems);
```

Bien sûr le constructeur doit non seulement ranger cette taille quelque part, mais encore allouer *dynamiquement* (en utilisant l'opérateur `new`) le tableau interne `_tab`.

Modifiez votre classe `Fifo` pour qu'elle corresponde à ces nouvelles spécifications.

2.3.7 Opérateur `new` ou vecteur ?

Modifier la classe générique `Stack` du cours (son code figure dans le répertoire `Fifo` du1) en implémentant le tableau interne `_tab` à l'aide d'un vecteur (`vector`) et en utilisant `push_back()` (voir 2.2.3). Profitez en pour réécrire le `main()` de façon qu'il utilise les `iostreams`.

Comparer la réalisation utilisant `new` et celle avec `vector` ? Quelles sont les avantages et inconvénients de chacune de ces solutions ? Laquelle préférez-vous *in fine* ?

TD 3
Programmation procédurale et introduction à la *Standard Template Library* (STL): strings, containers, itérateurs et algorithmes

OBJECTIF

Cet exercice apporte quelques compléments sur l'utilisation élémentaires de la STL (*Standard Template Library*). En particulier nous récapitulons les collections (*containers*) disponibles et introduisons deux concepts clés de la STL : les *itérateurs* et les *algorithmes*.

CODE FOURNI

Pour démarrer l'exercice, vous aurez besoin de télécharger depuis le site Web le fichier `Strings_and_Algos.zip` et de le décompresser à coté des autres exercices.

Le répertoire obtenu (`Strings_and_Algos`) contiendra alors les fichiers suivants :

- une `Makefile`, prête pour le TD tout entier ;
- `extra_string.h`, `extra_string.cpp` et `test_string.cpp` : les prototypes et les (squelettes) de la définition des fonctions demandées dans les exercices 3.1 et 3.3 ;
- `stl_print.h`, `test_print.cpp`, `camille.txt` : ces fichiers sont utilisés (et décrits) dans l'exercice 3.4.

3.1 QUELQUES FONCTIONS COMPLÉMENTAIRES SUR LES STRINGS

Dans une documentation de la STL (vous devriez assez facilement ent trouver une sur le Web), consulter la liste des fonctions-membres disponibles de la classe `string`. Si vous comparez à la classe `String` de Java, vous constaterez que certaines fonctionnalités sont manquantes. Votre mission est ici d'en réaliser quelques unes. Bien entendu, vous ne pouvez pas modifier la classe `string` : vous implémenterez donc ces fonctionnalités sous forme de fonctions « libres ». Les prototypes de ces fonctions sont donnés dans le fichier fourni `extra_string.h`. Dans ce premier exercice, on ne demande de réaliser que les trois fonctions suivantes :

`bool string_starts_with(const string& s, const string& prefix)`
: vrai si le début de la chaîne `s` est la chaîne `prefix` ;

`bool string_ends_with(const string& s, const string& suffix)`
: vrai si la fin de la chaîne s est la chaîne `suffix` ;

`string trim(const string& s, const string& delims = " \t\n\f\v")`
: retourne une copie de `s` où toutes les occurrences initiales et finales des caractères de `delims` ont été supprimés (mais pas les occurrences au milieu) ;par exemple
`trim("\t  \f\thello\t\nworld\f \v")`
retourne la chaîne `"hello\t\nworld"`. Notez que `delims` a une valeur par défaut qui peut évidemment être modifiée lors de l'appel.

Implémentez ces fonctions (dans `extra_string.cpp`) et testez-les (dans `test_string.cpp`). Ensuite, vous n'aurez plus qu'à faire

```
make test_string.exe
```

pour obtenir votre exécutable.

3.2 INTRODUCTION AUX CONTAINERS, ITÉRATEURS ET ALGORITHMES DE LA STL

Nous avons déjà introduit certains éléments de la STL au cours du TD 2 (classe `string`, `iostream` et `vector`). La STL est aussi brièvement décrite dans le chapitre 6 du cours. Enfin on pourra se reporter au site Web de ce cours ainsi qu'à l'un des sites fournissant une documentation en ligne de la STL (par exemple SILICON GRAPHICS, `www.sgi.com`, ou STLPORT, `www.stlport.org`).

Nous rappelons ici les éléments fondamentaux nécessaires à la compréhension des exercices suivants (3.4 et 3.4).

3.2.1 Collections (*containers*) de la STL

La STL définit un ensemble de classes génériques qui permettent de représenter des *collections homogènes de valeurs*. « Homogène » signifie que tous les éléments de la collection sont de même type ; « valeur » indique que les éléments en question sont copiés dans la collection[1].

La STL définit plusieurs types de collections qui peuvent différer par leur propriétés logiques (celles vues de l'utilisateur) : ordonnées ou non, mode de recherche et d'addition d'un éléments, etc. Elles diffèrent aussi par leur structure interne d'implémentation : listes chaînées, éléments consécutifs ou non en mémoire, arbre binaire, etc. La version initiale de la STL définit dix types de collections résumés dans le tableau ci-après..

Séquences (l'ordre des éléments dans la collection est celui de leur insertion)	
`vector<T>`	« consécutif » en mémoire ; indexation et insertion au milieu et à la fin ; pas d'insertion en tête
`list<T>`	liste doublement chaînée ; insertion partout
`dequeue<T>`	compromis entre `vector` (indexation facile) et `list` (insertion facile)
Adaptateurs de collections	
`stack<T>`	la pile (LIFO) habituelle avec les opérations `push()` et `pop()`
`queue<T>`	la file (FIFO) habituelle avec les opérations `put()` et `get()`
`priority_queue<T>`	une file dans laquelle les éléments sont rangés dans l'ordre (celui de `operator<` sur `T`).
Collections associatives (nécessitent une relation d'ordre — `operator<` par défaut — sur `T`)	
`map<K, T>`	tableau associatif : `K` est le type de la clé, `T` celui de la valeur associée (tableau de `T` indexé par `K`) ; pas de duplication des valeurs de `K` ; collection ordonnée en `K`
`multimap<K, T>`	collection semblable à `map`, mais duplication possible des valeurs de `K`
`set<K>`	ensemble (pas de duplication, donc) ordonné (par défaut avec l'opérateur `<`) de `K`
`multiset<K>`	semblable à `set`, mais duplication possible

1. Si l'on ne souhaite pas copier, on peut bien entendu utiliser des collections de *pointeurs* : c'est alors la valeur du pointeur qui sera rangée dans la collection.

Dans le tableau précédent, `T` désigne en général le « type de valeur » (*value type*) de la collection, c'est-à-dire le type de ses éléments. **Attention**, cependant : dans le cas des `map` et `multimap`, ce type de valeur n'est pas `T`, mais **`pair<const K, T>`** où `pair` est une structure générique prédéfinie :

```
template <typename K, typename T>
struct pair
{
    K first;
    T second;
    // ...
};
```

Pour pouvoir utiliser un type de collection, il suffit en général d'inclure le fichier d'en-tête qui porte son nom (`<list>`, `<dequeue>`, `<map>`, etc.) et de rendre accessible l'espace de noms `std`.

3.2.2 Itérateurs

L'opération la plus fréquente appliquée à une collection est de la parcourir, totalement ou partiellement. Pour cela, la STL propose une interface unique, un ***modèle abstrait*** auquel adhèrent toutes les collections *itérables*, *quelle que soit leur implémentation interne et donc le détail de leur parcours*. Sont itérables toutes les séquences et toutes les collections associatives du tableau précédent, mais *pas* les adaptateurs de collections.

Modèle abstrait des collections de la STL Ce modèle abstrait est le suivant : toute les collections itérables apparaissent (logiquement) et sont manipulables comme une liste doublement chainée de leur type de valeur. Cette liste à la structure indiquée sur la figure 3.1. Elle est

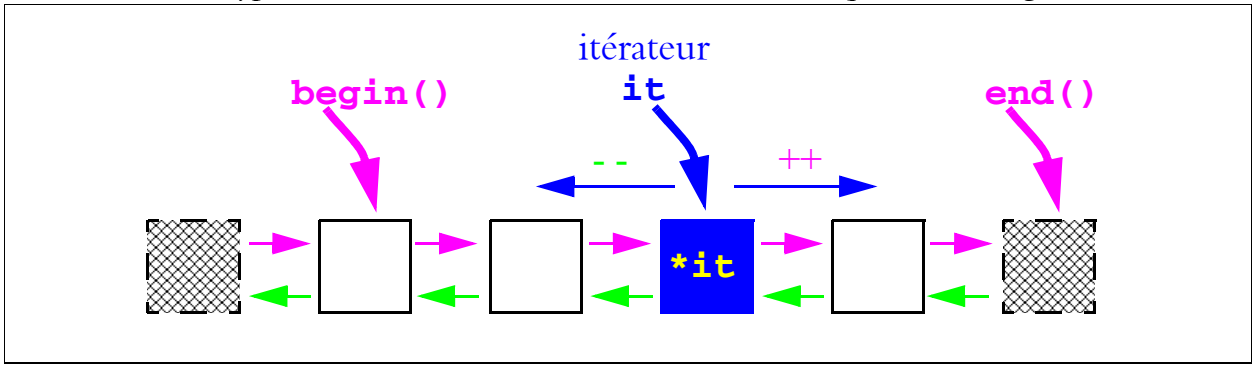

FIGURE 3.1 : Modèle abstrait des collections de la STL

constituée de cellules contenant chacune la valeur d'un élément et elle est bornée par deux « sentinelles » au début et à la fin (les sentinelles, hachurées sur la figure, ne font pas partie de la liste à proprement parler et ne contiennent donc pas de valeur utile). La liste du modèle abstrait respecte évidemment l'ordre de la collection de base, que ce soit celui d'insertion ou celui intrinsèque des éléments pour les collections ordonnées (c'est-à-dire les associatives `map`, `set`, `multimap` et `multiset`).

Itérateurs Pour permettre le parcours d'une collection à travers son modèle abstrait, la STL fournit, pour chaque type de collection, un certain nombre de types d'itérateurs, par exemple :

```
list<double> l;                 // une liste de réels
list<double>::iterator it;      // un itérateur sur liste de réels
```

Notez qu'un itérateur comme `it` n'est pas directement lié à une collection au moment de sa création. C'est en fait comme *une sorte de pointeur*[1] qui pointerait sur un élément de la collection : ainsi `it` ressemble-t-il à un `double *`.

Toutes les collections itérables de la STL fournissent deux fonctions-membres, `begin()` et `end()`, qui retournent respectivement un itérateur sur le début (i.e., la première cellule utile) et la fin (i.e., la sentinelle finale) de la liste du modèle abstrait. L'opérateur `++`[2] appliqué à un itérateur permet de le « faire pointer » sur la cellule suivante, et `--` permet d'aller à l'élément précédent. Enfin, l'opérateur unaire `*` (`*it`) fournit une référence sur l'élément « pointé » par l'itérateur.

À titre d'exemple voici une boucle (tout à fait canonique) qui imprime tous les éléments de la liste de réels précédente en la parcourant grâce à l'itérateur `it` :

```
for (it = l.begin();      // it pointe sur le début de la liste
     it != l.end();       // tant que l'on n'est pas sur la sentinelle finale
     ++it)                // aller à la cellule suivante
{
    cout << *it << endl;// imprimer l'élément courant
}
```

Quand on parcourt une `map`, il ne faut pas oublier que le type de valeur est une *paire* ; par exemple :

```
map<int, double> a_map;
// ...
for (map<int, double>::iterator it = a_map.begin();
     it != a_map.end(); ++it)
{
    cout << it->first << " " << it->second << endl;
}
```

L'opérateur `->` est bien sûr défini pour les itérateurs, et respecte l'identité habituelle

```
it->first ≡ (*it).first
```

Différents types d'itérateurs La STL dispose d'une riche classification des itérateurs. Ici nous nous contenterons de noter qu'il existe deux grands types d'itérateurs pour chaque collection :

- `const_iterator` : ces itérateurs ne permettent pas de modifier l'élément pointé par l'itérateur ; ainsi après
  ```
  list<double>::const_iterator cit;
  ```
  l'affectation
  ```
  *cit = 12
  ```
  est-elle incorrecte (ne compile pas) ;
- `iterator` : ces itérateurs permettent de modifier l'élément pointé.

Fonctions-membres principales Toutes les collections sont constructibles par défaut et copiables (copie des éléments à l'initialisation ou l'affectation). Elles disposent toutes d'un certain nombre de fonctions-membres communes dont les principales sont rappelées dans le tableau suivant (voir cours, section 5.7.3.2). Dans ce tableau *VT* désigne le type de valeur de la collection (noté `T` en général, mais rappelez-vous que c'est une paire dans le cas des maps) et

1. Ce n'est pas, en général, un vrai pointeur ; on parle donc souvent dans ce cas de « pointeur habile » (*smart pointer*).
2. Il en existe évidemment une forme préfixe et une forme postfixe.

IT désigne le type itérateur ; enfin, *container* désigne bien entendu le nom du type de collection considéré.

Interface pour les itérateurs	
`IT begin()` `IT end()`	retourne un itérateur sur le début ou la fin de la collection (voir ci-après)
Accesseurs	
`VT& front()` `VT& last()`	retourne une référence sur le premier ou le dernier élément de la collection
`VT& at(unsigned i)` `VT& operator[](unsigned i)`	indexation avec ou sans vérification (voir 2.3.3 : *seulement disponible pour* `vector` *et* `deque`.
Insertions	
`void push_front(const VT& e)` `void push_back(const VT& e)`	insère un élément au début ou à la fin *(disponible pour les séquences seulement)*
`void pop_front()` `void pop_back()`	retire le premier ou le dernier élément
`insert(IT pos, const VT& e)`	insère la valeur `e` devant l'élément « pointé » par `pos`
Operations globales	

`void clear()`
`void erase(IT pos)`
`void erase(IT beg, IT end)`
	détruit toutes les cellules (la collection (re)devient vide)
	détruit la cellule « pointée » par `pos`
	détruit toutes les cellules entre `beg` (inclus) et `end` (exclus)

`unsigned size()` `bool empty()`	nombre d'éléments dans la collection la collection est-elle vide ?
`bool operator==` et `!=  <  <=  >  >=`	égalité et plus généralement opérateurs relationnels (comparaison lexicale, *VT* doit posséder l'opérateur correspondant)
Constructeurrs et destructeur	
`container()`	constructeur par défaut (collection vide)
`container(int n)`	collection avec `n` elements initialisés au zéro de leur type (`VT()`)
`container(int n, const VT& x)`	collection avec `n` éléments tous initialisés à `x`
`container(IT first, IT last)`	copie de la collection itérée par [`first`, `last`[(les deux collections doivent avoir le mêm type de valeur)
`~container()`	destructeur : détruit toutes les cellules
Opérations de copie	

`constainer(const container&)`
`container& operator=(`
`const container&)`
	cosntructeur de copie
	affectation de copie

Fonctions-membres spécifiques des maps La classe `map` dispose (entr'autres) de deux fonctions spécifiques :

- La fonction-membre `find` est telle que, pour une `map` `m`, `m.find(k)` retourne un itérateur sur la cellule (unique par définition de la `map`) contenant la valeur de la clé `k` passée en paramètre ; si cette cellule n'existe pas, l'itérateur `m.end()` est retourné.
- L'opérateur d'indexation (`operator[]`) permet de considérer une `map<K, T>` comme un tableau de `T` indexé par des valeurs de type `K`. Les éléments de ce tableau s'allouent automatiquement sous l'effet de cet opérateur, et la valeur de `T` associée est initialisée à 0. Ainsi l'expression `m[k]` retourne toujours une référence sur un `T` (c'est-à-dire un `T&`) qui contient soit la valeur précédemment associée à la clé `k`, soit le zéro[1] du type `T` si l'opérateur d'indexation a dû allouer une nouvelle entrée pour `k`.

Voici un exemple d'utilisation de ces deux fonctions-membres:

```
map<string, string> months;      // map initialement vide
map<string, string>::iterator mit;
months["janvier"] = "january";   // alloue une nouvelle cellule
months["février"] = "February"; // alloue une nouvelle cellule
// etc.
mit = months.find("janvier");
assert(mit != months.end());     // vrai, "janvier" est une clé existante
mit->second = "January";         // corrige l'entrée correspondante
mit = months.find("Januar");
assert(mit == months.end());     // vrai, "Januar" n'est pas une clé existante
cout << months["Januar"];        // alloue une nouvelle cellule ...
                                 // et imprime la chaîne vide !
mit = months.find("Januar");
assert(mit != months.end());     // vrai, "Januar" est maintenant existante
```

3.2.3 Algorithmes

La STL fournit sous le nom d'*algorithmes* une soixantaine de fonctions génériques (*templates*) réalisant en un seul appel une opération sur tous les éléments d'une (ou plusieurs) collection(s). Pour utiliser les algorithmes, il vous faut inclure le fichier d'en-tête `<algorithm>`.

La collection sur laquelle opère un algorithme est fournie par deux itérateurs, disons `itf` et `itl`, qui définissent le début (inclus) et la fin (exclue) de la collection de travail (on désignera ceci par le terme « intervalle d'itération »). Par exemple, pour une liste `l`, `l.begin()` et `l.end()`. Bien sûr, `itf` et `itl` doivent pointer sur des éléments de la même collection.

Voici un exemple. L'algorithme `find` prend trois paramètres : les deux premiers constituent l'intervalle d'itération, et le dernier est une valeur dont le type est le même que celui des valeurs de la collection ; il retourne un itérateur sur le premier élément qui a la valeur indiqué ; si un tel élément n'existe pas, il retourne la borne supérieure de l'intervalle d'itération. Quand à l'algorithme `copy`, il copie les valeurs de tous les éléments d'un intervalle d'itération vers une autre collection, dont le début est fourni par l'itérateur qui constitue sont dernier paramètre.

```
list<int> l;
// ...
// recherche de la première occurrence de la valeur 5
```

1. Le zéro d'un type est ce que rend son constructeur par défaut. Pour les entiers c'est 0, pour un réel 0.0, pour un pointeur, le pointeur nul, pour une `string` la chaîne vide, etc.

```
list<int>::iterator itf = find(l.begin(), l.end(), 5);
if (itf == l.end())
    cerr << "première valeur non trouvée" << endl;
// recherche de la première occurrence de la valeur 10 à partir de itf
list<int>::iterator itl = find(itf, l.end(), 10);
if (itl == l.end())
    cerr << "deuxième valeur non trouvée" << endl;
else
{
    // copie de toutes les cellules entre les deux itérateurs précédents dans un vecteur
    // notez que si itf est l.end() rien n'est copié (l'intervalle est vide)
    vector<int> vec(l.size());
    copy(itf, itl, vec.begin());
}
```

Bien entendu, le vecteur et la liste doivent avoir le même type de valeur pour que `copy` fonctionne correctement. Notez aussi que la collection destinatrice doit être correctement dimensionnée : en effet, aucun algorithme de la STL ne modifie le nombre d'éléments d'une collection[1]. Ici la dimension de `vec` est la même que celle de `l`, donc plus grande que nécessaire en général[2].

Certains algorithmes prennent des fonctions en paramètres. Ces fonctions seront appliquées sur chacun des éléments de la collection, et donc doivent avoir un unique paramètre dont le type est celui des valeur de la collection (pour éviter des ennuis avec certaines réalisations de la STL, passez toujours ce paramètre par *valeur*). Voici un exemple avec l'algorithme `for_each` qui applique une fonction (son troisième argument) aux éléments de son intervalle d'itération :

```
list<double> l;
// ...
// cette fonction sera appliquée sur chaque élément de la liste
void print(double)
{
    cout << x << endl;
}
//...
for_each(l.begin(), l.end(), print);
```

Voici un exemple d'algorithme prennant une fonction à retour booléen, `count_if`. Il compte le nombre d'éléments dans l'intervalle d'itération qui satisfont la fonction booléenne :

```
list<double> l;
// ...
// une fonctionà résultat booléen, applicable sur chaque élément de la liste
bool square_is_gt_10(double x)
{
    return x * x > 10.0;
```

1. Nous verrons, à la fin du cours, des moyens plus élégants de dimensionner dynamiquement la collection cible.
2. Quand aux petits malins qui, essayant de pousser un peu trop loin l'analogie entre pointeurs et itérateurs, penseraient que l'expression `itl - itf` est le nombre d'éléments à copier, qu'ils sachent que ce serait faux ici (cela ne compilerait même pas) ! Les itérateurs n'ont pas nécessairement d'opérations arithmétiques, à part l'incrémentation et la décrémentation (`++` et `--`).

```
}
int n = count_if(l.begin(), l.end(), square_is_gt_10);
```

Ici n sera donc le nombre d'éléments de la liste dont le carré est supérieur à 10. (Pour ceux d'entre vous qui se demanderaient comment faire `square_is_gt_y`, pour *y* quelconque donné en paramètre, il faudra attendre les dernières séances du cours...).

3.3 ENCORE DES COMPLÉMENTS SUR LES STRINGS

On vous demande de réaliser la fonction de « tokenization » dont le prototype figure également dans le fichier fourni `extra_string.h`. La tokenization consiste à découper une chaîne de caractères en sous-chaines en utilisant certains caractères comme séparateurs. La tokenization la plus courante consiste à découper une chaîne en mots en prennant les espaces et la fin de ligne comme séparateurs. Ainsi la chaîne

```
"Le       petit   chat\n\nest  mort"
```

une fois découpée de cette manière rend 5 sous-chaînes (5 mots) :

```
"Le", "petit", "chat", "est", "mort."
```

On voit donc que l'on part d'une chaîne et l'on obtient une collection de chaînes. N'importe quelle séquence de la STL peut contenir le résultat, d'où le prototype désiré pour la fonction `tokenize` :

```
template <typename SeqContainer>
unsigned tokenize(const string& s, SeqContainer& seq,
                  const string& delims = " \t\n\f\v")
```

découpe la chaîne `s` en sous-chaînes selon les séparateurs de `delims` et ajoute les sous-chaînes à la suite du container `seq` ; on supposera que `seq` (et son type `SeqContainer`) est une *séquence*, c'est-a-dire une `list`, une `deque` ou un `vector` de la STL ; si plusieurs séparateurs sont consécutifs, ils comptent pour un seul; la fonction retourne le nombre de sous-chaînes ainsi obtenues.

En utilisant les algorithmes de la STL sur `string` (les `string` ont toutes les propriétés des séquences commes `vector`) définissez également (dans `extra_string.cpp`) les deux dernières fonctions prévues, à savoir `string_to_upper` et `string_to_lower`. **Attention** : la fonction `toupper` de C (définie dans `<cctype>`) est en fait une macro ! Pour avoir une *vraie* fonction en C++, il faut écrire `::toupper`. Même remarque pour `tolower` qui devient `::tolower`.

3.4 CONSTITUTION DE L'INDEX D'UN TEXTE

On veut constituer l'index par ligne d'un texte, c'est à dire pour chaque mot du texte, donner la suite ordonnée des numéros de lignes qui contiennent ce mot. Ainsi, avec le fichier de 3 lignes de texte suivant :

```
to be
or not
to be
```

on doit obtenir l'index :

```
be: 1 3
not: 2
or: 2
to: 1 3
```

Les mots doivent être eux-mêmes triés dans l'ordre alphabétique, un par ligne, sans doublon. Faites cet exercice incrémentalement dans le fichier `index.cpp`.

1. Quelle structure de données (utilisant la STL) allez-vous prendre pour représenter votre index ?
2. Lisez un texte sur l'entrée standard (`cin`), ligne par ligne : le plus simple est d'utiliser la fonction `getline` de `<string>`[1] qui rend un résultat (convertible en) `false` en fin de fichier. Vous pouvez utiliser le fichier fourni `camille.txt` comme texte d'essai.
3. Effectuer une tokenization de la ligne obtenue à l'aide de votre fonction `tokenize` de l'exercice précédent (3.3) pour la découper en mot. Les séparateurs doivent être les espaces (blancs, fins de lignes, tabulations) ainsi que les caractères de ponctuations du Français.
4. Supprimer tous les mots qui ne comportent qu'un seul caractère.
5. Transformer tous les mots en minuscules.
6. Insérer les mots ainsi obtenusdans votre index.
7. En fin de fichier, imprimer l'index obtenu. Pour celà vous utilisereez la fonction template `print` définie dans le fichier fourni `stl_print.h` (et dont le test est dans `test_print.cpp`).

Vous ne devez utiliser de boucle explicite (`for` ou `while`) que pour les points 1 et 6. Pour le reste, vous devez utiliser des *algorithmes* de la STL ; bien entendu, pour cela, vous pourrez avoir à définir quelques fonctions annexes. Il vous faudra également consulter une documentation de la STL, par exemple celle de SILICON GRAPHICS, `www.sgi.com`, déjà mentionnée.

1. *pas* la fonction-membre `getline` de `istream`, beaucoup moins commode

TD 4
Une classe `Date`

4.1 OBJECTIF

Il s'agit là de la première classe C++ que vous allez définir complètement. Sa complexité est relativement faible, assez similaire à celle de la classe `Rational` vue en cours. Comme cette dernière, c'est une classe qui définit des opérateurs arithmétiques et dont le constructeur réalise une sorte de normalisation (au moins une vérification).

Pour vous éviter de perdre du temps à développer des algorithmes dont l'intérêt est en l'occurrence limité, une partie du code vous est fourni (dans le répertoire `Date`), mais il est écrit en C. Cet exercice constitue donc aussi une occasion de comparer la réalisation du même type abstrait en C et en C++.

4.2 DESCRIPTION DE LA CLASSE `Date`

Votre mission est de définir une classe `Date` représentant des dates du calendrier occidental et permettant de les manipuler (opérations arithmétiques, entrées-sorties). On se limitera aux années postérieures à 1900[1] (incluse).

La classe `Date` a un constructeur par défaut : la date créée est celle du jour courant (aujourd'hui) ; elle possède aussi un constructeur prenant trois entiers (année, mois, jour) :

```
Date dnow, d;            // la date courante (aujourd'hui)
Date d14J(2002, 7, 14); // le 14 juillet 2002
```

Bien entendu, ce dernier constructeur doit vérifier que les trois entiers proposés forment bien une date valide et, dans le cas contraire, lever des exceptions.

Vous définirez aussi des fonctions d'accès permettant de consulter l'année, le mois et le jour d'une date.

Vous devez également définit des opérateurs permettant d'ajouter ou de soustraire un nombre entier de jours à une date[2] :

```
d = dnow + 1;            // demain
d = dnow - 1;            // hier
d = dnow + 365;          // dans un an, aujourd'hui
```

On doit aussi pouvoir soustraire deux dates, ce qui retourne le nombre *algébrique* de jours entre les deux dates. Par définition *demain - aujourd'hui =1* et a*ujourd'hui - demain = -1*.

```
int n = d14J - dnow;     // jours entre le 14 juillet 2002 et aujourd'hui
n = (dnow + 1) - dnow;   // 1, par définition
n = dnow - (dnow + 1);   //-1
n = Date(2003, 2, 10) - Date(2003, 6, 12); //-122
```

Remarquer, dans la dernière ligne, comment l'appel explicite du constructeur permet de créer « à la volée » un objet *temporaire* de type `Date`.

Les dates doivent pouvoir être comparées avec les opérateurs relationnels habituels (`==`, `!=`, `<`, `>`, `<=`, `>=`).

1. Ceci à cause d'une limitation de la norme POSIX qui ne considère pas de date antérieure à 1900.
2. Avec un peu d'aide, il vous sera même possible de définir des opérateurs comme `+=`, `-=`, `++` et `--`.

Enfin les dates doivent supporter les habituels opérateurs d'entrée-sortie des `iostreams`. L'opérateur d'affichage (<<) doit produire une forme lisible de la date. Ainsi:

```
cout << d14J << endl;
```

doit afficher sur la sortie standard

```
July 14, 2002
```

(ou son équivalent français). L'opérateur de lecture (>>), quant à lui, reçoit des dates composées d'entiers sous la forme *yyyy/mm/dd*. Il doit bien entendu refuser de lire des dates invalides.

4.3 NOTE SUR LE CODE C FOURNI

Le répertoire `Date` contient le code C réalisant un type abstrait date assez rustique. On trouvera trois fichiers de source C : `date_c.h`, la définition de la structure date, `date_c.c`, son implémentation, et `main_date_c.c`, un programme de test. Une `Makefile` permet de reconstruire l'exécutable de test (faire `make tst_date_c.exe`). Cette `Makefile` est également prête à accueillir le code que vous développerez, à condition que le fichier de définition de votre classe `Date` se nomme `Date.h`, son implémentation `Date.cpp`, et le programme de test `main_Date.cpp`. Il suffira alors de faire `make` et l'exécutable produit se nommera `tst_Date.exe`. Si vous ne respectez pas ces noms, il vous faudra éditer la `Makefile`.

Le code de la `struct date` vise juste à fournir les algorithmes pour calculer la date du jour et ajouter ou soustraire un nombre de jours à une date. Vous devrez inventer celui pour soustraire deux dates. Essayez de le faire économiquement (en code) et avec la même approche que celle utilisée dans le code fourni

Ce code C n'est certes pas le code définitif d'un vrai type abstrait date. En particulier aucune vérification n'est effectuée. Ceci dit, les vérifications nécessaires sont plus difficiles à imposer en C qu'en C++. Ceci est aussi une leçon à tirer de l'exercice.

Noter enfin que le code C (et sans doute votre code C++) nécessite a priori un système compatible avec la norme POSIX (Unix, Linux, Cygwin...) pour trouver la date du jour (fonction POSIX `localtime()` utilisée dans `today()`). Cependant, la plupart des environnements C, même non POSIX, en particulier Visual C++ supporte la fonction `localtime()`.

TD 5
Vecteurs et matrices

5.1 OBJECTIF

L'objectif de cette séance est de réaliser deux classes représentant (au moins partiellement) les vecteurs et les matrices de l'algèbre linéaire.

Ceci fournit l'occasion d'utiliser à nouveau la classe `vector` de la STL, d'améliorer la maîtrise de la construction et de la copie d'objets complexes, et d'aborder la redéfinition d'opérateurs un peu plus exotiques, comme l'indexation ou l'appel de fonction.

5.2 VECTEURS MATHÉMATIQUES : CLASSE `MVector`

5.2.1 Spécification

Votre mission est de définir une classe `MVector` représentant des vecteurs mathématiques de nombres réels (`double`) avec quelques unes de leurs opérations habituelles. Un tel objet devra être constructible en donnant sa dimension et éventuellement une valeur d'initialisation commune pour toutes les composantes :

```
MVector mv1(10, 3.5);    // vecteur de 10 éléments, tous initialisés à 3.5
MVector mv2(20);         // vecteur de 20 éléments, tous initialisés à 0.0
```

Il doit également exister un constructeur par défaut :

```
MVector mv3;             // vecteur de dimension 0
```

Voici une liste (non exhaustive, voir plus bas) des opérations qui doivent être possibles sur les `MVectors` :

- copie à l'initialisation comme à l'affectation, avec ajustement automatique de la dimension ;
- addition et soustraction de deux vecteurs de même dimension ;
- produit scalaire de deux vecteurs de même dimension ;
- comparaison (égalité et inégalité) de deux vecteurs ;
- indexation d'un vecteur constant ou non grâce à `operator[]` ;
- affichage du contenu d'un vecteur grâce à `operator<<`.

Le fichier `main_MVector.cpp` du répertoire `Linear_Algebra` contient un programme principal de test des fonctionnalités attendues de `MVector`. Il se peut que vous ayez à écrire quelques fonctions supplémentaires par rapport à la liste précédente pour pouvoir exécuter ce programme. Nous vous laissons les déterminer.

Quant une opération apparaît impossible, une exception soigneusement choisie doit être lancée.

5.2.2 Réalisation

Vous réaliserez vos `MVector` en utilisant de manière interne un `vector<double>` de la STL pour contenir les composants.

Vous devez éviter les implémentations inutiles. Essayez au maximum de déléguer les opérations de `MVector` aux opérations correspondantes de `vector`. N'implémentez pas les fonc-

tions automatiquement fournies par C++ (constructeur par défaut et de copie, affectation de copie, destructeur) lorsque la définition par défaut vous convient.

5.3 MATRICES RECTANGULAIRES : CLASSE `Matrix`

5.3.1 Spécification

Vous allez utiliser cette classe `MVector` pour réaliser la classe `Matrix` représentant des matrices, tableaux bidimensionnels (rectangulaires) de réels (double) avec quelques unes des opérations habituelles.

Une matrice se construit en passant ses deux dimensions et éventuellement une valeur d'initialisation commune pour toutes les composantes :

```
Matrix mat1(10,20); // une matrice 10x20, toutes composantes égales à 0.0
Matrix mat2(5, 5, 3.0); // une matrice 5x5, toutes composantes égales à 3.0
```

Il doit également exister un constructeur par défaut :

```
Matrix mat3;                    // matrice de dimension 0x0
```

Voici une liste (non exhaustive, voir plus bas) des opérations qui doivent être possibles sur les `Matrix`s :

- copie à l'initialisation comme à l'affectation, avec ajustement automatique de la dimension ;
- addition et soustraction de deux matrices de mêmes dimensions ;
- produit de deux matrices de dimensions convenables (le nombre de colonnes de la première doit être égal au nombre de lignes de la seconde) ;
- transposition d'une matrice (la méthode `transpose()` ou, si vous le souhaitez, `operator~`, doit retourner une copie transposée de la matrice de départ) ;
- comparaison (égalité et inégalité) de deux matrices ;
- indexation des matrices constantes et non constantes (voir 5.3.3 ci-dessous) ;
- affichage du contenu d'une matrice grâce à `operator<<` ;
- conversion d'un `MVector` en matrice-ligne ;
- éventuellement, conversion d'une `Matrix` en `MVector` pourvu que la matrice de départ soit une matrice-ligne.

Le fichier `main_Matrix.cpp` du répertoire `Linear_Algebra` contient un programme principal de test des fonctionnalités attendues de `Matrix`. Il se peut que vous ayez à écrire quelques fonctions supplémentaires par rapport à la liste précédente pour pouvoir exécuter ce programme. Nous vous laissons les déterminer.

Quant une opération apparaît impossible, une exception soigneusement choisie doit être lancée.

5.3.2 Réalisation

Vous réaliserez vos `MVector` en utilisant de manière interne un vecteur de la STL dont les composantes sont des `MVectors` (`vector<MVector>`) .

Vous devez éviter les implémentations inutiles. Essayez au maximum de déléguer les opérations de `Matrix` aux opérations correspondantes de `MVector`. N'implémentez pas les fonctions automatiquement fournies par C++ (constructeur par défaut et de copie, affectation de copie, destructeur) lorsque la définition par défaut vous convient.

5.3.3 Note sur les opérations d'indexation

En C++, il n'est pas possible de définir un opérateur d'indexation `operator[]` qui permette d'indexer une matrice en écrivant `mat[i, j]` (Pourquoi donc, à votre avis ?)

En revanche, l'implémentation proposée devrait vous permettre sans trop de peine d'écrire un opérateur d'indexation autorisant l'écriture `mat[i]`, qui retourne une référence sur la ligne d'indice `i` de la matrice (un `MVector`). Alors l'écriture `mat[i][j]` devient elle aussi légale (et adéquate !), n'est-ce-pas ? Il est même possible d'écrire `mat(i, j)` (noter les parenthèses *rondes*) en définissant l'opérateur *appel de fonction* (dont le nom est, bien évidemment, `operator()`).

Vous devrez également définir deux opérations qui entrent dans la catégorie des indexations :

- l'opération `line()`, telle que `mat.line(i)` retourne une copie de la ligne (le `MVector`) d'indice `i` de la matrice `mat` (noter que c'est différent de `mat[i]` évoqué juste au-dessus) ;
- l'opération `column()`, telle que `mat.column(j)` retourne une copie de la colonne d'indice `j` (un `MVector`) de la matrice `mat`.

5.4 REMARQUES ET CONSEILS

5.4.1 Remarque sur le code fourni

Le code est dans le répertoire `Linear_Algebra`, à l'endroit habituel. Il n'est utilisable que si la hiérarchie établie dans les séances de TD précédents est respectée : en particulier, vérifiez que vous avez bien le fichier `default.mk` et le répertoire `include` (avec son contenu !) deux niveaux de répertoires au-dessus de votre répertoire de travail (`../../default.mk` et `../../include`, donc).

Le répertoire `Linear_Algebra` contient le source des deux programmes de test déjà mentionnés (`main_MVector.cpp` et `main_Matrix.cpp` qui donneront les exécutables `tst_MVector.exe` et `tst_Matrix.exe`) ainsi que la trace de leur exécution avec ma propre solution (`tst_MVector.out` et `tst_Matrix.out`).

La `Makefile` elle-même est prête à accueillir votre travail et à le tester avec les deux programmes de test fournis, à condition que vous respectiez le nommage des fichiers : vous devez fournir 4 fichiers sources nommés `MVector.h` (la définition de la classe `MVector`), `MVector.cpp` (son implémentation), `Matrix.h` et `Matrix.cpp` (la même chose pour la classe `Matrix`). Bien entendu, vous devez aussi respecter les noms des classes et des méthodes proposés dans ce sujet. Il suffira alors de faire `make` pour construire les deux exécutables ou encore, par exemple,

```
make tst_Matrix.exe
```

pour construire uniquement le second.

5.4.2 Conseil

L'exercice n'est pas particulièrement court. N'attendez-donc pas d'avoir écrit la totalité d'une classe et de ses méthodes pour commencer à la tester.

Essayer de travailler *incrémentalement*. Définissez les méthodes progressivement dans un ordre compatible avec les programmes de test : par exemple le(s) constructeur(s) est(sont) à définir en premier, puis l'opérateur d'affichage (`operator<<`) qui est indispensable à la macro `OUT` définie dans `../../include/common_defs.h`, etc. Testez au fur et à mesure,

en commentant les parties du programme de test correspondant à des fonctionnalités non encore implémentées.

TD 6
Variations sur les matrices

6.1 OBJECTIF

Cet exercice prolonge le précédent (5) et nécessite d'en avoir réalisé la classe `Matrix`. Vous trouverez, à l'endroit habituel, dans le répertoire `Linear_Algebra`, la correction de la classe `MVector`, mais vous aurez à terminer la classe `Matrix`.

L'exercice illustre la dérivation de classes (héritage) en C++ et les problèmes de construction et de copie y afférents.

Cet exercice ne clôt pas la représentation des matrices en C++. On lira en particulier avec profit la section 6.4.3.

6.2 DIVERSES SORTES DE MATRICES

6.2.1 Matrices carrées

Une matrice carrée est une matrice, elle en a toutes les propriétés et tous les opérateurs. Elle a surement des propriétés supplémentaires (comme des valeurs propres par exemple) mais nous ne nous en préoccuperons pas ici.

On vous demande donc d'écrire une classe représentant des matrices carrées, nommée `SquareMatrix`, dérivant de `Matrix` et utilisant, dans la mesure du possible, les propriétés de cette dernière.

Vous apporterez une attention toute particulière aux opérateurs arithmétiques (`+`, `-`, `*`) et à leur typage : en particulier remarquez que la somme (la différence, le produit) de deux matrices carrées est une matrice carrée ; ou encore que le produit d'une matrice carrée par une matrice rectangulaire (de nombre de lignes convenable) n'est qu'une matrice rectangulaire[1].

6.2.2 Matrices diagonales

Les matrices diagonales sont un cas particulier de matrices carrées. On vous demande donc d'écrire une classe `DiagonalMatrix` les représentant. Elles doivent pouvoir, entre autres, se construire à l'aide d'un `MVector` qui sera la valeur de leur diagonale :

```
MVector mv(10, 1.0);// construit un vecteur de 10 composantes, toutes égales à 1
DiagonalMatrix dmat(mv);// construit la matrice identité 10x10
```

Ici encore, vous apporterez une attention toute particulière aux opérateurs arithmétiques (`+`, `-`, `*`) et à leur typage : en particulier remarquez que la somme (la différence, le produit) de deux matrices diagonales est une matrice diagonale, mais que la somme d'une matrice diagonale et d'une matrice carrée n'est qu'une matrice carrée[2]. Par ailleurs le calcul de la somme (différence, produit) de deux matrices diagonales peut être simplifié par rapport à celui des matrices ordinaires (rectangulaires ou carrées).

1. En tous cas, en ce qui concerne le *type*. Une matrice rectangulaire (`Matrix`) pouvant être carrée (« par hasard » à l'éxécution) la matrice résultat peut elle-même être de fait carrée ; il n'empêche que pour le compilateur, qui ignore ce qui va se passer à l'exécution, son type ne peut être que celui d'une matrice ordinaire (`Matrix`).
2. Même remarque que précédemment.

6.2.3 Matrices scalaires

Une matrice scalaire est une matrice diagonale dont tous les éléments diagonaux sont égaux (elle est de la forme $\lambda\mathbf{I}_n$ où $\mathbf{I}_n$ est la matrice identité de dimension n). Il s'agit donc d'un cas particulier de matrice diagonale. Une telle matrice doit pouvoir être construite, entre autres, à partir de sa dimension et de la valeur commune des éléments diagonaux :

```
int n = 10;
ScalarMatrix scmat(n, lambda); // construit λI_n
```

Ici encore la somme de deux matrices scalaires est scalaire, mais celle d'une matrice scalaire et d'une matrice diagonale n'est que diagonale, et celle d'une matrice scalaire et d'une matrice carrée n'est que carrée... Par ailleurs l'algorithme des opérations entre matrices scalaires est particulièrement simple par rapport à celui des autres types de matrices.

6.3 Question pour les meilleurs : métamorphoses des matrices

Si la hiérarchie que nous vous avez établie prend bien en compte un typage statique des matrices et de leurs opérations artihmétiques, elle ignore complètement le problème de la « métamorphose » des matrices. Cette métamorphose peut se présenter de deux manières :

1. Les opérateurs d'indexation (`operator[]`, `operator()` et fonctions-membres `at()`) peuvent changer la nature d'une matrice ; ainsi avec la matrice `dmat` précédente (qui était par ailleurs égale à la matrice identité de dimension 10), l'affectation
   ```
   dmat(2, 3) = 5.0
   ```
   lui fait perdre sa nature diagonale !
2. Par ailleurs, on pourrait souhaiter définir des conversions entre les différents types de matrices quand leur structure interne le permet ; ainsi si l'on a
   ```
   Matrix mat(5, 5);
   ```
   on pourrait souhaiter convertir `mat` en une matrice carrée (`SquareMatrix` de dimension 5) ou même diagonale (`DiagonalMatrix` de diagonale nulle) ou scalaire (`ScalarMatrix` de valeur 0.0) !

Le premier point est le plus crucial (si on ne fait rien, c'est un vrai bug !) mais aussi le plus délicat à résoudre. Une solution brutale serait d'interdire l'indexation pour *modifier* un élément (mais de conserver la possibilité d'indexation pour *lire* la valeur) : comment réaliseriez-vous cela ? *Évidemment une telle solution serait un pis-aller* et, de toutes façons, si vous l'essayez vous constaterez que cela casse pas mal de choses ! On peut imaginer des solutions plus satisfaisantes : par exemple, interdire la modification d'une matrice diagonale en dehors de sa diagonale ; ou encore, changer dynamiquement le type d'une matrice diagonale lorsque l'on crée un élément non nul en dehors de la diagonale... Ces solutions sont, pour l'instant, hors de votre portée (voir cependant 6.4.3). Si vous avez une idée, exprimez la, mais n'essayez pas de la réaliser tout de suite.

En ce qui concerne le second point, il est un peu plus facile d'autoriser certaines conversions. Essayez en particulier d'écrire un constructeur de la classe `SquareMatrix` qui permette de convertir (implicitement) une matrice ordinaire en matrice carrée si la matrice ordinaire est effectivement carrée et qui, sinon, lève une exception. Que deviennent alors les expressions de la forme `sqmat + mat` où `sqmat` est une matrice carrée et `mat` une matrice ordinaire ? Est-ce normal ? Préfixez la déclaration de votre constructeur par le mot clé `explicit`, cela devrait régler le problème (mais cela vous forcera à expliciter la conversion par un *cast* (`static_cast`).

6.4 REMARQUES ET CONSEILS

6.4.1 Code fourni

Le répertoire `Linear_Algebra_2`, à l'endroit habituel contient un exemple de programme de test (`main_SquareMatrix.cpp`) des différents types de matrices. Il n'est utilisable (ainsi que la `Makefile`)

- que si la hiérarchie de fichiers établie dans les séances de TD précédents est respectée : en particulier, vérifiez que vous avez bien le fichier `default.mk` et le répertoire `include` (avec son contenu !) deux niveaux de répertoires au-dessus de votre répertoire de travail (`../../default.mk` et `../../include`, donc) ;
- qui si vous respectez les noms de classes indiqués dans ce sujet ainsi que l'organisation des fichiers indiquée plus bas (sinon vous devrez éditer la `Makefile` et le programme de test)..

Le fichiers `tst_SquareMatrix.out` contient le résultat de l'exécution du programme de test avec ma propre solution.

6.4.2 Conseils

Méthode de développement Les mêmes remarques que lors des TDs précédents s'appliquent. Essayer de travailler *incrémentalement*. Définissez les méthodes et les classes progressivement, dans un ordre compatible avec les programmes de test. Testez au fur et à mesure, en commentant les parties du programme de test correspondant à des fonctionnalités non encore implémentées.

Par ailleurs ne définissez pas les fonctions automatiquement synthétisées par C++ si cette définition automatique vous agrée et évitez les duplications de code.

Organisation des fichiers Prenez de bonnes habitudes ! Bien que cela ne soit en rien obligatoire en C++, on place en général une seule classe C++ par « module ». Un module est constitué (ici aussi, *en général*) de deux fichiers sources dont le nom de base est celui de la classe : un fichier d'entête (d'extension `.h`) qui contient la définition de la classe, un fichier `.cpp` qui contient la définition du corps des fonctions membres et amies de la classe. Bien entendu, le fichier `.cpp` n'est pas nécessaire si l'ensemble de la définition de la classe et de ses fonctions peut-être placé dans le fichier `.h` : les fonctions sont alors `inline` et ceci doit être réservé à des fonctions simples et courtes, pour lesquelles il n'apparait pas souhaitable de payer le coût d'une séquence d'appel et de retour de fonction.

Inspirez-vous de la `Makefile` fournie pour avoir une idée des fichiers que vous avez à produire.

6.4.3 Limites de l'exercice

Ces différentes classes de matrices peuvent ne pas apparaitre complètement satisfaisantes à un programmeur rigoureux et soucieux aussi bien d'abstraction que d'efficacité (c'est à dire un vrai programmeur, en C++ ou tout autre langage !).

Du côté positif, la hiérarchie de classes que nous avons constituée reflète bien la relation logique (abstraite) entre les différentes sortes de matrices et assure le typage correcte des différentes opérations. Elle n'est évidemment pas complète, ni en ce qui concerne les types possibles (il y a bien d'autres sortes de matrices, creuses, bandes, symétriques, triangulaires, etc.), ni en ce qui concerne les opérations (calcul du rang, déterminant, inversion, calcul de valeur propres, triangulation, diagonalisation). Il parait cependant assez facile d'ajouter ces propriétés.

Le plus grand problème est certainement celui de l'efficacité, en temps (trop de copies ?) et surtout, en mémoire. En effet, le stockage des éléments de nos matrices ne dépend pas du type de ces dernières : elles héritent toutes leur implémentation de la classe `Matrix`, et donc allouent $m \times n$ éléments (ou n^2 si elles sont carrées) quelle que soit leur nature. Cela a l'avantage de faciliter les conversion en montant dans la hiérarchie, par exemple de `DiagonalMatrix` à `SquareMatrix`. Cependant, on pourrait souhaiter n'allouer que les éléments nécessaires (une demie matrice pour une matrice triangulaire ou symétrique, un vecteur pour une matrice diagonale, un simple `double` pour une matrice scalaire, etc.). Mais alors les conversions implicites d'héritage ne fontionneraient plus aussi bien, ce qui veut dire qu'il nous faudrait réviser complètement notre hiérarchie de classes ! Et il ne faudrait pas que cette réorganisation nous fasse perdre la vision logique de la hiérarchie des types de matrices...

Nous reviendrons sur cette question (et sur l'exemple des matrices) dans la seconde partie du cours, lorsque nous aborderons l'étude des schémas de conception (*Design Patterns*). Nous essayerons alors de donner une solution plus satisfaisante tout en conservant les propriétés abstraite de nos objets. Il nous faudra alors apporter également une réponse aux questions importantes soulevées en 6.3.

TD 7
Opérations de copie

7.1 Objectif

Cette série d'exercices vise à montrer l'importance qui doit être accordée à la définition correcte des opérations de copie en C++ : constructeur de copie et opérateur de copie à l'affectation.

7.2 Synthèse automatique des opérations de copie par C++

On considère la classe `A` dont la définition complèete est la suivante (fichier `Copy/A.h`) :

```
class A {
private:
    string _s;
    int _i;
public:
    A(const string& s = "", int i = 0)
    {
        _s = s;
        _i = i;
    }
};
```

Est-ce que les instructions suivantes (fichier `Copy/main_A.cpp`) compilent correctement ? Si oui, pourquoi ? Si non, pourquoi ?

```
A a1;
A a2("hello");
A a3 = a2;
a1 = a2;
string s = "bonjour";
a2 = s ;
a2 = "bonjour";
```

On modifie légèerement la définition des membres de données de la classe A :

```
class A {
private:
    const string _s;
    int _i;
    //...
};
```

et on demande de répondre aux mêmes questions.

7.3 Importance des opérations de copie

Dans le répertoire `List`, on trouvera dans les fichiers `List.h` et `List.cpp` la définition complète d'une liste simplement chainée d'entiers. Cette liste possède les trois opérations de base `append()` (ajouter un entier à la fin de la liste), `prepend()` (ajouter un entier au

début) et `get_first()` (retirer le premier élément de la liste et retourner sa valeur). Le contenu de la liste est également imprimable grâce au maintenant habituel `operator<<`. Enfin, le fichier `main_List.cpp` fournit un programme de test de la liste.

7.3.1 Expérimentation avec les destructeurs et les copies

1. Copiez le répertoire `List` chez vous et allez y (`cd List`). Vérifiez que vous avez bien le fichier `default.mk` et le répertoire `include` deux niveaux de répertoires au-dessus (`../../default.mk` et `../../include`, donc) ; cette structure de répertoires était déja utile pour les précédents TDs.
2. Compilez le programme de test (il suffit de faire `make`) et exécutez le (l"exécutable se nomme `tst_List.exe`). Tout doit bien se passer.
3. La classe `List` fournie n'a pas de destructeur. Modifiez la pour lui en donner un, qui détruit toutes les cellules de la liste. Compilez et exécutez le programme de test. Tout devrait continuer à bien se passer.
4. Instrumentez votre destructeur pour déterminer combien de fois il est invoqué par le programme de test. Justifiez le résultat.
5. Éditez le fichier `main_List.cpp` et décommentez les 4 lignes précédées par
   // First, uncomment the following 4 lines
   Compilez et exécutez à nouveau. Il se peut que le programme crashe. Même si ce n'est pas le cas, le résultat est-il alors correct ?
6. Commentez à nouveau les 4 lignes précédentes (pour éviter de crasher de la même manière) et décommentez la ligne précédée par
   // Second, uncomment the next 3 lines
   Compilez et exécutez à nouveau. Il se peut que le programme crashe une seconde fois. Même si ce n'est pas le cas, le résultat est-il alors correct ? Au passsage, combien de fois est appelé le destructeur dans ce cas ?
7. Analysez les causes des deux crashes précédents.
8. Modifier en conséquence la définition de la classe `List` pour que le programme de test ne crashe plus (une fois les deux zones décommentées).

Note Dans tout cet exercice, vous ne devez pas modifier le fichier `main_List.cpp` autrement que pour commenter et décommenter les zones indiquées.

7.3.2 Pour ne pas perdre la main...

En vous inspirant de ce que nous avons fait lors des premiers TDs avec les classes `Stack` et `Fifo`, transformez la classe `List` pour qu'elle devienne générique (*template*) : au lieu d'une liste d'entiers, on définit donc une liste de « n'importe quoi », pourvu qu'il soit copiable.

Attention Dans cette partie de l'exercice, il y a quelques subtilités techniques. Tout d'abord, rappelez-vous que, dans le cas des classes templates, la différence entre `.h` et `.cpp` devient illusoire : soit on met tout dans le `.h`, soit on inclut le `.cpp` à la fin du `.h`, et on modifie la `Makefile` afin de ne pas compiler séparément le `.cpp` (voyez comment cela est fait dans `Stack`). Ensuite, il se peut que vous ayez besoin d'aide pour traiter correctement la fonction amie `operator<<`.

TD 8
Variations sur les listes, les files et les piles

8.1 OBJECTIF

Jusqu'à présent, nous avons introduit l'héritage comme une manière de réaliser la relation *est-un* : un `Item_Paragraph` *est un* `Paragraph`, il en exhibe les propriétés, il en possède l'interface, il peut lui être *substitué* dans tout contexte. Pour ces raisons, cette relation est aussi appelée *sous-typage* : `Item_Paragraph` est un sous-type de `Paragraph`. Elle est réalisée en C++ par l'héritage public, qui valide la conversion d'héritage dans tout contexte.

L'héritage public de C++ consiste donc essentiellement en un *héritage de l'interface* de la classe. Mais, bien entendu, il s'accompagne aussi de l'héritage d'une partie du code de la classe de base : les membres de données en sont hérités et aussi le code de toutes les fonctions-membres.

Cet exercice introduit une nouvelle manière d'utiliser l'héritage afin d'assurer le partage (ou la réutilisation) de code sans hériter de l'interface. Cette technique est particulièrement utile dans la conception de bibliothèques, où l'accent est essentiellement placé sur la réutilisation et l'évolutivité.

Au passage, cet exercice permet aussi de revoir les classes génériques, et d'illustrer leur comportement en cas d'héritage (ce comportement ne présente d'ailleurs aucun mystère !). Il permet aussi, vers la fin, un retour sur le mécanisme de fonction virtuelle.

8.2 UNE SIMPLE CLASSE LISTE GÉNÉRIQUE

On trouvera, dans le répertoire `List_Queue_Stack`, la réalisation d'une classe générique `List<ELEM>` (`ELEM` est le type des éléments). En fait c'est exactement celle que vous avez du réaliser dans le TD 7 (plus précisément en 7.3.2). Le répertoire comporte quatre fichiers : `List.h`, la définition de la classe générique `List<ELEM>` ; `List.cpp`, la définition du corps de ses fonctions-membres et amies ; `main_List.cpp`, un programme de test ; et une `Makefile` (que vous aurez à compléter pour la suite de cet exercice). Notez que le fichier `List.cpp` est inclus à la fin de `List.h` et non pas compilé séparément (la `Makefile` ne génère pas de `List.o`). Rappelez-vous que c'est ainsi qu'il convient de faire dans le cas des classes *template* (voir 1.3).

La liste proposée à la structure décrite à la figure 8.1. La classe `List<ELEM>` est composée simplement de deux pointeurs, `head` et `tail`, qui pointent respectivement sur le début et la fin d'un liste simplement chainée de cellules (de type `Cell`, ou plus précisément `List<ELEM>::Cell`, car il s'agit d'une structure privée interne à `List<ELEM>`). Chaque cellule est elle-même composée d'une *valeur* de type `ELEM` (champ `val`) et d'un pointeur sur la cellule suivante (`next`). Le champ `next` de la dernière cellule de la liste est le pointeur nul.

La liste est dotée d'un constructeur par défaut (construisant la liste vide), d'un destructeur, et des opérations de copie. Elle dispose en outre des cinq fonctions-membres suivantes :

- `is_empty()` retourne un booléen indiquant si la liste est vide ;
- `append(ELEM)` ajoute un élément (et donc en particulier une cellule) à la fin de la liste ;
- `prepend(ELEM)` ajoute un élément (et donc en particulier une cellule) au début de la liste ;

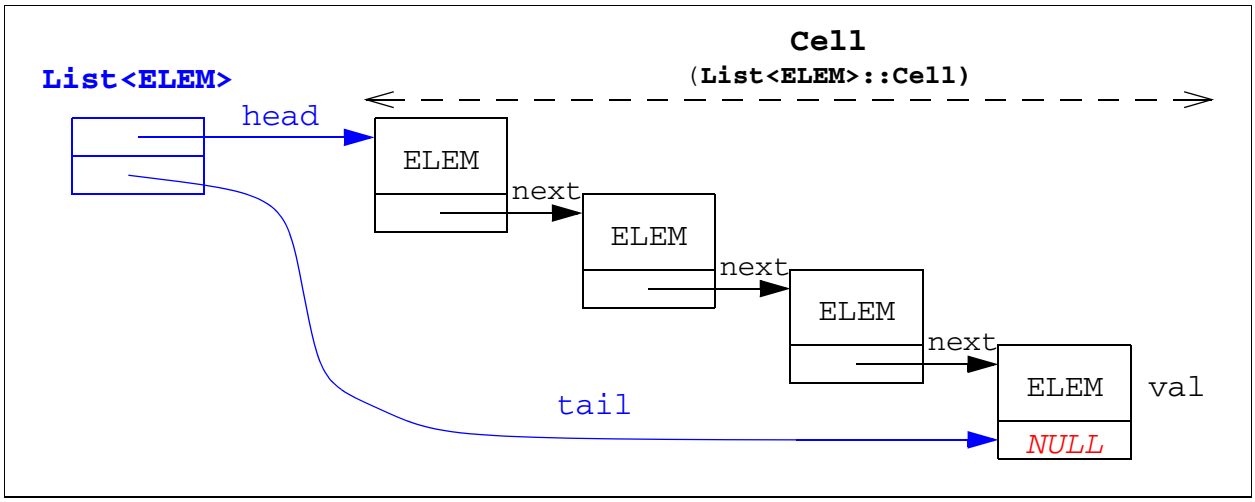

FIGURE 8.1 : Structure de `List<ELEM>`

- `insert(ELEM)` insère un élément devant le premier élément de la liste qui lui est supérieur ou égal (ou à la fin, s'il n'existe pas de tel élément) ; cette opération suppose que le type `ELEM` dispose des opérateurs de comparaison et en particulier d'`operator<` ; si on construit la liste uniquement à partir d'appels de `insert()`, la liste est donc toujours odronnée ;
- `get_first()` retourne la valeur du premier élément de la liste et *retire (et détruit) la cellule correspondante* ; `get_first()` lance l'exception `List<ELEM>::Empty` en cas de liste vide.

La classe dispose aussi d'une fonction amie d'affichage sur `ostream` (`operator<<`).

8.3 LES LISTES COMME IMPLÉMENTATION DES FILES ET DES PILES

Votre mission est d'utiliser cette classe `List<ELEM>` afin de réaliser de manière économique des collections bien connues, les piles et les files.

8.3.1 Les classes `Queue` et `Stack`

En prenant `List<ELEM>` comme classe de base, dérivez les deux classes génériques `Queue<ELEM>` et `Stack<ELEM>`. Vous utiliserez ici la dérivation publique, la seule vue en cours.

La classe `Queue<ELEM>` réalise la stratégie FIFO (*First In, First Out*) habituelle et doit disposer des fonctions membres suivantes :

- `put(ELEM)` qui ajoute un élément à la file ;
- `get()` qui retourne la valeur de l'élément le plus ancien dans la file, et en retire ce dernier.

La classe `Stack<ELEM>` réalise la stratégie LIFO (*Last In, First Out*) habituelle doit disposer des fonctions membres suivantes

- `push(ELEM)` qui ajoute un élément à la file ;
- `pop()` qui retourne la valeur de l'élément le plus récent dans la file, et en retire ce dernier.

Outre ces fonctions-membres les deux classes doivent être constructibles par défaut, disposer du prédicat `is_empty()` et être affichables sur les `ostream` par l'opérateur `<<`.

Vous devrez définir les fonctions spécifiques de ces classes le plus économiquement possible, en utilisant celles de `List<ELEM>` et en évitant toute duplication de code. En particulier, est-il nécessaire de définir pour ces deux nouvelles classes les opérations suivantes :

- le constructeur ?
- la fonction `is_empty()` ?
- les opérations de copie ?
- la fonction d'affichage (`operator<<`) ?

Bien entendu, votre programme de test devra vérifier ces points. En outre vous ne devez pas modifier le code de la classe `List<ELEM>` fournie ; vous installerez le code des nouvelles classes dans de nouveaux fichiers (`Queue.h` et `Stack.h`, il n'y a vraiment pas besoin de `.cpp` !).

8.3.2 Dérivation privée

Dans notre cas, l'utilisation de l'héritage public présente un défaut majeur sur le plan de l'architecture logicielle. Les instances de `Queue<ELEM>` et `Stack<ELEM>` étant substituables (partout) à celles de `List<ELEM>`, il est possible de leur appliquer directement n'importe quelle fonction-membre de cette dernière classe (`append()`, `prepend()`, `insert()`) ce qui a pour effet de briser la stratégie FIFO ou LIFO à laquelle elles sont censées obéir !

L'héritage privé permet de pallier cette inconvénient. Quand une classe dérive de manière privée d'une autre classe comme dans

```
class A
{
    // détails omis
};

class B : private A
{
    // détails omis
};
```

tous les membres publics et protégés[1] hérités de la classe de base (`A`) deviennent privés dans la classe dérivée (`B`). De plus, la conversion implicite d'héritage est alors invalide, sauf dans le corps des fonctions-membres et amies de la classe dérivée (un `B` n'est un `A` que dans le contexte d'un `B` !).

Modifiez la définition de vos classes `Queue<ELEM>` et `Stack<ELEM>` en utilisant l'héritage privé. Qu'en est-il alors de la question précédente sur la nécessité de redéfinir les fonctions suivantes :

- le constructeur ?
- la fonction `is_empty()` ?
- les opérations de copie ?
- la fonction d'affichage (`operator<<`) ?

Modifiez et testez vos classes en conséquence.

8.3.3 La classe `Priority_Queue`

Pour terminer, définissez une nouvelle classe générique, celle des files à priorité, `Priority_Queue<ELEM>`. Cette classe possède exactement la *même interface* que

1. Rappelons que les membres *privés* hérités de `A`, même s'ils sont présents dans la classe dérivée, n'y sont de toute façons pas accessibles. Il ne sont donc pas concernés par ce mécanisme.

`Queue<ELEM>`, et donc elle en dérivera *publiquement*. La différence est dans la sémantique de ses deux principales fonctions :

- `put(ELEM)` ajoute un élément dans la file, en faisant en sorte que la file à priorité reste ordonnée (dans l'ordre de l'opérateur < des `ELEM`, supposé exister) ;
- `get()` retourne la valeur du premier élément de la file à priorité et retire cet élément de la file.

N'avez-vous pas un problème en essayant de définir la fonction `put()` « à l'économie » ? L'héritage privé entre `List<ELEM>` et `Queue<ELEM>` ne pose-t-il pas problème ? Voyez vous une solution possible (autre que le rétablissement de l'héritage public à cet endroit) ?

Posez-vous également la même question sur les fonctions dont la redéfinition est nécessaire et modifiez et testez votre nouvelle classe en conséquence.

8.3.4 Typage dynamique

Dans la classe `List<ELEM>` fournie, aucune des fonctions-membres n'est déclarée virtuelle., à part le destructeur (tiens donc, et pourquoi lui ?). Quelles sont les fonctions pour lesquelles ce serait nécessaire (*indispensable*[1]) dans le cadre de l'architecture proposée ? Même question pour les fonctions spécifiques des classes `Queue`, `Stack` et `Priority_Queue` ? En particulier, réfléchissez au cas suivant :

```
Priority_Queue<int> prioq;
Queue<int> *pq = &prioq;
pq->put(3);
pq->put(2);
```

Quel doit être le résultat logique ? Codez en conséquence.

1. J'insiste : il ne s'agit pas ici de goûts ou de couleurs, ni de discipline ou d'habitudes de programmation. Il convient d'identifier les fonctions où l'oubli de la déclaration en tant que `virtual` mettrait en péril (logique ou mécanique) l'architecture de réutilisation de code mise en place.

TD 9
Fonctions virtuelles

OBJECTIF

Ces deux exercices illustrent la notion de *fonctions virtuelles* (aussi dénommées *méthodes*).

Le premier repose sur la classe `Expr`, introduite en cours, et qui permet de représenter des arbres d'expression arithmétique. Il montre comment l'approche objet bien conçue permet d'augmenter les fonctionnalités d'un programme sans modifier en aucune manière ce qui existe déjà (une propriété que l'on nomme souvent par *incrémentalité*). Il fournit aussi une occasion d'utiliser l'une des collections très utiles de la STL, la *map*.

Le second exercice est une incursion dans le monde de l'*héritage multiple*, mais dans un cas où ce type d'héritage ne pose vraiment aucun problème. Bien au contraire, il permet ici une grande élégance et une réelle économie de moyen.

9.1 EXTENSION DE LA CLASSE `Expr`

Vous trouverez dans le répertoire `Expr`, à l'endroit habituel, les fichiers source de la classe `Expr` et de ses dérivées. À quelques compléments évidents près, c'est le code vu en cours.

9.1.1 Deux extensions simples

Votre première mission est d'ajouter à la hiérarchie des nœuds d'expression deux nouveaux opérateurs :

1. l'opérateur (binaire) de modulo (opérateur `%` en C),
2. l'opérateur (ternaire) conditionnel (`?:` en C).

Attention ! Vos additions doivent se faire sans modifier en aucune manière les fichiers source contenant les classes fournies (`Expr.h` et `Exp-abstract.h`). Vous introduirez donc vos modifications dans deux nouveaux fichiers, disons `Modulo.h` et `Ternary.h`. En revanche vous aurez bien sûr à ajouter des tests pour les nouveaux opérateurs dans `main_Expr.cpp`.

9.1.2 Une extension un peu plus compliquée

Jusqu'à présent nous ne pouvons manipuler que des expressions dont les feuilles (nœuds terminaux) sont des constantes entières (`Constant`). Nous souhaiterions pouvoir manipuler aussi des *variables*.

Une variable a un nom (une simple `string`) et une valeur associée (un `int`). Pour traiter les variables, vous avez trois points à aborder :

- stocker les variables (c'est-à-dire représenter la correspondance nom-valeur ;
- définir un nouveau type de nœud d'expression 0-aire (feuille), la *référence à une variable* ;
- définir un nouveau type de nœud d'expression, l'*affectation*.

Stockage de la correspondance nom-valeur Notre premier problème est donc de ranger cette correspondance nom-valeur dans une table unique pour toutes les variables (une table globale ou un champ statique de classe). Pour cela, nous allons utiliser une collection de la STL, une *map*.

Une map est une collection générique dépendant de deux paramètres génériques :

```
map<K, T>
```

`K` est le type de la « clé » (dans notre cas `string`) ; `T` est le type de la valeur associée (dans notre cas `int`). Dans une map, il ne peut y avoir duplication de la même clé : pour une clé donnée, il y a donc au plus une valeur associée dans la map. Les maps se construisent évidemment par défaut (map vide). Enfin (et c'est vraiment la seule opération sur les maps dont vous avez besoin dans cet exercice), une map se manipule comme un tableau[1] dont le type de l'index est `K` et `T` le type des composantes. Cette indexation est bien entendu représentée par l'opérateur `[]`. Plus précisément soit `m` une `map<K, T>` et `k` une valeur de type `K` ; alors `m[k]` réalise l'algorithme suivant :

1. si il existe une entrée correspondant à la valeur de clé `k` dans la map (si elle existe, elle est unique par définition des maps), retourner une *référence* (`T&`) sur la valeur associée ;
2. sinon, allouer une nouvelle entrée correspondant à la valeur `k` de la clé, initialiser le `T` associé au *zéro* de son type et retourner une référence sur ce `T` ainsi initialisé.

On rappelle que le *zéro du type `T`* est ce que construit le constructeur par défaut (`T::T()`) de ce type. Dans le cas d'un `int`, c'est évidemment 0.

Les recherches et insertions dans une map sont garanties logarithmiques (réalisation sous forme d'un arbre binaire équilibré).

Pour accéder aux maps, vous devez inclure le fichier d'entête `<map>` et signifier que vous utilisez l'espace de nommage `std` (`using namespace std`).

Référence à une variable Pour utiliser les variables dans nos arbres d'expression nous avons besoin d'un nouveau type de nœud, la référence à une variable (`Variable_Ref`). Ces nœuds sont en fait des feuilles (tout comme `Constant`) ; ils contiennent juste le nom de la variable référencée et leur fonction `eval()` retourne simplement la valeur couramment associée à ce nom (dans la map).

Nous ferons l'**hypothèse simplificatrice** (et réaliste) suivante : si on évalue une référence à une variable qui n'a pas encore été définie, ce n'est pas une erreur : au contraire, cela a pour effet de définir la variable correspondante (c'est-à-dire de l'enregistrer dans la map) avec la valeur initiale 0.

Affectation Pour changer la valeur d'une variable, nous introduisons un nouveau type de nœud, l'affectation (`Assignement`). Bien qu'il ait deux opérandes, il ne s'agit pas d'une classe dérivée de `Binary_Expr` (pourquoi ?). Le premier opérande est (un pointeur sur) un noœud de référence à une variable (`Variable_Ref`). Le second est (un pointeur sur) une expression quelconque (`Expr`). La fonction `Assignment::eval()` calcule la valeur de cette expression et en fait la nouvelle valeur de la variable (*i.e.*, la range dans la map).

Attention ! Comme dans la question précédente, vous devez introduire cette extension sans modifier les fichiers source fournis. Vous introduirez donc un nouveau fichier `Variable.h`. Le fichier de test `main_Expr.cpp` contient (en commentaire) un exemple d'utilisation des variables (reprenant les notations indiquées ci-dessus).

1. On parle de tableau *associatif*.

9.2 MENUS EN CASCADE

On souhaite définir un ensemble de classes afin de réaliser une gestion simple (et même simpliste) de menu. Pour cela on se donne la spécification suivante de la classe `Menu` (fichier `Menu/Menu.h`, à l'endroit habituel) :

```
class Menu
{
private:
    string _title;    // titre du menu
    vector<Menu_Item *> _items;
                      // tableau de pointeurs sur les items

public:
    Menu(const string& t, int n, const vector<Menu_Item *>& its)
        : _title(t), _items(its)
    {}
    const string& title(void) const {return _title;}
    int nitems(void) const {return _items.size();}
    void activate(void) const;
};
```

Un menu est donc constitué d'un titre et d'un certain nombre d'*items*, instances de la classe `Menu_Item`. Un menu est initialisé à l'aide d'un tableau (`std::vector`, la classe vecteur de la STL) de pointeurs sur ses items (pourquoi des pointeurs ? voir plus loin). La fonction `activate()` active (!) le menu :

- elle affiche le titre du menu suivi de la liste des choix possibles, chaque choix étant identifié par un entier ;
- elle demande à l'utilisateur de faire sa sélection et lit la réponse (un entier) sur l'entrée standard ;
- elle exécute la sélection de l'utilisateur puis se termine ;
- si la sélection de l'utilisateur est incorrecte, elle émet un message d'erreur et demande à nouveau une sélection à l'utilisateur.

On voit donc qu'à une exécution de la fonction `activate()` correspond l'exécution d'*une seule* sélection.

La classe `Menu_Item` est *abstraite* ; en voici la spécification telle qu'elle apparait dans le fichier `Menu/Menu_Item.h` :

```
class Menu_Item
{
private:
    string _text;     // le message de selection

public:
    Menu_Item(const string& t) : _text(t) {}
    virtual ~Menu_Item() {}

    virtual string text(void) const {return _text;}
    virtual void execute(void) const = 0;
};
```

Un item est donc constitué d'un texte (qui est affiché pour identifier la sélection) et d'une action associée que l'on peut exécuter grâce à la fonction-membre `execute()`.

Il existe deux sortes d'items de menu :

- les *items simples* (classe `Simple_Menu_Item`) pour lesquels l'action est représentée par un simple pointeur sur une fonction (disons de type `void (*)()`),
- et les *menus déroulants* (classe `Walking_Menu`) pour lesquels l'action est d'activer un sous-menu (c'est-à-dire d'en exécuter la fonction `activate()`).

On voit donc qu'un menu déroulant est à la fois un menu et un item de menu. Son message de sélection (en tant qu'item) sera son titre (en tant que menu) que l'on fera suivre de la chaîne de caractères `" ->"`.

Le but de l'exercice est de spécifier et d'implémenter les classes `Simple_Menu_Item` et `Walking_Menu`, de terminer d'implémenter la classe `Menu` conformément aux spécifications données ici, et de fournir un programme de test de l'ensemble. Bien entendu, la solution doit être élégante, économique en code écrit comme en code déroulé, et permettre *un nombre quelconque de niveaux* de menus déroulants.

Voici un exemple d'exécution attendue (sous **zsh**) où l'on reconnaît aisément un menu principal avec un sous-menu. Les actions ont été simplifiées et se contentent d'afficher un message d'exécution. Le programme principal exécuté ici est une boucle infinie d'appel de la fonction `activate` du menu. On en sort seulement sur fin de fichier ou par sélection de *quitter*. Dans cet exemple, les caractères entrés par l'utilisateur sont en ***gras italique***, ceux affichés par la machine en `police normale`.

```
2-borobudur% test_Menu

        LE MENU
0- emacs
1- xload
2- COMMUNICATIONS ->
3- quitter
Votre choix? 0
************** Execution de EMACS

        LE MENU
0- emacs
1- xload
2- COMMUNICATIONS ->
3- quitter
Votre choix? 2

        COMMUNICATIONS
0- news
1- xwais
2- xftp
3- xarchie
Votre choix? 2
************** Execution de XFTP

        LE MENU
0- emacs
1- xload
2- COMMUNICATIONS ->
3- quitter
Votre choix? 3
************** Execution de QUITTER
2-borobudur%
```

TD 10
Algorithmes de la STL

OBJECTIF

L'exercices suivant a pour but de continuer l'initiation à l'utilisation (débutée avec le TD 3) de la bibliothèque STL de C++. Il donne l'occasion d'utiliser des objets-fonctions dans les algorithmes de la STL ainsi que des fonctionnelles associées. Il montre également comment on peut étendre la STL avec ses propres algorithmes. L'exercice explore en outre certains des problèmes posés par les collections de pointeurs, indispensables avec des types polymorphes.

Quelques fichiers sont fournis dans le répertoire `Algorithms`, situé à l'endroit habituel. Il sera sans doute utile de consulter la documentation de la STL, par exemple sur `http://www.sgi.com` et aussi de se reporter au TD 3.

La « règle du jeu » ici est de ne pas utiliser de boucles d'itération explicites et donc de les remplacer par l'utilisation des algorithmes de la STL. Pour cela vous aurez sûrement besoin d'introduire de (courtes) fonctions annexes, voire des objets-fonctions. Cela vous introduira au style très particulier mais, à mon avis, très plaisant de la programmation avec la STL.

10.1 DÉFINITION DE COLLECTIONS POLYMORPHES

Dans le répertoire `Algorithms` mentionné précédemment, on trouvera le fichier `Figure.h` contenant la définition d'une hiérarchie de classes représentant des figures géométriques élémentaires (rectangle, carré, ellipse et cercle) avec quelques fonctions associées. On trouvera également le fichier `stl_print.h` déjà utilisé dans le TD 3.

Écrivez un programme principal qui utilise cette hiérarchie en créant une liste de pointeurs (pourquoi des pointeurs ?) sur quelques[1] figures de différents sous-types (ces figures seront créées par `new`).

Imprimez les figures de cette liste (noter que la classe `Figure` et ses dérivées disposent d'un opérateur `<<` pour les `ostream`). Pour cela vous pouvez utiliser l'algorithme `print` défini dans le fichier `stl_print.h` (déjà utilisé lors du TD 3).

10.2 MANIPULATION DE COLLECTIONS POLYMORPHES

10.2.1 Utilisation des fonctions-membres de `Figure`

Appliquez une rotation de $\pi/2$ à toutes les figures de la liste et imprimez la liste en résultant.

Faites de même en appliquant une homothétie (méthode `scale()`) d'un facteur 2.

10.2.2 Tri d'une collection

Modifier la classe `Figure` et ses dérivées (vous avez donc ici le droit d'éditer le fichier `Figure.h`) afin que les figures soient *comparables*, c'est-à-dire dotée d'un opérateur `<`. Pour simplifier, nous définirons la comparaison de la façon suivante : une figure `f1` est plus petite que `f2` (`f1 < f2`) si le rectangle englobant `f1` tient strictement à l'intérieur de celui englobant `f2`.

1. N'en exagérez pas le nombre !

Triez votre liste par ordre de taille croissante d'abord, puis décroissante. Imprimez chaque fois la collection obtenue. Pour l'ordre décroissant, avez-vous besoin de définir un nouvel opérateur de comparaison ou pouvez-vous vous en passer ?

10.3 IDENTIFICATION DU TYPE DES FIGURES D'UNE COLLECTION

Définissez une fonction (ou un objet-fonction) *template*, disons `is_kind_of`: si `p` est un pointeur sur `Figure`, `is_kind_of<T>(p)` retourne vrai si le type dynamique de l'objet pointé par `p` est en fait de type `T` ou dérivé de `T`. Pour cela vous aurez certainement besoin de l'opérateur `dynamic_cast`.

Tester votre fonction `is_kind_of<T>` en copiant toutes les ellipses (et donc aussi les cercles) de votre liste dans une nouvelle collection. Imprimez cette collection. Comme l'algorithme `copy_if` serait bien utile et qu'il n'existe pas (encore) dans la STL, définissez le donc.

10.4 DESTRUCTION DES FIGURES D'UNE COLLECTION

Ensuite on cherche non seulement à retirer de la liste toutes les figures qui sont des ellipses ou dérivées de `Ellipse`, mais aussi à les détruire définitivement.

À titre de première tentative, utiliserez votre fonction `is_kind_of<T>` dans le cadre d'un simple `remove_if` et imprimez la totalité (de `begin()` à `end()`) de la liste résultante. Que constatez-vous ? Avec un petit effort de mémoire, ceci devrait vous rappeler la fin de l'exercice 3.4.

Pour faire correctement cette dernière question, il faut tout d'abord détruire les objets pointés par les éléments de la liste. Pour cela vous définirez une fonction ou un objet-fonction `delete_Figure` qui, recevant un pointeur sur `Figure` en paramètre, lui applique l'opérateur `delete` et met le pointeur à 0 (***attention à cette dernière spécification !***). Ensuite vous définirez un algorithme qui n'existe pas dans la STL, bien qu'il soit souvent pratique, `for_each_if`. L'appel

```
f = for_each_if(first, last, uf, pred);
```

applique la fonction `uf()` à tous les éléments de l'intervalle d'itération `[first, last[` qui satisfont le prédicat `pred()`; comme pour `for_each`, la valeur retournée est une copie de `uf`, après traversée de toute la collection.

Tout ceci devrait vous permettre de détruire tous les éléments de la liste qui sont des sortes d'ellipses et de mettre à zéro les pointeurs correspondants ; pour achever la question, il suffira de détruire effectivement (méthode `erase()`) les éléments ainsi annulés.

Note importante Dans tout cette exercice, ***vous n'avez droit qu'à deux boucles d'itération explicites,*** celles qui permettent d'implémenter les algorithmes `copy_if` et `for_each_if`. Tout le reste doit être réalisé à l'aide d'algorithmes de la STL. Vous devez également utiliser, chaque fois que faire se peut, les objets-fonctions et les fonctionnelles prédéfinis de la STL.