C++ / TP

Polytech Paris-Saclay - Et4 info
Marc Fonvieille et Joël Gay

Première partie

Règles et prise en main

0 Bonnes pratiques

La programmation devient rapidement une affaire de communication avant tout. La part du temps passé à lire et à comprendre le code d'un autre ou son propre code peut devenir prédominante. Il est donc plus important d'écrire un code compréhensible qu'un code juste : un code faux mais clair est vite corrigé, un code juste mais incompréhensible provoque des erreurs. Voici un guide de bonnes pratiques que nous vous conseillons de suivre scrupuleusement. Tout écart de votre part sera à vos risques et périls.

0.1 Organisation des projets

- Chaque exercice doit être effectué au sein d'un nouveau projet ou dossier.
- La réponse à chaque question doit être conservée, qu'elle soit bonne ou fausse. Utilisez les commentaires et le renommage pour mettre de côté une mauvaise réponse corrigée.
- Si le nombre de fichiers source d'un projet devient important, organisez votre projet.
- La compilation complète d'un projet doit être possible en une seule commande simple : utilisez un Makefile.

0.2 Conventions de codage

Ne pas prendre exemple sur la STL qui est un cas particulier et possède une histoire contraignante, contrairement à vos projets ;).

- Seul le nom d'une classe commence par une majuscule.
- Les noms utilisés dans le préprocesseur sont en majuscules.
- Les macros à la C doivent être évitées en C++.
- Les noms doivent être aussi explicites que nécessaire mais pas plus. Prendre en compte la portée d'un nom : un nom utilisé dans tout le code doit être très explicite mais une variable locale peut être nommée plus succinctement à l'aide d'abréviations par exemple.
- Préfixer les variables membres d'un `m_` ou utiliser le signe distinctif de votre choix.

- L'usage de _ est rarement en faveur de la lisibilité. `int nbVoitures` ou `int nb_voitures` ? Prenez un parti et gardez-le.
- Jamais de nombre magique, c'est-à-dire de valeur littérale tombée du ciel dans le code. Utiliser des constantes à la place.

0.3 Bonnes pratiques diverses

- `cppreference.com` et `stackoverflow.com` sont indispensables.
- La meilleure documentation est le code lui-même mais elle ne suffit jamais.
- Un commentaire ne doit jamais dire ce que le code dit explicitement. Un commentaire donne des informations de haut niveau, pas une explication du langage. Par exemple, ne dites pas qu'une fonction virtuelle est virtuelle dans un commentaire. Cela n'apporte rien de plus que le mot clef `virtual`. Ceci n'est en aucun cas une dispense d'écrire des commentaires, au contraire. Une bonne pratique est d'utiliser la convention du générateur de documentation **Doxygen** (Cf. `https://en.wikipedia.org/wiki/Doxygen`). Vous devriez donc utiliser les mots clés `@param` et `@return` dans vos commentaires de fonctions ou méthodes ainsi qu'une brève phrase de description de ce qu'elles font.
- Une seule classe par header.
- Tout header doit être protégé contre l'inclusion multiple.
- Une seule action par ligne.
- Indentation cohérente : toujours le même pas et placement cohérent des accolades.
- Sauter des lignes à bon escient : un code trop compact est indigeste mais un code trop aéré dilue l'information.
- Garder les fonctions courtes mais ne pas les surdécomposer non plus. Une fonction ne sert pas à raccourcir les autres, elle sert à réutiliser du code.
- Tester son code au moins succinctement.
- Toujours déclarer `const` ce qui doit l'être si sa modification peut causer des effets de bord seulement. Par exemple, il est inutile de déclarer `const` un argument numérique dans une fonction car cela alourdit la signature de la fonction.
- Éviter l'utilisation de `using namespace std;` car l'on risque, par inadvertance, d'avoir des collisions entre les noms que l'on définit et ceux présents dans la bibliothèque standard. Au pire, on peut utiliser, par exemple, `using std::cin;`, si l'on veut éviter d'avoir à taper `std::` devant `cin`.

1 Hello world !

1.1 Compilation manuelle, le B-A-BA

Il est essentiel de savoir utiliser un compilateur C++ tel que `g++` (compilateur C++ du projet GNU). De très nombreux projets industriels d'envergure sont encore à ce jour compilés à l'aide de fichiers Makefile et donc à l'aide de commandes de compilation rédigées manuellement. C'est

cette méthode qui offre la plus grande souplesse d'utilisation même si elle peut devenir complexe à utiliser.

1. Créer le fichier `hello_world.cpp` et y copier le code donné dans Source 1 (`hello_world.cpp`).
2. Compiler ce code à l'aide de la commande
   `$ g++ hello_world.cpp`
3. Exécuter le fichier `a.out` qui a été généré. Constater qu'il affiche le message attendu.

   La commande `g++` n'est jamais utilisée ainsi en pratique. Voici ses options principales :

   - `-o output_name` permet de spécifier le nom du fichier (exécutable) créé par la commande.
   - `-c` permet de créer un fichier objet. Il est préférable de compiler séparément chaque fichier source (`*.cpp`) en fichier objet (`*.o`) dans un premier temps et de les linker ensuite.
   - `-g` permet d'ajouter des informations de débogage exploitables par un débogueur comme `gdb`.
   - `-Wall -Wextra` indiquent à `g++` de donner davantage de messages d'avertissement. Il est recommandé de toujours utiliser ces options et de ne les désactiver qu'en cas de nécessité.
   - `-Werror` transforme les `Warnings` en `Errors` qui empêchent la compilation. Il est également recommandé d'activer cette option afin d'apprendre les bonnes pratiques et de détecter certaines étourderies qui arrivent même aux meilleurs. À ne désactiver qu'en cas de nécessité.
   - `-std=c++11` indique la version du langage qui est utilisée dans le code à compiler.

4. Recompiler le fichier `hello_world.cpp` à l'aide des commandes suivantes :
   ```
   $ g++ -Wall -Wextra -Werror -std=c++11 hello_world.cpp -c -o hello_world.o
   $ g++ hello_world.o -o hello_world
   ```
5. Télécharger le `Makefile` donné et l'examiner à l'aide du tutoriel sur Makefile. Recompiler le programme `hello_world.cpp` à l'aide de la commande suivante :
   `$ make hello_world`

1.2 Les IDE

Un IDE est un Environnement de Conception Intégré. C'est un logiciel qui rassemble de nombreux outils nécessaires au développement logiciel. Entre autres, on retrouve un éditeur de texte, un compilateur et un débogueur. **On découragera dans ce module l'utilisation d'un IDE.**

⇒ il est fortement conseillé d'utiliser un simple éditeur de texte, vous obligeant ainsi à vous concentrer sur le code et sa syntaxe.

2 Entrées et sorties

Il est toujours utile de pouvoir échanger des informations entre le programme et l'extérieur. Cet exercice utilise les flux standards d'entrée et de sortie ainsi que les paramètres passés par

la ligne de commande. La source 2 est un exemple minimal d'utilisation de ces techniques.
`Source 2 - inout.cpp`
À l'aide de la documentation pour la fonction `rand()` (`http://en.cppreference.com/w/cpp/numeric/random/rand`), écrivez un jeu plus ou moins obéissant aux règles suivantes :

- La taille maximale du nombre à deviner est passée par la ligne de commande.
- Le programme choisit un nombre au hasard entre 0 et la valeur passée par la ligne de commande.
- L'utilisateur tente de le deviner et le programme indique si le nombre mystère est supérieur ou inférieur au dernier nombre entré par l'utilisateur.

Deuxième partie
Exemples de classes simples

3 Nombres complexes

3.1 De la structure...

1. Créer une structure `cpx` qui stocke un nombre complexe sous la forme de deux réels (`double`).
2. Implémenter une fonction `add` qui reçoit deux nombres complexes, fait une addition et retourne le résultat.
3. Modifier la fonction en ajoutant un paramètre `bool printRes` pour choisir d'afficher ou de ne pas afficher le résultat. Imposer `false` comme valeur par défaut pour ce paramètre.
4. Implémenter une (autre) fonction `add` qui reçoit trois nombres complexes, additionne les deux premiers et stocke le résultat dans le troisième.
5. Implémenter une fonction `main` pour tester les fonctions précédentes.

3.2 ...à la classe

A partir du travail précédent, créer une classe `Complexe` qui possède :

- deux attributs membres (privés) ;
- un constructeur `Complexe(double, double)` (utiliser une liste d'initialisation) ;
- une méthode membre d'affichage ;
- une méthode `add(Complexe)` qui ajoute un autre complexe passé en argument à l'objet courant.
- Est-ce que la syntaxe `z1.add(z2).add(z3)` est valide et a l'effet attendu ? Si ce n'est pas le cas le rendre possible.

Ne pas oublier de rajouter le mot-clé `const` à chaque fois que c'est pertinent. Tester. **On n'a pas besoin d'utiliser le mot clé `new` !**

A retenir : type `bool`*, surcharge de fonction, paramètre par défaut, classe, constructeur et liste d'initialisation, manipulation d'objet par référence (paramètre et retour),* `const`*.*

4 File d'attente

Cet exercice permet de manipuler les méthodes et attributs de classe statiques. Nous cherchons à implémenter un système de file d'attente pour un passage au guichet. Le passage est effectué suivant le numéro de ticket affiché au guichet.

1. Créer une classe `Ticket` avec les propriétés suivantes :

   - Chaque `Ticket` instancié possède un numéro unique.
   - La classe a en commun entre toutes ses instances deux compteurs : un qui donne le prochain ticket créé (`nextCreatedTicket`), et l'autre contenant le prochain ticket appelé (`nextCalledTicket`).
   - À chaque instanciation d'un `Ticket`, le compteur `nextCreatedTicket` est automatiquement incrémenté.
   - Après appel du `Ticket` au guichet, ce dernier doit être détruit. Autrement dit chaque destruction d'un ticket incrémente `nextCalledTicket` indiquant le `Ticket` suivant appeler au guichet.
   - Chacun des deux compteurs possède un accesseur de lecture. Il doit être utilisable sans disposer d'une instance de la classe `Ticket`.
   - Une méthode retournant un booléen indique si le tour d'un `Ticket` est venu.

   Implanter cette classe dans un fichier `.hpp` et les méthodes dans un fichier séparé.

2. Tester la classe `Ticket` et chacune de ses méthodes. Afin de causer la destruction d'un objet, vous pouvez l'allouer dynamiquement, mais on utilisera plutôt ici un bloc entre accolades. Les variables déclarées dans ce bloc sont détruites à sa sortie. Un tel bloc peut exister sans suivre un if ou toute autre structure de contrôle.

A retenir : attributs et méthodes statiques, allocation dynamique avec `new` *et* `delete`*.*

5 Vecteurs et matrices

5.1 Classe Vector

Le but est d'implémenter une classe `Vector` qui permet de manipuler un tableau de type `int` (créer, détruire, accéder aux éléments par l'indice et les modifier). `Vector` aura pour attributs membres privés :

Nom	Type
`m_size`	`int`
`m_v`	`int *`

1. (a) Déclarer la classe avec ses membres et ses méthodes (accès aux données membres, accès et modification des éléments de `v` par l'indice) dans un fichier *header* (**.h** ou **.hpp** usuellement). Faire en sorte qu'il soit impossible de modifier les éléments de `v` lorsqu'on utilise l'accesseur `getV()`.

   (b) Implémenter les méthodes de la classe.

   (c) Implémenter le constructeur `Vector(int size)` en allouant la mémoire avec l'opérateur `new[]`.

   (d) Implémenter le destructeur.

2. Implémenter une fonction `main()` pour tester la classe. Tentez de modifier les éléments de `v`.

3. Implémenter `inline` la fonction `isInLimits(int idx, int lim1, int lim2)` qui vérifie que `idx` est bien compris entre les limites `lim1` et `lim2`. Utiliser cette fonction pour implémenter un contrôle de dépassement d'indice au sein des fonctions d'accès et de modification des éléments par l'indice. Si l'indice n'est pas valide, le programme s'arrêtera et affichera un message d'erreur (fonction `std::exit` de `<cstdlib>`)

4. Grâce à une **variable statique**, garder trace du nombre total d'éléments manipulés par les différentes instances de `Vector` (la somme de toutes les tailles d'instances de `Vector` existantes).

5.2 Classe Matrix

1. Implémenter une classe `Matrix` qui représente une matrice $n \times m$.

Nom	Type
`m_n`	`int`
`m_m`	`int`
`m_M`	`int*`

2. Implémenter les méthodes `getElement` et `setElement`.

3. Écrire une fonction **amie** `mulMatVec` qui multiplie une matrice par un vecteur de type `Vector` et stocke le résultat dans le dernier paramètre :

```
void mulMatVec(const Matrix &M, const Vector &v, Vector &res)
```

A retenir : allocation dynamique, tableaux, fonction inline, fonction amie (il existe aussi des classes amies), passage de paramètres par référence.

6 Dates

1. Créer la classe **Date** qui aura pour attributs membres :

Nom	Type
`m_year`	`int`
`m_month`	`int`
`m_day`	`int`
`m_today`	`static Date`

2. Implémenter un constructeur, un destructeur, une méthode pour afficher la date. Ne pas oublier d'initialiser `m_today` à une valeur par défaut (en tête du fichier .cpp).

3. Implémenter deux méthodes **static** pour lire et écrire la date actuelle : *

```
static void setToday();
static const Date& getToday();
```

4. Implémenter une méthode `getAge(const Date& d)` pour calculer le nombre d'années entre la date contenue dans l'objet et la date d.
(*i.e.* `Date(18,12,1994).getAge(Date::getToday())` doit retourner l'âge de quelqu'un né le 18/12/1994)

* Pour obtenir la date actuelle on utilisera la bibliothèque `<ctime>` :

```
#include <ctime>
/* ... */
time_t now = time(0);
tm* dt = localtime(&now);
// dt->tm_mday renvoie le numero du jour
// dt->tm_mon+1 renvoie le numero du mois
// dt->tm_year+1900 renvoie l'annee
```

7 Surcharge d'opérateurs avec Vector

Reprendre la classe `Vector` que vous avez implémentée précédemment.

1. Implémenter : **copy constructor, copy assignment, move constructor, move assignment** (vérifier que le compilateur utilise le C++11 pour ces deux derniers [1]).

```
Vector (const Vector& x);           // copy constructor
Vector& operator= (const Vector& x); // copy assignment
Vector (Vector&& x);                // move constructor
Vector& operator= (Vector&& x);     // move assignment
```

2. Tester ces membres. On utilisera `std::move` de la bibliothèque `<utility>` pour tester le mouvement.

3. Surcharger et tester les opérateurs suivants : *(fonctions membres de* `Vector`*)*

```
// acces a un element de Vector
int& operator [] (int n) const;
// multiplication par -1
Vector operator - () const;
// addition
Vector operator + (const Vector& x) const;
// soustraction
Vector operator - (const Vector& x) const;
// produit scalaire
double operator * (const Vector& x) const;
// comparaison (egalite)
bool   operator == (const Vector& x) const;
// comparaison (inegalite)
bool   operator != (const Vector& x) const;
// ajouter une valeur a Vector
Vector operator + (int value) const;
// soustraire une valeur a Vector
Vector operator - (int value) const;
```

Penser à réutiliser (si possible) les opérateurs que vous avez déjà surchargés.

4. Surcharger et tester l'opérateur d'insertion : *(fonction amie de* `Vector`*)*

```
friend std::ostream& operator<<(std::ostream& s, const Vector& v);
```

A retenir : règle de 3 (établir les invariants, conserver les invariants, terminer correctement les ressources), constructeur/affectation de copie, surcharge d'opérateurs.

1. option `-std=c++11` de g++

Troisième partie

Héritage

8 Héritage simple : classes Personne, Employe, Etudiant

1. Créer la classe **Personne** avec les membres suivants :

Nom	Type
`m_nom`	`string`
`m_prenom`	`string`
`m_birthDate`	`Date`

   (`Date` est la classe créée en Section 6.)

2. Implémenter un constructeur, un destructeur, les accesseurs et mutateurs.

3. Implémenter une méthode `getAge()` pour calculer l'âge de la personne.

4. Implémenter une méthode `print()` pour afficher tous les attributs.

5. Créer la classe **Etudiant** qui hérite de la classe **Personne** et qui contient les membres supplémentaires :

Nom	**Type**
`m_numCarteEtud`	`string`
`m_nomUniversite`	`string`

6. Implémenter un constructeur, un destructeur, les accesseurs et mutateurs.

7. Créer la classe **Employe** qui hérite de la classe **Personne** et qui contient les membres supplémentaires :

Nom	**Type**
`m_salaire`	`float`
`m_nomEntreprise`	`string`

8. Implémenter un constructeur, un destructeur, les accesseurs et mutateurs.

9. Pour chaque classe, implémenter une méthode `print` pour afficher **tous** ses attributs (y compris ceux de sa classe mère).

10. Tester les classes et les méthodes dans `main`. Quel serait le danger si le destructeur de la classe mère n'était pas appelé lors de la destruction d'une instance de la classe fille ?

11. Implémenter une fonction `affiche_personne(const Personne* p)` qui va appeler la méthode `print` de l'objet. Tester cette fonction dans `main` avec les objets `Personne`, `Employe`, `Etudiant`. Quelle méthode `print` est appelée ?

A retenir : héritage, constructeurs/destructeurs dans une relation d'héritage, redéfinition de fonction, résolution dynamique de liens

9 Héritage multiniveaux : classe Manager et polymorphisme

La classe `std::vector` de la STL est déclarée dans l'en-tête `<vector>`.

1. Créer la classe `Manager` qui hérite de la classe `Employe` et qui contient les membres supplémentaires :

Nom	Type
`m_equipe`	`vector<Employe>`
`m_primeAnnuelle`	`float`

2. Implémenter un constructeur, un destructeur, les accesseurs, mutateurs, une méthode `print()`.
3. Implémenter une méthode pour ajouter des employés dans le groupe du manager.
4. Créer dans le `main` des instances des classes `Employe, Etudiant, Manager`. Créer ensuite des pointeurs de type `Personne*` et leur assigner les références des objets créés (ce qui est possible puisque ces classes héritent toutes de la classe `Personne`). Par exemple :

```
Employe e1(...);
Personne *p1 = &e1;
```

5. Que va afficher `p1->print()` dans chacun des cas ?
6. Rendre maintenant la méthode `print()` de la classe `Personne` virtuelle (mais pas virtuelle pure !). Que va maintenant afficher `p1->print()` ?
7. Que se passe-t-il lors de la destruction d'une instance d'`Employe` si on la crée de la façon suivante :

```
Personne *p1 = new Employe(/*...*/);
```

8. Pour éviter cela, rendre le destructeur de la classe `Personne` virtuel. De cette manière, le destructeur à utiliser sera choisi à l'exécution et non à la compilation.

A retenir : polymorphisme, méthode virtuelle

10 Les classes abstraites

Les classes que vous créez dans cet exercice doivent fonctionner avec la fonction `main` donnée dans `Ex10.cpp`. *(Sortie attendue indiquée en commentaire dans le fichier)*

1. Créer la classe abstraite **Shape** avec les membres suivants :

Nom	Type
`m_width`	`double`
`m_height`	`double`

2. Créer les méthodes virtuelles pures `area()`, `perimeter()` pour calculer la surface et le périmètre.
3. Créer la classe **TriangleRectangle** qui hérite de la classe **Shape** et implémenter les méthodes `area()` et `perimeter()` (utiliser `std::pow` de la bibliothèque `<cmath>`).
4. Créer la classe **Rectangle** qui hérite de la classe **Shape** et implémenter les méthodes `area()` et `perimeter()`.
5. Surcharger l'opérateur d'insertion `<<` pour la classe `Shape`, de sorte que l'expression `std::cout << shp`, où `shp` est un objet de type **TriangleRectangle** ou **Rectangle**, ait le résultat attendu (afficher une description textuelle de `shp`).

   Attention : Typiquement on surcharge l'opérateur `<<` en déclarant une fonction amie :

```
friend std::ostream& operator<<(std::ostream& s, const Shape& shp);
```

Cette fonction n'étant pas une fonction membre, elle ne profite pas directement du polymorphisme.

A retenir : classe abstraite

11 Héritage multiple

1. Créer la classe **CarteMere**

Nom	Type
m_id	int
m_typeProcesseur	string

2. Implémenter un constructeur et la méthode `getId()` ;

3. Créer la classe **Moniteur**

Nom	Type
m_id	int
m_resolutionX	int
m_resolutionY	int

4. Implémenter un constructeur et la méthode `getId()` ;

5. Créer la classe **Ordinateur** qui hérite des classes **CarteMere** et **Moniteur**

Nom	Type
m_nbdisqueDur	int
m_graveurCd	bool

Implémenter un constructeur.

6. Dans le `main` créer un objet de la classe **Ordinateur** et appeler la méthode `getId()`. Que se passe-t-il ?

Pour explicitement appeler la méthode `getId()` d'une classe précise, on peut utiliser l'opérateur de portée :

```
Ordinateur ord1(...);
int id1 = ord1.CarteMere::getId();
int id2 = ord1.Moniteur::getId();
```

7. Créer la classe **Materiel**

Nom	Type
m_id	int

8. Implémenter la méthode `getId()` pour lire `id`. Modifier les classes **CarteMere** et **Moniteur** pour les faire hériter de la classe **Materiel** et utiliser l'`id` de cette classe au lieu d'un attribut propre (elles n'ont donc plus de méthode `getId()`).

9. Dans le `main`, appeler la méthode `getId()` à partir d'un objet de la classe **Ordinateur**. Quel est le problème ?

A retenir : héritage multiple, opérateur de portée

12 Diamond problem

Pour résoudre le problème d'accès aux champs de la classe de base (Diamond problem), on peut utiliser l'*héritage virtuel*. Dans la partie précédente, les instances de **CarteMere** et **Moniteur** créées par lors de l'appel au constructeur d'**Ordinateur** héritent de deux instances de **Materiel** différentes. L'héritage virtuel leur permet de partager une seule et même instance mère (donc un seul `id` et une seule méthode `getId()`).

1. Modifier le type d'héritage à `public virtual` pour les classes **CarteMere** et **Moniteur**.
2. Modifier le constructeur de la classe **Ordinateur** pour appeler directement le constructeur de la classe de base.
3. Dans le main, appeler la méthode `getId()` à partir d'un objet de la classe **Ordinateur**. Qu'est-ce qui s'affiche ?
4. Quel est l'ordre d'appel des constructeurs (destructeurs) ?

A retenir : diamond problem, héritage virtuel

Quatrième partie

Divers

13 Templates

Reprendre la classe `Vector` que vous avez implémentée précédemment.

1. Modifier la classe `Vector` pour obtenir une version générique de type `template` qui peut travailler avec différents types de base `T` (a priori des types numériques comme `int, short, double, etc.`). Changer l'implémentation de toutes les méthodes. Tester la classe dans le `main`.

   Note : l'implémentation des fonctions « templatées » se fait obligatoirement dans un fichier header (p. ex. `cVector_templ.hpp`). Inclure ce fichier après la déclaration de la classe.

   Conseil :

   - Version « templatée » de l'opérateur d'insertion `<<` : Le déclarer comme fonction `friend` opérant sur un type de base `U` :

     ```
     template<class U> friend std::ostream& operator<<(std::ostream& s, const
         Vector<U>& x);
     ```

2. Changer les méthodes d'addition, de soustraction, de multiplication et de comparaison, pour travailler avec deux vecteurs de types différents (`T` et `U` respectivement).

   Conseils :

   - Dans la classe `Vector<T>`, déclarer la deuxième classe `Vector<U>` comme une classe `friend` :

     ```
     template<class U> friend class Vector;
     ```

   - Ensuite, déclarer les opérateurs comme dans l'exemple ci-dessous :

     ```
     template<class U> Vector operator+(const Vector<U>& x) const;
     ```

3. Spécialiser les méthodes de comparaison `operator==` et `operator!=` de la classe `Vector` pour travailler avec des données de type `std::string`. La comparaison entre deux `std::string` peut être faite grâce à la fonction membre `std::string::compare` (cf. documentation). Vérifier si d'autres fonctions ont besoin d'une spécialisation pour le type `std::string` (par exemple on peut traiter l'addition comme concatenation ...)

   Conseil :

   - Vous mettrez les déclarations des fonctions spécialisées juste après la déclaration de la classe `Vector`. Vous mettrez les définitions de ces fonctions dans un fichier `.cpp` (on n'a pas eu besoin d'un fichier `.cpp` jusqu'ici dans cet exercice).

14 Exceptions

1. Dans la classe `Vector`, modifier les méthodes `operator[]`, `getElement`, `setElement` pour générer une exception générique de type `std::exception` lorsqu'on tente d'accéder à un élément hors des limites du vecteur. Traiter l'erreur dans le `main` grâce au bloc :

   ```
   try {/*...*/} catch (std::exception&) {/* ... */}
   ```

2. Dans la classe `Vector`, créer une nouvelle classe d'exceptions `OutOfRangeException` qui hérite de la classe de base des exceptions `std::exception`. Redéfinir la méthode virtuelle `what()` qui retourne le texte de l'exception. Faire en sorte qu'une `OutOfRangeException` soit lancée lorsque c'est approprié. Tester dans le `main`.

15 Itérateurs et manipulation de fichiers de texte

Le fichier `info.txt` contient des données confidentielles concernant 200 employés de 10 géants de l'informatique (prénom, nom, salaire, employeur).

1. On utilisera dans cet exercice le `vector` de la STL. Lire le fichier `info.txt` et stocker toutes les informations dans un

   ```
   std::vector< std::vector<std::string> >
   ```

   Notez les espaces entre les chevrons fermants qui peuvent être confondus avec l'opérateur de flux par certains compilateurs.

   Conseil :

   - Pour la lecture, utilisez une boucle comme la suivante :

     ```
     string w[4];
     while (ifs >> w[0] >> w[1] >> w[2] >> w[3]) {
         ...
     }
     ```

     où `ifs` est un flux de type `std::ifstream` que vous devez initialiser correctement. (cf. documentation de `std::ifstream` sur `cppreference.com`)

2. Parcourir les informations lues **avec uniquement des itérateurs** afin d'identifier qui travaille à Microsoft ou à Apple, en observant que l'employeur est toujours la dernière information qu'on a sur la personne. Supprimer ces employés du vecteur.

   Fonctions utiles : `std::vector::cbegin`, `std::vector::cend`, `std::vector::erase` (cf. documentation).

   Note : si on doit travailler avec des objets constants il faut utiliser un `vector<string>::const_iterator` par exemple.

3. Écrire le vecteur dans un autre fichier texte, en gardant le même format (prénom, nom, salaire, employeur). Pour l'écriture vous utiliserez toujours des itérateurs sur le vecteur, ainsi qu'un flux de type `std::ofstream`.

4. (facultatif) Créer une classe d'itérateurs **Iter** pour parcourir les éléments de vos `Vector` (avec *a minima* les opérateurs `operator++()` (préfixe), `operator++(int)` (postfixe), `operator*()` (déréférencement)). Ajouter également les fonctions

   ```
   Iter Vector::begin()
   Iter Vector::end()
   ```

   qui renvoient des itérateurs vers le premier et le dernier élément du `Vector`.