# TD 1 : Rappels de programmation C et orientée objet

**Objectifs pédagogiques :**

- **mémoire, pointeur, classe**
- **syntaxe générale du c++**
- **constructeur, destructeur**
- **opérateurs**

## Introduction

Dans ce premier TD, nous allons réaliser en C++ une classe `String` appuyée par une représentation interne à l'aide d'une chaîne de caractères du C : un pointeur `char *` vers un espace mémoire zéro terminé. Cet exercice est à vocation pédagogique, on préférera en pratique utiliser la `std::string` standard du header `<string>` et les fonctions de manipulation de chaînes du C rangées dans `<cstring>`.

Pour être sûr de ne pas entrer en conflit avec d'autres applications, nous utiliserons le namespace `pr` pour notre implémentation.

## 1.1 Rappels chaîne du C, const.

Les chaînes de caractères du C, sont représentées par un pointeur de caractère qui cible une zone mémoire contenant la chaîne, et se terminant par un `'\0'`.

On souhaite implanter nos propres petites fonctions utilitaires travaillant avec ces chaînes. Cet exercice est à vocation pédagogique, en pratique on préférera utiliser les fonctions standard du C (`strcat,strcmp,strcpy...`) qui se trouvent dans le header standard `<cstring>` en C++.

On considère :

- une fonction `length` rendant une taille pour la chaîne de caractère.
- une fonction `newcopy` prenant une chaîne de caractère en argument et rendant une nouvelle copie de cette chaîne de caractères.

**Question 1.** Quel sont donc les signatures de ces fonctions ?

**Question 2.** Dans quel fichier et comment les déclarer, sachant qu'on veut en faire des fonctions utilitaires rangées dans le namespace "pr".

**Question 3.** Donnez deux implantations la fonction `length`, comparer une version utilisant la notation str[i] à une version en pointeurs pur. Où placer ce code d'implantation ?

**Question 4.** Donnez une implantation de la fonction `newcopy` utilisant une boucle (avec des pointeurs ou des index).

**Question 5.** Pour l'allocation, comparer l'utilisation de l'opérateur `new[]` du C++ à `malloc`. On supposera dans toute la suite que nos programmes n'utilisent que `new/new[]`.

**Question 6.** Construisez une version qui utilise `memcpy` au lieu d'une boucle.

> On rappelle la signature de *memcpy*, définie dans le header `<cstring>`.
>
> ```
> void* memcpy( void* dest, const void* src, std::size_t count );
> ```
>
> Copies count bytes from the object pointed to by *src* to the object pointed to by *dest*. Both objects are reinterpreted as arrays of unsigned char. If the objects overlap, the behavior is undefined.

**Question 7.** Ecrivez un programme dans un fichier `exo1.cpp` qui construit une copie d'une chaîne "Hello World", puis affiche les deux chaines, leurs addresses, leur longueurs, et sort **proprement** (pas de fuites mémoire).

**Question 8.** Expliquez comment compiler séparément ces fichiers puis les assembler (linker) faire un programme exécutable, et l'exécuter.

> On rappelle ici les principaux flags utilisés à la compilation :
> - -std=c++1y : dialecte utilisé, on pourrait avoir -std=c++11, -std=c++14, -std=c++17 ... La version 1y veut dire >= à c++11 sur notre compilateur.
> - -O0 : niveau d'optimisation, -O0 pour pouvoir debugger, -O2 ou -O3 en production
> - -g, -g3 : de plus en plus de symboles préservés dans le .o, pour faciliter le debug
> - -Wall : active tous les warnings de base, il y a d'autres warnings cela dit (cf -Wextra...)
> - -c : arrêter la compilation sur la production du .o
>
> Au link, on passe :
> - -o : output file, l'exe qu'on va produire
> - tous les .o construits à la compilation, l'un d'eux doit contenir un "main", pas de double définition.
> - les bibliothèques statiques libFoo.a
> - les bibliothèques dynamiques -lFoo pour libFoo.so/.dylib/.DLL

## 1.2 Une classe String

La gestion manuelle des pointeurs et de la mémoire pose des problèmes :

- accès hors d'une zone allouée (par dépassement, ou par accès à un pointeur non initialisé, ou par deref de `nullptr`),
- branchement sur une mémoire non allouée,
- double désallocation.

La structure de classe permet d'éviter un certain nombre de ces erreurs, en encapsulant ces comportements, de manière à assurer par exemple que les allocations et désallocations sont cohérentes. Le reste de cet exercice consiste à écrire une classe String qui se comporte "bien".

**Question 9.** Dans un fichier "String.h", définir une classe `String` minimale munie d'un attribut logeant un pointeur vers une chaîne du C et d'un constructeur permettant de positionner cet attribut. Ajoutez une opération membre "length" qui calcule la longueur de la chaîne. Donnez également le code de l'implantation de la classe, qu'on placera dans "String.cpp".

On doit pouvoir s'en servir de la manière suivante :

```
String str = "Hello";
size_t len = str.length();
```

**Question 10.** Ajoutez les mécanismes permettant d'imprimer une `String` à la manière C++ standard.

> La signature de l'opérateur standard pour imprimer un type MyClass :
>
> ```
> ostream & operator << (ostream & os, const MyClass & o)
> ```
>
> L'opérateur est binaire et prend deux arguments, ce qui est à sa gauche, et ce qui est à sa droite. A gauche donc, on s'attends à trouver un "ostream &", c'est à dire un flux de sortie, sur lequel on peut écrire.
>
> Le flux en question pourrait être une sortie standard, std::cout ou std::cerr (soit les stdout/stderr du C++). Mais ça pourrait également être un flux sur un fichier de sortie (voir <fstream>), ou un flux dans une zone mémoire (`stringstream`, déclaré dans <sstream>).
>
> Les deux arguments sont passés par référence ; le flux va être modifié (on va pousser des choses dedans), donc il n'est pas const. Au contraire l'affichage ne doit pas modifier la String, on peut donc utiliser une réference *const*.

> Par convention, l'opération rend le flux sur lequel on travaille, de manière à pouvoir chaîner les appels :
>
> `std::cout << str << " et " << str.length() << std::endl;`
>
> Donc on résoud à partir de la gauche, `std::cout << str` , ça produit un "ostream &" o1, on traite `o1 << " et "` etc...

**Question 11.** Le code actuellement proposé ne copie pas la chaîne passée en argument. Rappelez en quoi consiste le comportement par défaut qui est généré par le compilateur pour les opérations :

```
// dtor
virtual ~String();
// par copie de ori
String(const String &ori);
// mise à jour du contenu
String & operator=(const String & other);
```

Qu'affiche actuellement par exemple le programme suivant ?

```
String getAlphabet() {
      char [27] tab;
      for (int i=0; i < 26 ; ++i) {
            tab[i] = 'a'+i;
      }
      tab[26]='\0';
      return String(tab);
}
int main() {
      String abc = getAlphabet();
      std::cout << abc << std::endl;
}
```

Citer d'autres problèmes que cela peut poser.

**Question 12.** Ajoutez un constructeur qui copie la chaîne argument dans une zone nouvellement allouée, et un destructeur qui libère cette zone mémoire.

**Question 13.** Si l'on se contentait de ne modifier que le destructeur et le constructeur, expliquez les problèmes que cela pose sur cet exemple.

```
int main() {
      String abc = "abc";
      {
            String bcd(abc);
      }

      std::cout << abc << std::endl;

      String def = "def";
      def = abc;

      std::cout << abc << " et " << def << std::endl;
}
```

# TME 1 : Programmation, compilation et exécution en C++

**Objectifs pédagogiques :**

- **mise en place**
- **classe simple**
- **opérateurs**
- **compilation, debugger, valgrind**

## 1.1 Plan des séances

Cette UE de programmation avancée suppose une familiarité avec le C et un langage O-O comme Java (syntaxe de base, sémantique), les premières séances sont l'occasion de se remettre à niveau. Cet énoncé contient donc plusieurs encadrés, en gris, qui rappellent les notions clés à appliquer.

Comme le niveau est assez hétérogène, prenez le temps de bien absorber les concepts si nécessaire, nous pouvons prendre un peu de retard. Les TME proposent souvent des extensions dont les réalisations sont optionelles (bonus) mais qui permettent d'aller un peu plus loin.

Vous soumettrez l'état de votre TME à la fin de chaque séance, en poussant votre code sur un git. La procédure que l'on va mettre en place ce semestre n'étant pas encore opérationnelle pour cette séance, créez vous un git ou contentez vous de zipper le dossier "src" pour cette séance.

## 1.2 Prise en main

**Eclipse (CDT)** : un environnement de développement (IDE) configuré pour C/C++.
Cet environnement graphique gérera votre projet localement sur votre compte PPTI. Il vous permettra d'éditer les sources, de les compiler et de les exécuter. Son utilisation est fortement recommandée (instructions précises dans les supports), mais d'autres outils peuvent aussi être utilisés (NetBeans, Visual Studio Code, ...).

**Question 1.** Lancement d'Eclipse

Attention, plusieurs versions d'Eclipse cohabitent à la PPTI. Il faut utiliser une version pour développement CDT, qu'on a déployé dans "/usr/local/eclipseCPP/". La manière la plus sûre est d'ouvrir un terminal et d'y entrer la commande : `/usr/local/eclipseCPP/eclipse`.
Si c'est la première fois que vous utilisez Eclipse, celui-ci vous demande le nom du répertoire *workspace* où il placera par défaut vos projets. On recommande d'utiliser un nouveau dossier `~/workspacePR` comme dossier de workspace qui va loger les projets.

**Question 2.** Contruire un nouveau projet C++

Démarrer eclipse puis construire un "File->New->C/C++ Project"; on va utiliser le "C++ Managed Build" assez facile d'emploi et bien pris en charge par l'IDE comme système de build. On va construire pour commencer un projet hébergeant simplement un exécutable, prenez le template fourni "Hello World". Assurez-vous que la chaîne de compilation sélectionnée est bien "Linux GCC". Appelez le projet "TME0". On peut valider les options par défaut sur les autres onglets.

On a à présent un projet avec un main C++ qui affiche un message.

**Question 3.** Compiler le projet

Sélectionner le menu "Project->Build Project".

La "console CDT" (un des onglets dans la partie basse de l'IDE) montre les instructions de compilation qui ont été faites. On y voit une étape de compilation et une étape de link.

Eclipse a configuré une version compilée en mode Debug par défaut. Il place les makefile qu'il a engendré et les fichiers produits par la compilation dans un dossier Debug. A priori, on n'édite pas directement ces fichiers, mais plutôt les "Properties" du projet (clic droit sur le projet, dernier item).

Il propose aussi une version Release, compilée avec des flags plus optimisés, utiliser la flèche/triangle à côté du marteau (build) dans le ruban d'outils en haut de l'IDE pour basculer sur cette

configuration. Si vous ne voyez pas cet outil, assurez vous d'avoir basculé en Perspective C/C++ (avec le bouton dans le coin en haut à droite de eclipse). On va rester en configuration Debug pour l'instant.

Un nouvel élément "Binaries" est visible, il contient les binaires qu'on vient de construire. On peut clic-droit sur un binaire et faire "Run As...->Local C/C++ application". Cela lance le programme, dans la console d'eclipse.

**Question 4.** Ajoutez dans le main un un tableau "tab" de dix entiers que vous remplirez avec les entiers 0 à 9. Affichez le contenu du tableau.

On constate une bonne qualité du soulignement au cours de la frappe, et l'auto-complétion disponible sur ctrl-espace.

Certaines fautes ne sont pas soulignées au cours de la frappe, mais seulement si on lance le build complet. Par exemple cette erreur d'utilisation d'une variable non initialisée ne sera indiquée qu'après une compilation.

```
char * a;
cout << a ;
```

**Question 5.** Découverte du debugger

Recopier le code suivant dans votre main et lancer le programme. On rappelle que *size_t* désigne un type entier non signé de la taille d'un mot machine (8 octets sur une x64).

```
for (size_t i=9; i >= 0 ; i--) {
if (tab[i] - tab[i-1] != 1) {
cout << "probleme !";
}
}
```

Si "probleme" s'affiche, on sait qu'on a un souci. Double-cliquez dans la marge de l'éditeur, sur la ligne qui contient l'affichage de "probleme", l'outil crée un Breakpoint pour le debug.

Cliquez sur le binaire, mais cette fois-ci faites "Debug As..->Local C++ Application" au lieu de "Run As". Accepter de basculer en perspective Debug.

La ligne verte indique la position actuelle dans le code, on la fait évoluer en utilisant les outils dans le ruban du haut.

- Continue : poursuit l'exécution jusqu'au prochain breakpoint (qu'on place en double clic dans la marge)
- Step into, Step Over, Step Return : exécution ligne à ligne

Avec le breakpoint positionné sur notre message, avancer jusqu'à l'atteindre, puis inspecter les valeurs des variables (à droite). Pour l'affichage des pointeurs sous forme de tableau, faire un clic droit sur la variable et demander le mode tableau.

**Question 6.** Visualisation de l'état sous debugger

Quel est le contenu des cellules du tableau ? Combien vaut "i" ? Modifier la déclaration du type de "i", pour que la boucle aie effectivemet lieu 10 fois.

**Question 7.** Valgrind et détection des fautes mémoire

A présent, sans avoir complètement debuggé le problème, lancer une analyse avec valgrind. Sélectionner le binaire, et faire "Profiling Tools->Profile with Valgrind". Il doit vous aider à détecter vos erreurs (fautes mémoires et fuites mémoire). Les résultats sont visible dans l'onglet Valgrind.

Lire cette page http://valgrind.org/docs/manual/mc-manual.html#mc-manual.errormsgs décrivant les erreurs détectées par cet outil.

Standard C++ dans Eclipse CDT.
Par défaut la version du C++ utilisée est celle par défaut de notre compilateur, ce qui est insuffisant pour notre usage.
Il faut configurer le dialecte pour la version "-std=c++1y", soit la plus récente disponible. Malheureusement il faut faire ce réglage dans deux endroits :

- Pour l'éditeur/correction à la volée des erreurs, on règle une préférence globale, valable pour tous les projets d'un workspace donné. Naviguer vers
  - "Window -> Preferences -> C/C++ -> Build -> Settings"
  - Ouvrir l'onglet "Discovery".
  - Sélectionner dans la liste le "CDT GCC Built-in compiler Settings"
  - Ajouter un "-std=c++1y" au flags, de façon à avoir
    `${COMMAND} ${FLAGS} -std=c++1y -E -P -v -dD "${INPUTS}"`
- Pour le compilateur à proprement parler, on doit faire un réglage pour chaque projet séparément. En partant d'un clic droit sur le projet, accéder à ses propriétés:
  - "Project properties -> C/C++ Build -> Settings"
  - Sous le "GCC C++ compiler" on trouve une rubrique "Dialect"
  - sélectionner : "-std=c++1y" dans le menu déroulant.

Attention, il faut faire le premier réglage dans tout nouveau workspace, et le deuxième pour chaque nouveau projet.

## 1.3 Compilation, Link, Debug, Valgrind

*Notions testées : connaissance du langage, de la chaîne de compilation, des outils de debug (gdb, valgrind).*

Cet exercice est basé sur le partiel de Novembre 2018. Téléchargez le projet fourni https://pages.lip6.fr/Yann.Thierry-Mieg/PR/exo1.zip.

- On vous fournit une archive contenant un projet eclipse CDT, qu'il faudra modifier. Décompresser cette archive.
- Pour importer le projet dans Eclipse, le plus facile :"File->Import->General->Existing Projects into Workspace", Pointer le dossier `XXX/exo1`, Eclipse doit voir un projet, l'importer.
- Si vous préférez utiliser un autre IDE ou la ligne de commande, on vous fournit dans le répertoire source un Makefile trivial.

On vous fournit un programme `exo1` composé de trois fichiers, implantant une manipulation d'une liste chaînée de string. Malheureusement ce code contient un nombre important de bugs et d'erreurs.

List.h

```
#ifndef SRC_LIST_H_
#define SRC_LIST_H_

#include <cstddef>
#include <string>
#include <ostream>

namespace pr {

class Chainon {
public :
      std::string data;
      Chainon * next;
      Chainon (const std::string & data, Chainon * next=nullptr);
      size_t length() ;
      void print (std::ostream & os) const;
```

```
};

class List {
public:

      Chainon * tete;

      List(): tete(nullptr) {}

      ~List() {
            for (Chainon * c = tete ; c ; ) {
                  Chainon * tmp = c->next;
                  delete c;
                  c = tmp;
            }
      }

      const std::string & operator[] (size_t index) const ;

      void push_back (const std::string& val) ;

      void push_front (const std::string& val) {
            tete = new Chainon(val,tete);
      }

      bool empty() ;

      size_t size() const ;
};


std::ostream & operator<< (std::ostream & os, const List & vec) ;

} /* namespace pr */

#endif /* SRC_LIST_H_ */
```

List.cpp

```

namespace pr {

// ****************** Chainon
Chainon::Chainon (const std::string & data, Chainon * next):data(data),next(next) {};

size_t Chainon::length() {
      size_t len = 1;
      if (next != nullptr) {
            len += next->length();
      }
      return length();
}

void Chainon::print (std::ostream & os) {
      os << data ;
      if (next != nullptr) {
            os << ", ";
      }
      next->print(os);
}

```

```
// ****************** List
const std::string & List::operator[] (size_t index) const {
      Chainon * it = tete;
      for (size_t i=0; i < index ; i++) {
            it = it->next;
      }
      return it->data;
}

void List::push_back (const std::string& val) {
      if (tete == nullptr) {
            tete = new Chainon(val);
      } else {
            Chainon * fin = tete;
            while (fin->next) {
                  fin = fin->next;
            }
            fin->next = new Chainon(val);
      }
}

void List::push_front (const std::string& val) {
      tete = new Chainon(val,tete);
}

bool empty() {
      return tete == nullptr;
}

size_t List::size() const {
      if (tete == nullptr) {
            return 0;
      } else {
            return tete->length();
      }
}

} // namespace pr

std::ostream & operator<< (std::ostream & os, const pr::List & vec)
{
      os << "[";
      if (vec.tete != nullptr) {
            vec.tete->print (os) ;
      }
      os << "]";
      return os;
}
```

main.cpp

```
#include "List.h"
#include <string>
#include <iostream>
#include <cstring>

int main () {

      std::string abc = "abc";
      char * str = new char [3];
      str[0] = 'a';
```

```
      str[1] = 'b';
      str[2] = 'c';
      size_t i = 0;

      if (! strcmp (str, abc.c_str())) {
            std::cout << "Equal !";
      }

      pr::List list;
      list.push_front(abc);
      list.push_front(abc);

      std::cout << "Liste : " << list << std::endl;
      std::cout << "Taille : " << list.size() << std::endl;

      // Affiche à l'envers
      for (i= list.size() - 1 ; i >= 0 ; i--) {
            std::cout << "elt " << i << ": " << list[i] << std::endl;
      }

      // liberer les char de la chaine
      for (char *cp = str ; *cp ; cp++) {
            delete cp;
      }
      // et la chaine elle meme
      delete str;

}
```

**Question 8.** Identifiez et corrigez les erreurs dans ce programme.

On recherche au total

- Cinq fautes plutôt de nature syntaxiques empêchant la compilation ou le link
- Trois fautes graves à l'exécution entraînant le plus souvent un crash du programme
- Deux fautes de gestion mémoire incorrecte, mais ne plantant pas nécessairement le programme

Pour chaque faute, ajoutez un commentaire débutant par `//FAUTE :` et sur une ligne décrivez la faute au dessus de votre modification. Par exemple,

```
// FAUTE : i n'est pas initialisé
i = 0;
```

## 1.4 Réalisation d'une classe String

Cette partie "bonus" est à traiter en fonction du temps qu'il vous reste.

**Question 9.** Définir et tester les fonctions `length` et `newcopy` du TD1.

**Question 10.** Implanter la classe String conformément aux instructions du TD 1.

Au fur et à mesure de la réalisation de la classe, construire un `main` qui teste les éléments réalisés. S'assurer qu'il ne provoque aucune faute mémoire sous valgrind.

On commencera par réaliser et tester :

- Constructeur par copie de l'argument
- Destructeur qui invoque delete
- Ajout dans un flux / impression de la String
- Constructeurs par copie, operator= redéfinis

**Question 11.** Ajouter une fonction utilitaire `compare` aux fonctions utilitaires rangées dans "strutil.h". Elle doit se comporter comme la fonction `strcmp` standard, elle prend deux chaînes $a$ et $b$ du

C en argument, et elle rend une valeur négative si $a < b$, 0 si les deux chaînes sont logiquement égales, et une valeur positive sinon.

Plus précisément, on itére sur les deux chaînes simultanément (avec deux pointeurs) tant que les valeurs pointées sont égales et que la première est différente de '\0'. On compare ensuite les caractères pointés (on peut faire simplement la différence de leurs valeurs).

**Question 12.** Pour la comparaison entre deux String, on peut proposer soit des opérateurs membres, soit des fonctions extérieures à la classe (plus symétriques).

Ajouter (en s'appuyant sur `compare`) :

- Un opérateur de comparaison d'égalité `bool operator==(const String &a,const String &b)` symétrique, déclaré friend et en dehors de la classe String.
- Un opérateur fournissant une relation d'ordre `bool operator<(const String & b) const` membre de la classe String.

# TD 2 : Mémoire, Conteneurs, Map

**Objectifs pédagogiques :**

- **gestion mémoire, allocations**
- **vecteur<T>, liste<T>, map<K,V>**
- **introduction à la lib standard**

## 1.1 Copie et Affectation

On considère, une classe String définie comme suit :

String.h

```
#pragma once
#include <cstddef> // size_t
#include "strutil.h"

namespace pr {

class String {
      const char * str;
public:
      // ctor : copie
      String(const char *cstr=""): str(newcopy(cstr)){}
      // dtor : libère
      virtual ~String() { delete [] str;}

      // taille
      size_t length() const { return pr::length(str);}
};

} // fin namespace pr
```

**Question 1.** Expliquez les problèmes que cette version pose sur cet exemple.

```
int main() {
      String abc = "abc";
      {
            String bcd(abc);
      }

      std::cout << abc << std::endl;

      String def = "def";
      def = abc;

      std::cout << abc << " et " << def << std::endl;
}
```

**Question 2.** Modifiez la classe String, afin qu'elle se comporte bien : ajoutez les opérateurs d'affectation et la construction par copie.

**Question 3.** En conclusion, quelles opérations faut-il définir pour une classe $C$ qui stocke un pointeur en attribut dans le cas général ?

## Conteneurs

Dans ce TD, nous allons réaliser en C++ des classes `Vector<T>`,`List<T>`,`Map<K,V>` offrant le confort habituel de ces conteneurs classiques. Cet exercice est à vocation pédagogique, on préfér-

era en pratique utiliser les classes standard, qu'on trouvera dans les header `<vector>`, `<list>`, `<unordered_map>`.

L'objectif de l'exercice est de bien comprendre les conteneurs standards du c++ et leur API.

Pour être sûr de ne pas entrer en conflit avec d'autres applications, nous utiliserons le namespace `pr` pour notre implémentation.

## 1.2 Vecteur : stockage contigü.

Un vecteur stocke de façon contigüe en mémoire des données. C'est une des structures de données les plus simples, mais pour cette raison ça reste un bon choix pour de nombreuses applications.

**Question 4.** Ecrivez une classe `Vector<T>` en respectant les contraintes suivantes :

- Le vecteur est muni d'un pointeur vers l'espace mémoire alloué pour stocker les données
- Le vecteur est muni d'une taille `size`, qui indique le remplissage actuel
- Le vecteur a une taille d'allocation, toujours supérieure ou égale à size
- Les objets de type T ajoutés sont copiés dans l'espace mémoire géré par le vecteur

On fournira pour cette classe les operateurs et fonctions suivantes :

- `Vector(int size=10)` : un constructeur qui prend la taille d'allocation initiale
- `~Vector()` un destructeur
- `T& operator[](size_t index)` et `const T& operator[](size_t index) const` dans ses deux variantes (const ou non), pour consulter ou modifier le contenu.
- `void push_back (const T& val)` : ajoute à la fin du vecteur, peut nécessiter réallocation et copie.
- `size_t size() const` la taille actuelle (nombre d'éléments)
- `bool empty() const` vrai si la taille actuelle est 0

**Question 5.** Est-il utile de définir un constructeur par copie et un opérateur d'affectation pour Vector ?

**Question 6.** Quelle est la complexité de $push\_back$, $push\_front$ et de operator[] sur un vecteur de taille $n$ ?

## 1.3 Liste : stockage par Chainon.

Une liste simplement chaînée stocke les données dans des chaînons.

**Question 7.** Ecrivez une classe `List<T>`.

- Un Chainon est composé d'un attribut de type T (le data) et d'un pointeur vers le prochain Chainon (ou nullptr).
- Une Liste est munie d'un pointeur vers le premier chaînon qui la constitue (ou `nullptr` si elle est vide)
- La liste ne stocke pas sa taille, il faut la recalculer

On fournira pour cette classe les operateurs et fonctions suivantes :

- `List()` : un constructeur par défaut pour la liste vide
- `T& operator[](size_t index)` dans sa variante permettant de modifier le contenu.
- `void push_front (const T& val)` : ajoute en tête de la liste
- `~List()` un destructeur, qui libère tous les chaînons
- `void push_back (const T& val)` : ajoute à la fin de la liste
- `size_t size() const` la taille actuelle (nombre d'éléments)
- `bool empty() const` vrai si la liste est vide

**Question 8.** Est-il utile de définir un constructeur par copie et un opérateur d'affectation pour List ?

**Question 9.** Quelle est la complexité de $push\_back$, $push\_front$ et de operator[] sur une liste de taille $n$ ?

## 1.4 Table de Hash

On souhaite à présent implanter une table de hash assez simple, en appui sur les classes `forward_list<T>` et `vector<T>` du standard, et qui sont des versions plus complètes des classes qu'on vient de développer.

On rappelle les principes d'une table de hash :

- La table stocke un vecteur de taille relativement grande appelé *buckets*.
- Dans chaque case du vecteur, on trouve une liste de paires (clé,valeur)
- Pour rechercher une entrée à partir d'une clé $k$, on hash la clé, ce qui rend un entier $hash(k)$ dont la valeur dépend du contenu de la clé.
- On cherche dans le bucket d'indice `hash(k) % buckets.size()`, un élément de la liste dont la clé serait égale (au sens de ==) avec la clé $k$ recherchée.
- Si on le trouve, on peut exhiber la valeur qui lui est associée, sinon c'est qu'il n'est pas présent dans la table.

**Question 10.** Ecrire une classe générique `HashTable<K,V>` où $K$ est un paramètre qui donne le type de la clé, et $V$ donne le type des valeurs.

On pourra dans l'ordre :

- définir le cadre de la classe, ses paramètres génériques
- définir un type `Entry` pour contenir les paires clés valeur ; les clés sont stockées de façon const, pas les valeurs
- ajouter un attribut typé `vector< forward_list< Entry > >` dans la classe
- définir un constructeur prenant une taille, qui initialise le vecteur avec des listes vides dans chaque bucket

Ensuite définir les méthodes d'accès suivantes, dont la signature est proche de l'API proposée en Java (l'API C++ de `std::unordered_map` s'appuie sur les itérateurs, que l'on présentera au TD3).

- Définir `V* get(const K & key)` qui rend l'adresse de la valeur associée à la clé si on la trouve, ou `nullptr` dans le cas contraire. Pour calculer la valeur de hash de l'objet $key$, on utilisera `size_t h = std::hash<K>()(key);`. Cette fonction ne doit pas itérer toute la table, seulement le "bucket" approprié.
- Définir `bool put (const K & key, const V & value)` qui ajoute l'association $(key, value)$ à la table. La fonction rend vrai si la valeur associée à $key$ a été mise à jour dans la table, et faux si on a réalisé une insertion (la clé n'était pas encore dans la table).
- Une fonction `size_t size() const` qui rend la taille actuelle.

**Question 11.** Si la taille actuelle est supérieure ou égale à 80% du nombre de buckets, la table est considérée surchargée : la plupart des accès vont nécessiter d'itérer des listes. On souhaite dans ce cas doubler la taille d'allocation (nombre de buckets). Ecrivez une fonction membre `void grow()` qui agrandit une table contenant déjà des éléments. Quelle est la complexité de cette réindexation ?

# TME 2 : Conteneurs, Map, Lib Standard

**Objectifs pédagogiques :**

- **conteneurs**
- **map**
- **algorithm, lib std**

## 1.1 std::vector, std::pair

On part du programme suivant, qui extrait et compte les mots contenu dans un livre.

Compte le nombre mots

```
#include <iostream>
#include <fstream>
#include <regex>
#include <chrono>

int main () {
      using namespace std;
      using namespace std::chrono;

      ifstream input = ifstream("/tmp/WarAndPeace.txt");

      auto start = steady_clock::now();
      cout << "Parsing War and Peace" << endl;

      size_t nombre_lu = 0;
      // prochain mot lu
      string word;
      // une regex qui reconnait les caractères anormaux (négation des lettres)
      regex re( R"([^a-zA-Z])");
      while (input >> word) {
            // élimine la ponctuation et les caractères spéciaux
            word = regex_replace ( word, re, "");
            // passe en lowercase
            transform(word.begin(),word.end(),word.begin(),::tolower);

            // word est maintenant "tout propre"
            if (nombre_lu % 100 == 0)
                  // on affiche un mot "propre" sur 100
                  cout << nombre_lu << ": "<< word << endl;
            nombre_lu++;
      }
      input.close();

      cout << "Finished Parsing War and Peace" << endl;

      auto end = steady_clock::now();
   cout << "Parsing took "
           << duration_cast<milliseconds>(end - start).count()
           << "ms.\n";

   cout << "Found a total of " << nombre_lu << " words." << endl;

   return 0;
}
```

**Question 1.** Exécutez le programme sur le fichier WarAndPeace.txt fourni. Combien y a-t-il de mots ?

**Question 2.** Modifiez le programme pour compter le nombre de mots différents que contient le texte. Pour cela on propose dans un premier temps de stocker tous les mots rencontrés dans un vecteur, et de traverser ce vecteur à chaque nouveau mot rencontré pour vérifier s'il est nouveau ou pas. Exécutez le programme sur le fichier WarAndPeace.txt fourni. Combien y a-t-il de mots différents ?

**Question 3.** Modifiez le programme pour qu'il calcule le nombre d'occurrences de chaque mot. Pour cela, on adaptera le code précédent pour utiliser un vecteur qui stocke des `pair<string,int>` au lieu de stocker juste des string. Afficher le nombre d'occurrences des mots "war", "peace" et "toto".

**Question 4.** Que penser de la complexité algorithmique de ce programme ? Quelles autres structures de données de la lib standard aurait-on pu utiliser ?

## 1.2 Table de Hash

**Question 5.** En suivant les indications du TD 2, implanter la classe générique HashMap<K,V>.

## 1.3 Mots les plus fréquents

**Question 6.** Utiliser une table de hash `HashMap<string,int>` associant des entiers (le nombre d'occurrence) aux mots, et reprendre les questions où l'on calculait de nombre d'occurrences de mots avec cette nouvelle structure de donnée.

**Question 7.** Après avoir chargé le livre, initialiser un `std::vector<pair<string,int> >`, par copie des entrées dans la table de hash. On pourra utiliser le contructeur par copie d'une range : `vector (InputIterator first, InputIterator last)`.

**Question 8.** Ensuite trier ce vecteur par nombre d'occurrences décroissantes à l'aide de `std::sort` et afficher les dix mots les plus fréquents.

`std::sort` prend les itérateurs de début et fin de la zone à trier, et un prédicat binaire. Voir l'exemple suivant.

exemple sort et lambda

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

class Etu {
      public :
      std::string nom;
      int note;
};

int main_sort () {
      std::vector<Etu> etus ;
      // plein de push_back de nouveaux étudiants dans le désordre

      // par ordre alphabétique de noms croissants
      std::sort(etus.begin(), etus.end(), [] (const Etu & a, const Etu & b) { return a.nom <
           b.nom ;});
      // par notes décroissantes
      std::sort(etus.begin(), etus.end(), [] (const Etu & a, const Etu & b) { return a.note
          > b.note ;});
      return 0;
}
```

**Question 9.** S'il vous reste du temps, remplacer les `forward_list` et `vector` utilisés par votre classe HashMap par vos propres implantations `List,Vector` comme discuté en TD. Ajouter les opérations nécessaires au fonctionnement de votre table de hash dans ces classes.

# TD 3 : Iterator, Algorithm

**Objectifs pédagogiques :**

- **for each** $c++$
- **iterator, const_iterator**
- **algorithm**

## Introduction

Dans ce TD, nous allons ajouter des itérateurs à nos conteneurs et explorer quelques algorithmes de la lib standard.

## 1.1 Boucle For Each

## 1.2 Itérateurs.

On rappelle l'expansion syntaxique d'une structure de contrôle "for each" ou "range for" en C++11.

```
// boucle "foreach"
for (DeclType i : cont) {
      // body
}

// Expansion C++11
{
  for(auto it = cont.begin(),_end = cont.end(); it != _end; ++it)
  {
      DeclType i = *it;
            // body
  }
}
```

Dans ce code, `cont` est un conteneur, et `DeclType` est une déclaration de type compatible avec le contenu (`T, T&, const T&` si `cont` est un `conteneur<T>`).

**Question 1.** En déduire les méthodes que doit fournir un conteneur du C++, et les contraintes sur le type de retour de ces méthodes. Quelles opérations doit fournir un itérateur comme "it" dans ce code ?

**Question 2.** On considère le $Vector < T >$ développé au précédent TD. Si son itérateur se réduit simplement à un pointeur sur le contenu, montrez que la sémantique des opérateurs $!=, *, ++$ se conforme au contrat attendu. Enrichissez votre classe pour permettre une boucle "range-for" dessus.

**Question 3.** On considère la liste simplement chaînée `List<T>` développée au précédent TD. Si l'on suppose que son itérateur se limite à stocker à un pointeur de `Chainon`, comment aménager les opérateurs $!=, *, ++$ dans cette classe ? Enrichissez votre classe `List<T>` pour permettre une boucle "range-for" dessus.

## 1.3 Itérations constantes

**Question 4.** On considère la boucle suivante, qui permet d'afficher un vecteur. Le compilateur se plaint de problèmes liés au fait que le vecteur est *const*, et refuse de compiler ce code. Expliquez le problème qui se pose ici et ce qu'il faut faire pour le corriger.

```
template<typename T>
ostream & operator<< (ostream & os, const Vector<T> & vec) {
   for (const T & elt : vec) {
```

```
      os << elt << ", ";
   }
   os << endl;
}
```

**Question 5.** Expliquez les différences entre les signatures des opérations d'un `iterator` et d'un `const_iterator`. Expliquez comment il faut modifier Vector et List pour permettre des itérations const.

## 1.4 Algorithme et itérateur

**Question 6.** Ecrivez une fonction générique

```
template<typename iterator, typename T>
iterator find (iterator begin, iterator end, const T& target)
```

qui rend un itérateur *it* satisfaisant `target==*it` si l'on en trouve entre `begin` et `end`, ou `end` dans le cas contraire. Comment l'invoquer sur une `List<int>` ? Sur un `Vector<string>` ?

**Question 7.** La fonction `find_if` du standard prend un prédicat *pred* en troisième paramètre, c'est à dire une fonction, un foncteur, ou une lambda qui offre la signature : `bool matches (const T & elt)`.

Elle rend comme la fonction `find` un itérateur *it* satisfaisant `pred(*it)` si l'on en trouve entre `begin` et `end`, ou `end` dans le cas contraire.

Comment l'invoquer sur un `Vector<string>` pour trouver une string de trois caractères de long ?

**Question 8.** Comment l'invoquer pour trouver une string de longueur $n$ arbitraire, la valeur de $n$ étant connue au runtime ?

# TME 3 : Iterator, Algorithm

**Objectifs pédagogiques :**

- **iterator**
- **algorithm**
- **unordered_map**

On poursuit dans ce TME le travail démarré en séance 2. Inutile donc de créer un nouveau projet.

## 1.1 Algorithme

**Question 1.** Ecrivez une fonction générique `size_t count (iterator begin, iterator end)` qui compte le nombre d'éléments entre *begin* et *end*.

**Question 2.** Ecrivez une fonction générique `size_t count_if_equal (iterator begin, iterator end, const T & val)` qui compte le nombre d'éléments entre *begin* et *end* qui sont égaux à *val* au sens de ==.

**Question 3.** Invoquez ces fonctions dans votre programme qui compte les mots, et contrôlez le résultat : par exemple une invocation à *count* et la valeur rendue par *size* doivent être cohérentes.

## 1.2 Itérateurs sur HashMap

**Question 4.** Enrichir la classe `HashMap<K,V>` développée au précédent TD, pour permettre d'itérer sur son contenu. Le type contenu (i.e. ce qui est rendu par $*it$ sur un itérateur) sera une entrée dans la table, i.e. une paire clé valeur.

On donne les indications suivantes pour définir un itérateur (version non const), son opération de déréférencement, sa comparaison, son incrément.

En attribut,

- il porte une réference vers la table *buckets*, ou un itérateur positionné sur *buckets.end*() au choix
- il porte un indice ou un itérateur *vit* dans cette table *buckets*, désignant la liste (*bucket*) qu'il explore actuellement
- il porte un itérateur *lit* de liste, qui pointe l'élément courant dans la liste que désigne l'itérateur

En méthodes,

- `iterator & operator++()` pour l'incrémenter, on incrémente d'abord *lit*, si l'on est au bout de la liste, on se décale sur *vit* à la recherche d'une case non vide. *lit* devient alors la tête de cette liste (attention à ne paas déborder avec *vit* !).
- `bool operator!=(const iterator &other)` pour la comparaison d'inégalité, on compare les itérateurs *vit*, *lit*
- `Entry & operator*()` le déréférencement rend une *Entry*, celle qui est pointée par *lit*

**Question 5.** A l'aide de cet itérateur, recopier les entrées de votre table de hash dans un `std::vector` contenant des paires (clé,valeur).

**Question 6.** Enfin trier ce vecteur, par fréquence décroissante, et afficher les dix mots les plus fréquents du livre.

## 1.3 unordered_map

La classe `std::unordered_map<K,V>` est une table de hash plus complète que celle que l'on a implémenté. Son API utilise des itérateurs pour retrouver une clé avec *find* (joue le rôle de *get* dans notre implantation) et insérer une nouvelle paire clé valeur avec *insert* (joue le rôle de *put* de notre implantation).

- `iterator find (const K & key)` rend un itérateur correctement positionné, ou *end*() si la clé n'est pas trouvée.
- `std::pair<iterator,bool> insert (const Entry & entry)` qui essaie d'insérer la paire clé valeur fournie dans la table.
  - Si la clé était déjà présente dans la table, le booléen rendu est faux, et l'itérateur pointe l'entrée qui existait déjà dans la table (qui n'a pas été modifiée).
  - Si la clé n'était pas encore présente, le booléen rendu est vrai, et l'itérateur pointe l'entrée qui vient d'être ajoutée.

**Question 7.** Dans une copie de votre main, substituer une `unordered_map` à votre map actuelle et corriger les problèmes liés à cette API différente de la notre.

**Question 8.** On souhaite à présent inverser notre table associative : on souhaite créer une `unordered_map<int,forward_list<string> >`, où pour une fréquence donnée, on trouve la liste des mots ayant cette fréquence dans le livre. Construisez cette table, puis afficher tous les mots qui ont $N$ occurrences (choisissez des valeurs de $N$).

Pour construire la table, on itère la table actuelle les paires $(k, v)$, et l'on ajoute dans la table cible à la liste désignée par la clé $v$ avec $push\_front(k)$.

**Question 9.** Sur ce même modèle, si l'on a une classe `Personne` avec différents champs `nom, prenom, age, sexe, ...` et un `vector<Personne>`. Expliquez comment construire efficacement les sous-groupes de Personnes qui ont le même age, ou le même prénom.

NB: Cette opération orientée-objet très utile en pratique qui construit des classes d'équivalence d'objets vis à vis d'une de leurs propriétés est à rapprocher du *GROUP BY* de SQL.