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 ?

> `size_t length (const char *str);`
>
> Avec `size_t` un entier non signé représentant une taille, `#include <cstddef>` ou `#include <cstring>` pour le voir. Sa taille est plateforme dépendante, généralement compatible avec la taille d'un pointeur (e.g. size_t prend 8 bytes sur une x64).
>
> L'argument est certainement const, on ne souhaite pas le modifier. De plus les littéraux comme "abc" sont typés `const char *` et non `char*`.
>
> `char * newcopy (const char *str);`
>
> Le type de retour peut être `const char*` si on préfère qu'il ne soit pas modifiable ; cependant il paraît plus logique que l'on veuille parfois modifier la copie (sinon pourquoi copier ?). L'argument est certainement const, on ne souhaite pas le modifier.

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

> fichier strutil.h
>
> strutil.h
>
> ```
> #ifndef SRC_STRUTIL_H_                                                    1
> #define SRC_STRUTIL_H_                                                    2
> ```

```
#include <cstring>

namespace pr {

size_t length (const char *str);

char * newcopy (const char *str);

}


#endif /* SRC_STRUTIL_H_ */
```

NB : protection double include + namespace.

La protection double include peut être remplacée par `#pragma once` en tête de fichier. Elle évite les problèmes de dépendances doubles : *a.cpp* qui inclut *a.h* et *b.h*; *a.h* et *b.h* eux même incluent *c.h* -> sans protection double déclaration de *c.h*.

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 ?

On place dans le fichier "strutil.cpp", on inclut le *.h*, on encadre avec le namespace les déclarations.

strutil.cpp

```
#include "strutil.h"

#include <cstdlib>

namespace pr {


// version index
size_t length2 (const char *str) {
      size_t ret = 0;
      for (int i=0 ; str[i] ; ++i) {
            ++ret;
      }
      return ret;
}

// version pointeurs
size_t length3 (const char *str) {
      size_t ret = 0;
      for ( ; *str ; ++str) {
            ++ret;
      }
      return ret;
}

// version pointeurs + arithmétique (sans compteur)
size_t length (const char *str) {
      const char *cp=str;
      for ( ; *cp ; ++cp) {
      }
      return cp-str;
```

```
}
```

On n'hésite pas faire des dessins pour la version pointeurs.

(Les ++i au lieu de i++, c'est plus du style qu'autre chose, mais l'opérateur post fixé en général induit un temporaire, pas le préfixé. Bonne habitude en C++, pour la manipulation e.g. des itérateurs ça peut faire une différence. Ce n'est pas important.)

cf. ce "Best practice" tiré de c++ Primer :

***

Advice: Use Postfix Operators only When Necessary Readers from a C background might be surprised that we use the prefix increment in the programs we've written. The reason is simple: The prefix version avoids unnecessary work. It increments the value and returns the incremented version.The postfix operator must store the original value so that it can return the unincremented value as its result. If we don't need the unincremented value, there's no need for the extra work done by the postfix operator.

For ints and pointers, the compiler can optimize away this extra work. For more complicated iterator types, this extra work potentially might be more costly. By habitually using the prefix versions, we do not have to worry about whether the performance difference matters. Moreover—and perhaps more importantly—we can express the intent of our programs more directly.

***

Les tests de fin de boucle etc. sur $*cp$ ou cp est un $char*$, s'expandent en $*cp != '\0'$, mais la plupart des programmeurs C préfèrent la syntaxe plus compacte. Toute valeur différente de 0 est vrai dans les tests.

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

fichier strutil.cpp

strutil.cpp

```cpp
// version index
char * newcopy2 (const char *src) {
      size_t n = length(src);
      char * dest = new char[n+1];

      // avec <= pour attraper le dernier '\0'
      for (size_t i=0; i <= n ; ++i) {
            dest[i] = src[i];
      }

      return dest;
}

// une version pointeurs
// il y a beaucoup de variantes.
char * newcopy4 (const char *src) {
      size_t n = length(src);
      char * dest = new char[n+1];

      char *cd=dest ;
      while (true) {
            *cd = *src;
            if (! *src) {
                  break;
            }
            ++cd;
```

```
            ++src;
      }
      return dest;
}
```

On n'hésite pas faire des dessins pour la version pointeurs.

(Les ++i au lieu de i++, c'est plus du style qu'autre chose, mais l'opérateur post fixé en général induit un temporaire, pas le préfixé. Bonne habitude en C++, pour la manipulation e.g. des itérateurs ça peut faire une différence. Ce n'est pas important.)

Les tests de fin de boucle etc. sur $*cp$ ou cp est un $char*$, s'expandent en $*cp != '\0'$, mais la plupart des programmeurs C préfèrent la syntaxe plus compacte. Toute valeur différente de 0 est vrai dans les tests.

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[]`.

fichier strutil.cpp

strutil.cpp

```
// version index + malloc
char * newcopy3 (const char *src) {
      size_t n = length(src);
      // void * malloc (size_t sz)
      // pour allouer un tableau, multiplier la taille (sizeof) par le nombre de
          cellules
      // la signature force un cast pour remonter de void * vers le vrai type voulu
      char * dest = (char *) malloc( (n+1)*sizeof(char) );

      // avec <= pour attraper le dernier '\0'
      for (size_t i=0; i <= n ; ++i) {
            dest[i] = src[i];
      }

      return dest;
}
```

Le new[] du C++ est mieux typé que malloc : le type et la cardinalité sont donnés. Ici $sizeof(char)$ vaut 1, on le sait, mais c'est la syntaxe générale d'un malloc qui est utilisée.

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.

fichier strutil.cpp

strutil.cpp

```cpp


// version memcpy
// *toujours* la version à recommander en production
// habite dans #include <cstring>
char * newcopy (const char *src) {
      size_t n = length(src);
      char * dest = new char[n+1];

      // dest, source, nombre de bytes = n+1 avec le \0
      memcpy(dest,src,n+1);

      return dest;
}

}
```

memcpy est plus efficace que la boucle, dès qu'on a un peu de distance à copier. La copie est faite par blocs et gérée par l'OS, c'est ce qu'on peut faire de mieux pour copier le contenu de la mémoire d'un endroit à un autre. C'est une fonction essentielle, son API pointeur+ size_t en bytes pour la taille est caractéristique du C.

La dernière version du code à chaque fois est sans doute la plus proche de `strlen/strcpy`. Les "vraies" versions intègrent du code pour faire les tests mot par mot (e.g. 4 bytes) au lieu de char par char.

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).

fichier exo1.cpp

exo1.cpp

```cpp
/*
 * exo1.cpp
 *
 * Created on: Sep 11, 2018
 * Author: ythierry
 */


#include "strutil.h"

#include <iostream>

using namespace pr;
using namespace std;


int main12 () {
      const char * str = "Hello";
      char * copy = newcopy(str);

      cout << str << length(str) << endl;
      cout << copy << length(copy) << endl;

      // A ajouter dans un deuxieme temps
```

```
    delete [] copy;

    return 0;
}
```

Donc les “using” permettent d’alléger la syntaxe, sinon il faut “pr::length” et “std::cout, std::endl”.

On fait “return 0” si pas d’erreur, les codes de retour différents de 0 indiquent un problème (convention Unix).

A la fin du programme, “str” est libéré.

Les litéraux comme “Hello” sont stockés directement dans le segment de code, ils sont non modifiables (const char *), et ont une durée de vie égale à celle du programme (sauf s’il vient d’une lib dynamique chargée par dlopen, et qu’on la dlclose intempestivement.)

Sans le “delete[]” à la fin, on a une fuite mémoire, le “new[]/malloc” de “newcopy” n’a pas de “delete[]/free” symétrique.

Pour la détecter, on utilise valgrind, avec l’option “-leak-check=full”.

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

```
Invoking: GCC C++ Compiler

g++ -std=c++1y -O0 -g3 -Wall -c -o "src/strutil.o" "../src/strutil.cpp"

g++ -std=c++1y -O0 -g3 -Wall -c -o "src/exo1.o" "../src/exo1.cpp"


Invoking: GCC C++ Linker
g++  -o "td1"  ./src/exo1.o ./src/strutil.o

Execution
./td1

Valgrind
valgrind --leak-check=full --track-origins=yes ./td1
```

Donc à la compil, on produit un .o (fichier binaire compilé) par source .cpp, avec les flags

- -c : arrêter la compil sur la production du .o

- -o : output file, par défaut même nom que le source avec .o (donc ici par défaut allait bien)
- le fichier .cpp à compiler

En pratique on rajoute des flags pour que le compilo garde la trace des fichiers (include) dont dépendent le source compilé, il faut reconstruire le .o si n'importe lequel de ces fichiers inclus transitivement changent.

Au link, on assemble tous les morceaux, et on cherche un main.

Pas de dialecte à passer, ni de niveau d'optimisation. Cependant sur compilo récent des options comme –fwhole-program font un link optimisé.

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();
```

Donc on avance ligne par ligne dans le programme, et on ajoute ce qu'il faut à la classe String.

Cadre général, on déclare une classe dans le bon namespace :

```
#ifndef STRING_H_
#define STRING_H_

#include <cstddef> // pour size_t
#include <iosfwd> // pour ostream

// pas de "using" dans les .h
namespace pr {

class String {
// private tant qu'on a rien dit
const char * str;
public:
// tout ce qui suit est public
// un ctor
String (const char * ori);
// taille, membre public et const, i.e. ne modifie pas *this
size_t length() const;

}; // attention au point virgule pour terminer la déclaration de classe !

```

```
} // fin namespace pr

#endif /* STRING_H_ */
```

On note :

- les include, le namespace englobant
- visibilité : private tant qu'on n'a rien dit (accès aux membres de la classe seulement), public = tout le monde voit, protected = membres et membres des classes dérivées (mais PAS du namespace/package comme en Java)
- la classe n'a pas de notion de visibilité comme en Java, la notion de visibilité ne s'applique qu'aux membres des classes C++. Le reste se fait avec des namespace.
- l'attribut est déclaré "const char *", et sera donc non modifiable; on pourra changer de représentation interne ($str = autrechose$ OK) mais pas modifier str ($str[0] = '\backslash 0'$ NOK). C'est un choix qui est fait ici, qui nous rapproche des String de Java, immuables. La `std::string` est au contraire modifiable.
- l'invocation fournie `String str = "Hello";` indique l'existence d'un constructeur prenant un "const char *" en argument.
- pour le reste, cf les commentaires dans le code ci dessus

Et pour l'implantation :

```
#include "String.h"
#include "strutil.h" // pour length et newcopy

#include <iostream>

namespace pr {

String::String (const char * ori) : str(ori) {
}

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

} // fin namespace pr

#endif /* STRING_H_ */
```

On note :

- les include, le namespace englobant
- les opérations de String sont rangées dans son namespace, i.e. toute classe définit implicitement un namespace.
- l'opérateur :: permet de qualifier les noms, sans plus de précision on cherche dans le namespace courant
- "using std;" importe les noms dans le namespace courant; au lieu de "std::cout" on peut écrire juste "cout".
- sur le constructeur, on a une liste d'initialisation (entre le ':' et l'accolade ouvrante du ctor). Les attributs n'ont pas de valeur par défaut en C++ (non initialisé par défaut). Cette liste d'initialisation peut comporter plusieurs initialisations (séparés par des virgules), on y place également les appels aux constructeurs des classes parentes d'une classe dérivée. Enfin (standard c++11), on peut déléguer à un autre constructeur à cet endroit.
- sur length, pas de grosse surprise, on délègue sur la fonction de exo 1. Il faut include "strutil.h". Mais si on fait juste "return length(str)", on a un StackOverflow (on va par défaut trouver String::length) ! En qualifiant avec pr:: on tombe sur la bonne fonction de exo1.

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...

La signature standard pour un type MyClass :

```
ostream & operator << (ostream & os, const MyClass & o)
```

Donc on instancie ici pour String :

```
ostream & operator << (ostream & os, const String & s)
```

Niveau déclaration, comme l'opérande qui est à gauche est un stream standard, on ne peut pas l'enrichir (i.e. définir un "operator«" membre de ostream). On déclare donc l'opérateur de façon externe, dans le même namespace que String, mais à côté de la classe.

Le corps de l'opération est assez simple :

```
std::ostream & operator << (std::ostream & os, const pr::String & s) {
return os << s.str ;
}
```

Ce code utilise le fait que les "char *" du C ont déjà un cas particulier dans le standard, plus précisément ajouter un "char *" ou un "const char *" dans un flux l'interprète comme un string du C. Si on veut afficher vraiment l'adresse (et pas le contenu de la string) on peut faire :

```
os << (void *) s.str ;
```

.

Notons aussi que le standard définit déjà operator« pour tous les types simples et pas mal des types de la lib standard (e.g. std::string).

Le corps de l'opération accède directement à l'attribut "str" de la String, qui est private dans le contexte de l'opérateur. Pour résoudre ce problème on ajoute la déclaration "friend", dans la déclaration de la classe. Toute fonction déclarée friend peut accéder aux champs privés de la classe.

C'est à la classe de déclarer ses amis (elle rompt ainsi sélectivement son encapsulation), on ne peut pas se déclarer ami sans modifier la classe dont on veut être l'ami.

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.

Comportement par défaut du dtor = vide (mais invoque implicitement le super destructeur si héritage).

Comportement par défaut du constructeur par copie : affecter chacun des champs/attributs par copie.

Comportement par défaut de opérateur = : affecter par = tous les attributs de la classe.

Problèmes : la String ne maîtrise pas sa mémoire, elle n'encapsule pas correctement son comportement.

Exemple de problème sur l'exemple, le tableau est stack alloc dans le frame de getAlphabet, donc se fait nettoyer quand on pop le contexte. Notre String se retrouve à pointer qq part (d'illégal) sur la pile, ce qui provoque une faute mémoire quand va essayer de la coller dans cout (car on va lire la mémoire pointée, et même brancher sur son contenu en cherchant des '0').

Donc la faute, c'est qu'on a *free* la mémoire que je croyais pouvoir adresser. Autres fautes possibles, on me modifie le contenu pointé (rupture d'encapsulation).

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

Pour qu'elle soit responsable de sa mémoire, on va copier à la construction le char * qui m'est passé dans une zone nouvellement allouée. Et donc libérer cette mémoire quand on la détruit.

On ajoute donc :

```
String::~String() {
delete [] str;
}
String::String(const char *cstr) {
str = newcopy(cstr);
}
```

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;
}
```

Si on en reste là et qu'on a les versions par défaut du reste, on a des tas de problèmes. (NB : le compilo ne générera PAS de comportement par défaut si on fournit une des briques : ctor par copie, operator=, dtor...). Donc ici les problèmes ce serait des problèmes de compilation heureusement.

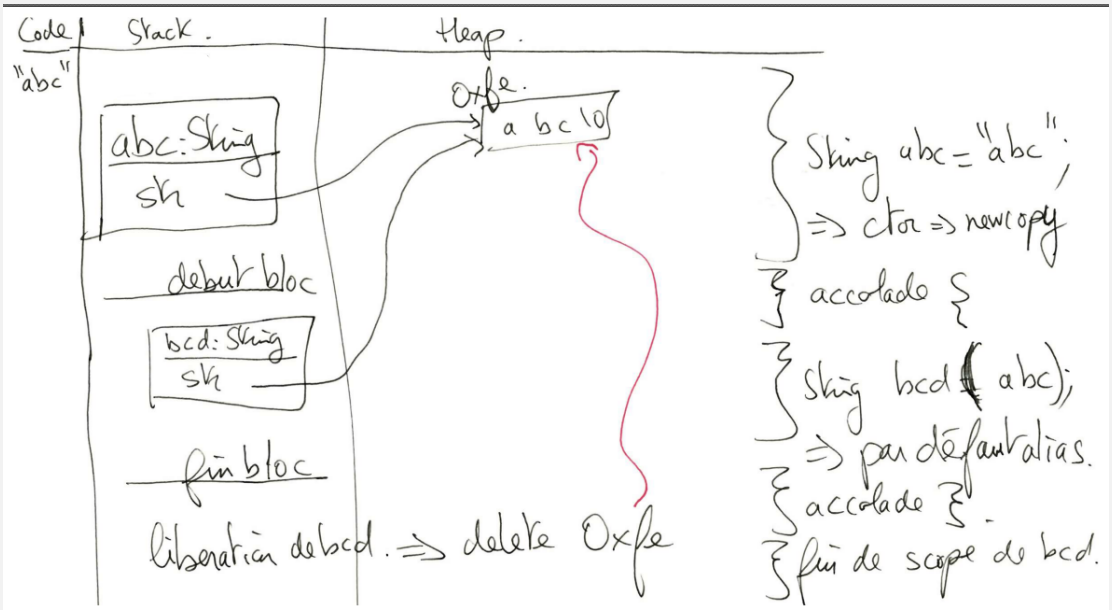
```
int main() {
String abc = "abc";// copie donc dans un bloc nouveau, on a fait le ctor
{
String bcd(abc); // ctor par copie par défaut = aliasing ! on pointe la même zone
} // fin de scope de bcd => dtor de bcd invoqué

std::cout << abc << std::endl; // faute mémoire, la zone est déjà  free

String def = "def"; // copie donc, on a fait le ctor
def = abc; // fuite mémoire, on perd l'addresse stockée dans def

std::cout << def << std::endl; // faute mémoire, la zone est déjà  free
} // faute mémoire double free sur dtor de abc, triple free avec le dtor de def
```

Il faut dessiner la mémoire et exécuter pas à pas pour expliquer. Ici un début de dessin.



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.

> c'est un size_t donc il ne sera jamais négatif.
>
> on utilise le type int à la place.

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.

> on doit avoir des erreurs de mémoire non initialisée.

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>
```

```cpp
#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,

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

main.cpp

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


int main () {
```

```
      std::string abc = "abc";
      char * str = new char [4];
      str[0] = 'a';
      str[1] = 'b';
      str[2] = 'c';
      // FAUTE : manque un '\0'
      str[3] = '\0';
      // FAUTE : size_t ne passera jamais en négatif
      int 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;
      }

      // FAUTE : NE SURTOUT PAS FAIRE CA !
      // liberer les char de la chaine
// for (char *cp = str ; *cp ; cp++) {
//  delete cp;
// }

      // et la chaine elle meme
      delete [] str;

}
```

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) ;

      // FAUTE : corps dans le cpp
      void push_front (const std::string& val) ;

      bool empty() ;

      size_t size() const ;


      class ConstListIte {
            const Chainon * cur;
      public:
            // default ctor = end()
            ConstListIte (const Chainon * cur=nullptr) : cur(cur) {}

            const auto & operator* () const {
                  return cur->data;
            }
            const auto * operator-> () const {
                  return & cur->data;
            }
            ConstListIte & operator++ () {
                  cur = cur->next;
                  return *this;
            }
            bool operator!= (const ConstListIte &o) const {
                  return cur != o.cur;
            }
      };

      typedef ConstListIte const_iterator;

      const_iterator begin() const {
            return tete;
      }
      const_iterator end() const {
            return nullptr;
      }
};
```

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

} /* namespace pr */

#endif /* SRC_LIST_H_ */
```

List.cpp

```
// FAUTE : manque include
#include "List.h"

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();
      }
      // FAUTE : stack overflow !
      return len;
}

// FAUTE : manque qualifier const
void Chainon::print (std::ostream & os) const {
      os << data ;
      if (next != nullptr) {
            os << ", ";
            // FAUTE MEMOIRE : acces nullptr
            next->print(os);
      }
}


// List

const std::string & List::operator[] (size_t index) const {
      auto it = begin();
      for (size_t i=0; i < index ; i++) {
            ++it;
      }
      return *it;
}

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);
}

// FAUTE : manque List::
bool List::empty() {
      return tete == nullptr;
}

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

// FAUTE : doit etre dans le namespace pr
std::ostream & operator<< (std::ostream & os, const pr::List & vec)
{
      os << "[";
      if (vec.tete != nullptr) {
            vec.tete->print (os) ;
      }
      os << "]";
      return os;
}

} // namespace pr
```

Les fautes 10 chaque :

- Manque include "List.h"
- print est const dans .cpp
- List::empty est qualifié
- namespace pr inclut operator«
- double implantation de push front
- length provoque un SO
- print provoque un NPE
- main contient une boucle infinie
- abc alloué sans '0' terminal
- delete correct du tableau abc

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

Inclut aussi la réponse aux question suivantes.

String.h

```
#ifndef STRING_H_
#define STRING_H_

#include <cstddef> // pour size_t
#include <iosfwd> // pour ostream

// pas de "using" dans les .h

namespace pr {

class String {
      const char * str;

      // déclaration d'accès friend
      friend String operator+ (const String &s1, const String & s2);
      friend bool operator== (const String &s1, const String & s2);
      friend std::ostream & operator << (std::ostream &os, const pr::String & s);

public:
      // ctor par copie de cstr
      // + déclaration d'une conversion : const char * -> const String &
      // + ctor par défaut à chaîne vide
      String(const char *cstr="");
      // dtor
      virtual ~String();
      // par copie de ori
      String(const String &ori);
      // mise à jour du contenu
      String & operator=(const String & other);

      // accès (non modifiable !)
      char operator[] (size_t index) const;
      // accès en modification (illégal, on stocke un const char *)
      // permettrait de faire : str[0]='a';
      // char & operator[] (size_t index);

      // taille
      size_t length() const;
      // égalité logique
      bool operator<(const String & o) const;
};

// encore dans namespace pr.

// déclaration d'un opérateur + symétrique (non membre)
String operator+ (const String &s1, const String & s2);
// déclaration d'une sérialisation par défaut (toString du c++)
std::ostream & operator << (std::ostream &os, const pr::String & s);
// comparaison d'égalité
bool operator== (const String &s1, const String & s2);

} // fin namespace pr

```

```
#endif /* STRING_H_ */
```

String.cpp

```
#include "String.h"
#include "strutil.h" // pour length et newcopy

#include <iostream>

namespace pr {


String::~String() {
      delete [] str;
}

String::String (const String & ori) : String(ori.str) {
}

String & String::operator=(const String & other) {
      if (this != &other) {
            delete[] str;
            str = newcopy(other.str);
      }
      return *this;
}

String::String(const char *cstr) {
      str = newcopy(cstr);
}

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

String operator+ (const String &s1, const String & s2) {
      size_t sz1 = s1.length();
      size_t sz2 = s2.length();
      // + 1 pour le '\0'
      char res [sz1+ sz2 +1];

      if (true) {
            // version memcpy
            memcpy(res,s1.str,sz1);
            // + 1 pour le '\0'
            memcpy(res+sz1,s2.str,sz2+1);
      } else {
            // version à la main
            char * pr = res;
            for (const char * p1=s1.str ; *p1 ; ++p1,++pr) {
                  *pr = *p1;
            }
            for (const char * p2=s2.str ; *p2 ; ++p2,++pr) {
                  *pr = *p2;
            }
            *pr = '\0';
      }

      return String (res);
```

```
}

bool operator==(const String & a,const String &b) {
      return ! compare(a.str,b.str);
}

bool String::operator<(const String & o) const {
      return compare(str,o.str) < 0;
}

char String::operator[] (size_t index) const {
      return str[index];
}

// Version modifiable, illégale si on stocke un const char *
//char & String::operator[] (size_t index) {
// // Cette ligne induit un cast qui perd le const du char pointé
// return str[index];
//}

std::ostream & operator << (std::ostream & os, const pr::String & s) {
      return os << s.str ;
}

} // namespace pr
```

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).

strutil.h

```
#ifndef SRC_STRUTIL_H_
#define SRC_STRUTIL_H_

#include <cstring>

namespace pr {

size_t length (const char *str);

char * newcopy (const char *str);

int compare (const char *a, const char * b);

}


#endif /* SRC_STRUTIL_H_ */
```

strutil.cpp

```cpp
#include "strutil.h"

#include <cstdlib>

namespace pr {


// version pointeurs + arithmétique (sans compteur)
size_t length (const char *str) {
      const char *cp=str;
      for ( ; *cp ; ++cp) {
      }
      return cp-str;
}


// version memcpy
// *toujours* la version à recommander en production
// habite dans #include <cstring>
char * newcopy (const char *src) {
      size_t n = length(src);
      char * dest = new char[n+1];

      // dest, source, nombre de bytes = n+1 avec le \0
      memcpy(dest,src,n+1);

      return dest;
}


int compare (const char *cp, const char * cq) {
      for ( ; *cp == *cq && *cp; ++cp,++cq) {
      }
      return *cp - *cq;
}

}
```

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;
}
```

Si on en reste là et qu'on a les versions par défaut du reste, on a des tas de problèmes. (NB : le compilo ne générera PAS de comportement par défaut si on fournit une des briques : ctor par copie, operator=, dtor...). Donc ici les problèmes ce serait des problèmes de compilation heureusement.

```
int main() {
String abc = "abc";// copie donc dans un bloc nouveau, on a fait le ctor
{
```

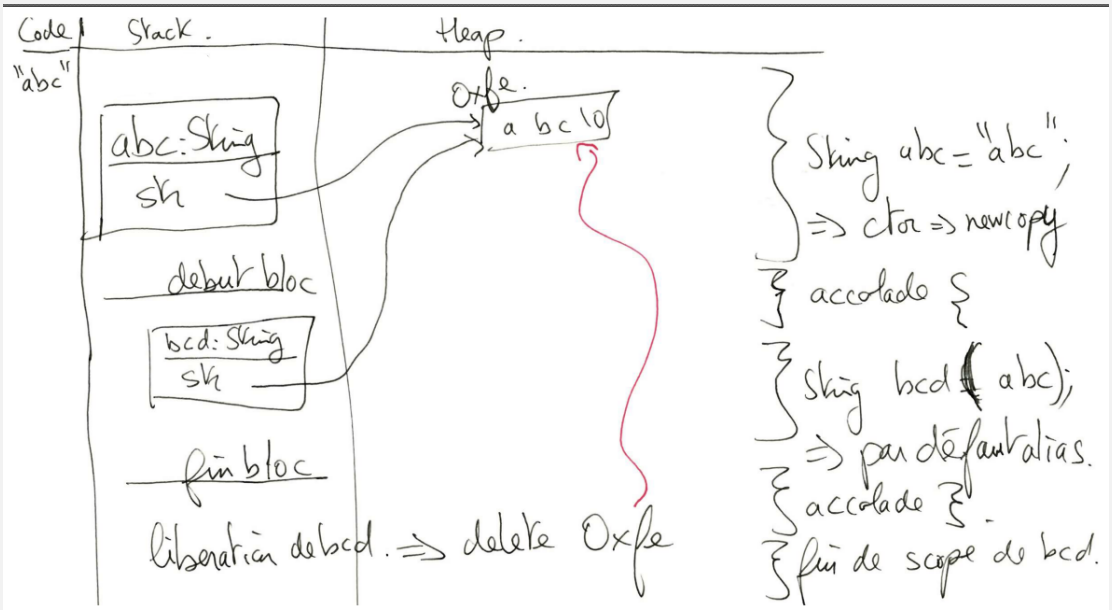
```
String bcd(abc); // ctor par copie par défaut = aliasing ! on pointe la même zone
} // fin de scope de bcd => dtor de bcd invoqué

std::cout << abc << std::endl; // faute mémoire, la zone est déjà  free

String def = "def"; // copie donc, on a fait le ctor
def = abc; // fuite mémoire, on perd l'addresse stockée dans def

std::cout << def << std::endl; // faute mémoire, la zone est déjà  free
} // faute mémoire double free sur dtor de abc, triple free avec le dtor de def
```

Il faut dessiner la mémoire et exécuter pas à pas pour expliquer. Ici un début de dessin.



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

On corrige avec la classe complétée ainsi (on peut bien sûr faire tout ça dans le cpp)

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;}
      // ctor par copie, implanté par délégation sur le ctor à partir de const char *
      String(const String & other):String(other.str){}
      // ou sans utiliser de délégation
      // String(const String & other):str(newcopy(other.str)){}
```

```
      // operator =
      String & operator=(const String & other) {
            if (this != & other) {
                  delete[] str;
                  str = newcopy(other.str);
            }
            return *this;
      }

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

} // fin namespace pr
```

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

On doit donc avoir :

- Un constructeur, qui assure que la mémoire pointée nous appartient (e.g. copie les arguments)
- Un constructeur par copie, qui copie la mémoire pointée
- Un destructeur, qui va delete le pointeur
- Un opérateur d'affectation, qui libère l'espace actuel, et copie

+ les variantes move.

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 préférera 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

Au niveau de la résolution,

1. on fait d'abord ctor et dtor.
2. Ensuite on fait les operator[] ; on commence par le cas non const supportant $vec[12] = 42$.

On peut comparer le code avec pointeurs au références C++ :

```
// membre de
T* accessAt (size_t index) {
return & tab[index]; // ou : tab+index
}
//client
*vec.accessAt(12) = 42;
```

avec des refs :

```
// membre de
T& operator[] (size_t index) {
return tab[index]; // ou : *(tab+index)
}
//client
vec.operator[](12) = 42; // ou vec[12]=42;
```

Morale : la ref est un peu comme un pointeur tout le temps déréférencé.

Ensuite on fait la version *const*, en supposant

```
void affiche1 (const Vector<string> & vec) {
std::cout << vec[1];
}
```

Donc on est forcés de noter que *operator*[] est const (mot clé après la déclaration de la fonction), car vec est *const* dans cet situation.

Du coup, on est forcés de rendre une *const&* au lieu d'une ref.

3. On fait le push_back, et son ensureCapacity

Vector.h

```
#ifndef SRC_VECTOR_H_
#define SRC_VECTOR_H_

#include <cstddef>
#include <ostream>
#include <algorithm> // std::max

namespace pr {

template <typename T>
class Vector {
      size_t alloc_sz;
      T * tab;
```

```
      size_t sz;

      void ensure_capacity(size_t n) {
            if (n >= alloc_sz) {
                  // double alloc size
                  alloc_sz = std::max(alloc_sz * 2,n); // juste si: n > 2 * alloc_sz
                  T * newtab = new T[alloc_sz]; // attention, construit par défaut les
                      instances
                  for (size_t ind=0; ind < sz ; ind++) {
                        // on utilise operator =, et pas memcpy, effet différent si T
                            est compliqué.
                        newtab[ind] = tab[ind];
                        // idéalement c'est plutôt std::move qu'il faudrait utiliser ici
                            .
                  }
                  delete[] tab;
                  tab = newtab;
            }
      }

public:
      Vector(int size=10): alloc_sz(size),sz(0) {
            tab = new T [alloc_sz];
      }
      virtual ~Vector() {
            delete [] tab;
      }
      const T & operator[] (size_t index) const {
            return tab[index];
      }
      T& operator[] (size_t index) {
            return tab[index];
      }
      void push_back (const T& val) {
            ensure_capacity(sz+1);
            tab[sz++]=val;
      }
      size_t size() const { return sz ; }
};

} /* namespace pr */

#endif /* SRC_VECTOR_H_ */
```

4. On peut juste discuter size et empty, sans trop insister.

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

OUI !
la classe stocke un pointeur détenu par l'instance, qu'on delete dans le dtor.

Donc sur le modèle discuté pour string, il faut écrire ces fonctions.

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

operator[] est O(1) (super efficace).
push_back est O(1), tant qu'on ne réalloue pas, mais O(N) sur réallocation.
Cependant en pratique, N push_back d'affillée produira une complexité O(N ln N), car on double la taille à chaque débordement (1000 insertions => 10 réalloc).
push_front est O(N), même sans réallocation (horrible).

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

On ne corrige pas tout in extenso.
1. Le constructor de List.
2. Operator[] : on voit comment itérer la liste.
3. push front : ajout/alloc d'un chainon
4. destructeur : selon le temps disponible
5. les autres on les évoque un peu, sans présenter les détails.

List.h

```
#ifndef SRC_LIST_H_
#define SRC_LIST_H_

#include <cstddef>
#include <ostream>

namespace pr {

template <typename T>
class List {

      class Chainon {
      public :
            T data;
            Chainon * next;
            Chainon (const T & data, Chainon * next=nullptr):data(data),next(next) {};
      };
```

```
      Chainon * tete;


public:
      List(): tete(nullptr) {
      }
      virtual ~List() {
            for (Chainon * cur = tete ; cur != nullptr ; ) {
                  Chainon * tmp = cur;
                  cur = cur->next;
                  delete tmp;
            }
      }

      T& operator[] (size_t index) {
            Chainon * cur = tete;
            for (size_t i=0; i < index ; i++) {
                  cur = cur->next;
            }
            return cur->data;
      }

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

      void push_front (const T& val) {
            tete = new Chainon(val,tete);
      }

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

      size_t size() const {
            size_t sz = 0;
            for (Chainon * cur = tete ; cur != nullptr ; cur = cur->next) {
                  sz++;
            }
            return sz;
      }
};

} /* namespace pr */

#endif /* SRC_LIST_H_ */
```

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

OUI !
la classe stocke un pointeur détenu par l'instance, qu'on delete dans le dtor.

Donc sur le modèle discuté pour string, il faut écrire ces fonctions. Attention la copie doit bien copier toute la liste, pas juste le premier chaînon.

Question 9. Quelle est la complexité de *push_back*, *push_front* et de operator[] sur une liste de taille n ?

operator[] est O(N) (horrible).
push_back est O(N), on doit retrouver la queue de liste. Certaines impléms stockent tete **ET** queue pour avoir cette opération pas cher O(1).
push_front est O(1) (super)

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.

Le TME va demander de faire cette classe, le but est donc de ne pas donner trop de code, ou ils vont recopier, mais leur donner assez pour bootstrap leur développement.

1. on donne le code complet du cadre : paramètres template, includes, classe Entry ou pair, attributs.
2. on esquisse *get* en pseudo-code / dessin : hash la clé, modulo nombre de buckets, parcours de liste, return si bonne clé.
3. on parle un peu de *put* sans pseudo-code.

HashMap.h

```
#ifndef SRC_HASHMAP_H_
#define SRC_HASHMAP_H_

#include <cstddef>
#include <ostream>

#include <forward_list>
#include <vector>

namespace pr {

template <typename K, typename V>
class HashMap {

public:
      class Entry {
      public :
            const K key;
            V value;
            Entry(const K &k,const V& v):key(k),value(v){}
      };
private :

      typedef std::vector<std::forward_list<Entry> > buckets_t;
      // storage for buckets table
      buckets_t buckets;
      // total number of entries
      size_t sz;

public:
      HashMap(size_t size): buckets(size),sz(0) {
            // le ctor buckets(size) => size cases, initialisées par défaut.
      }

      V* get(const K & key) {
            size_t h = std::hash<K>()(key);
            size_t target = h % buckets.size();
            for (Entry & ent : buckets[target]) {
                  if (ent.key==key) {
                        return & ent.value;
                  }
            }
            return nullptr;
      }

      bool put (const K & key, const V & value) {
            size_t h = std::hash<K>()(key);
            size_t target = h % buckets.size();
            for (Entry & ent : buckets[target]) {
                  if (ent.key==key) {
                        ent.value=value;
```

```
                        return true;
                    }
                }
                sz++;
                buckets[target].emplace_front(key,value);
                return false;
        }

        size_t size() const { return sz ; }

        void grow () {
                HashMap other = HashMap(buckets.size()*2);
                for (auto & list : buckets) {
                        for (auto & entry : list) {
                                other.put(entry.key, entry.value);
                        }
                }
                buckets = other.buckets;
                // NB : ce serait mieux de faire move ici
                // buckets = std::move(other.buckets);
        }
};

} /* namespace pr */

#endif /* SRC_HASH_H_ */
```

NB: une version plus complète avec plus de commentaires de cette classe est présente dans le corrigé du TME3.

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 ?

Donc on initialise une nouvelle table, on itère sur this, on insère tout ce qu'on rencontre dans la nouvelle table (avec insert).

Bilan : il faut recalculer tous les hash et les modulo.

Vu qu'on sait ce qu'on est en train de faire, et que la propriété de clé est présente initialement, on peut se contenter de `push_front` sur le bon bucket, et ne jamais itérer sur une liste dans l'algo. Ce n'est pas fait dans cette implémentation naïve.

Ca reste une opération chère, en mémoire (le double d'une taille déjà calculé pour être un peu grosse) et en temps (rehasher toutes les clés).

Cf corrigé précédent.

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 ?

> Sur ma machine, j'ai
>
> ```
> Parsing War and Peace
> Finished Parsing War and Peace
> Parsing with took 17751ms.
> Found a total of 20333 words.
> ```

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".

> On doit trouver :
>
> ```
> Frequency : war:298
> Frequency : peace:114
> Word :toto not found.
> ```

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

> Oui, ben c'est pas terrible, on est en $O(N^2)$, pour chaque mot rencontré, au pire des cas on scanne tous les mots qu'on a déjà vus.
> Le hash (unordered_map) offre du $O(N)$ si on a pas trop de collisions, et pas besoin de réindexer (dimensionnement initial adéquat).

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.

> On a beau avoir fait une hashtable un peu naive, on doit avoir divisé les temps par 10 à 100.
> Corrigé de cette série de questions dans le code à la fin.

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

```cpp
#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;
}
```

main_hash.h

```cpp

#include "HashMap.h"
#include <iostream>
#include <fstream>
#include <regex>
#include <algorithm>
#include <vector>
#include <chrono>

using namespace std;
using namespace pr;


void printFrequency (const string & word, HashMap<string,int> & map) {
      auto it = map.find(word);
      if (it != map.end()) {
            std::cout << "Frequency : " << it->key << ":" << it->value << endl;
      } else {
            std::cout << "Word :"<< word << " not found." << endl;
      }

}
```

```
int main () {

      HashMap<std::string,int> map(2<<18); // 256 k word capacity
      ifstream input = ifstream("/tmp/WarAndPeace.txt");

      std::chrono::steady_clock::time_point start = std::chrono::steady_clock::now();
      std::cout << "Parsing War and Peace" << endl;
      std::string s;
      regex re( R"([^a-zA-Z])");
      while (input >> s) {
            s = regex_replace ( s, re, "");
            std::transform(s.begin(),s.end(),s.begin(),::tolower);
            auto ret = map.insert({s,1});
            if (! ret.second) {
                  (*ret.first).value++;
            }
      }
      input.close();
// std::cout << "Counts :" << map << endl;
      std::cout << "Finished Parsing War and Peace" << endl;

      std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
   std::cout << "Parsing with hash took "
         << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()
         << "ms.\n";

   start = std::chrono::steady_clock::now();
   std::cout << "Parsing War and Peace" << endl;
   vector<pair<string,int> > counts;
   counts.reserve(5000);
   input = ifstream("/tmp/WarAndPeace.txt");
   while (input >> s) {
      s = regex_replace ( s, re, "");
      std::transform(s.begin(),s.end(),s.begin(),::tolower);
      auto it = counts.begin();
      while (it != counts.end()) {
            if (it->first == s) {
                  break;
            }
            ++it;
      }
      if (it != counts.end()) {
            it->second++;
      } else {
            counts.push_back(make_pair(s,1));
      }
   }
   input.close();
   end = std::chrono::steady_clock::now();
   std::cout << "Parsing with pair<string,int> took "
             << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).
                count()
             << "ms.\n";

      printFrequency("war",map);
      printFrequency("peace",map);
      printFrequency("toto",map);

      start = std::chrono::steady_clock::now();
```

```
      std::cout << "Most frequent" << endl;                                                    82
      std::vector<pr::HashMap<std::string,int>::Entry * > entries;                             83
      entries.reserve(map.size());                                                             84
      for (auto it = map.begin() ; it != map.end() ; ++it) {                                   85
            entries.push_back(& (*it) );                                                       86
      }                                                                                        87
      std::sort(entries.begin(),entries.end(), [] (auto a,auto b) { return a->value > b        88
          ->value ;});
      int line=0;                                                                              89
      for (const auto & val : entries) {                                                       90
            std::cout << val->key << "->" << val->value << " ,";                             91
            if (++line % 10 == 0) {                                                          92
                  std::cout << endl;                                                         93
            }                                                                                94
            if (line == 100) {                                                               95
                  break;                                                                     96
            }                                                                                97
      }                                                                                        98
      std::cout << std::endl;                                                                  99
      end = std::chrono::steady_clock::now();                                                  100
   std::cout << "Sorting by frequency took "                                                 101
          << std::chrono::duration_cast<std::chrono::microseconds>(end - start).count()      102
          << "us.\n";                                                                        103
                                                                                             104
   return 0;                                                                                 105
}                                                                                            106
```

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 ?

Donc l'itération la plus "standard" est sûrement le foreach ou range-for,

On voit qu'il faut :

- begin et end membres du conteneur, fonctions qui rendent des "itérateurs". On voit simplement que les types rendus par ces méthodes begin/end doivent être compatibles (une seule déclaration de type dans le for pour les deux). Cette contrainte est allégée en C++17, et disparait complètement en C++20 cependant (cf ici pour la doc :https://en.cppreference.com/w/cpp/language/range-for).
- L'itérateur doit supporter
  - la comparaison d'inégalité (operator !=), pour comparer *it* à *end*. Cet itérateur *end* "past the end" permet est le critère d'arrêt de l'itération.
  - l'opérateur de déréférencement (operator*, parfois confortable d'avoir aussi operator->). On rend a priori des références sur le contenu du conteneur.
  - le préincrément, (operator++), pour décaler notre itérateur entre chaque tour deboucle.
- L'itérateur pourrait être un type simple, en particulier, les pointeurs nus satisfont le contrat d'un itérateur (on reviendra sur cette remarque quand on traite Vector).

Pour référence ici l'expansion Java, mais je ne pense pas qu'on ait le temps de présenter ce contenu en TD. Ca reste intéressant de comparer si on connaît bien l'API de Java autour de Iterable.

```
// boucle "foreach"
// contrainte : cont implements Iterable<DeclType> => offre : Iterator<DeclType>
    iterator().
for (DeclType i : cont) {
// body
}

// Expansion Java 5+
{
  for(Iterator<DeclType> it = cont.iterator() ; it.hasNext() ; /*NOP*/)
  {
      DeclType i = it.next();
// body
  }
}
```

On voit que l'API est assez différente, la principale différence étant la présence explicite d'une notion de range en C++, on est entre begin et end, alors qu'en Java si on ne veut pas tout itérer c'est plus compliqué (cf "java.util.ArrayList.subList" cependant), de base on ne me donne que le début de l'itération, c'est l'itérateur lui même qui décide d'arrêter ($!hasNext()$).

Au niveau de l'itérateur, celui de C++ sépare $*$ pour accéder de ++ pour décaler : un itérateur pointe toujours un élément (sauf *end* qui est au delà du dernier). En Java, l'itérateur est entre deux éléments, et *next* décale et déréférence en une seule opération.

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.

Donc effectivement, $! =$ permet de comparer deux adresses, $*$ sur un pointeur qui vise une case du vecteur permet d'accéder au contenu de la case, et ++ sur un pointeur de T décale le pointeur de $sizeof(T)$ ce qui est pile la taille d'une cellule de notre vecteur quel que soit le type T.

Ce n'est pas par hasard qu'on a cette concordance, le standard est délibérément formulé pour que le pointeur soit l'itérateur le plus simple du C++.

L'itérateur est donc tout bêtement un pointeur dans le tableau, pas besoin de définir de classe. Un bon `typedef T* iterator;` (dans la partie public) est cependant une bonne idée, pour que `Vector<T>::iterator` soit défini pour les clients. On masque ainsi notre implantation, et on a des signatures homogènes dans List et Vector pour begin et end (utilisant la définition locale de $Vector < T >:: iterator$ ou $List < T >:: iterator$)

Donc on ajoute `iterator begin()` et `iterator end()`, qui rendent bêtement des $T*$ en pratique.

Begin va rendre un pointeur sur le début de la zone du tableau, end va rendre un pointeur un au delà du dernier élément.

begin rend *tab*, c'est à dire le pointeur vers le début de la zone allouée et stockée par le Vector.

end rend $tab + size$, c'est à dire un pointeur un au delà de la dernière case remplie actuellement (NB: on regarde *size* et pas $alloc\_size$ ici).

Vector.h

```
#ifndef SRC_VECTOR_H_
#define SRC_VECTOR_H_

#include <cstddef>
#include <ostream>

```

```
namespace pr {

template <typename T>
class Vector {
      size_t alloc_sz;
      T * tab;
      size_t sz;

      void ensure_capacity(size_t n) {
            // cf TD2
      }

public:
      Vector(int size=10): alloc_sz(size),sz(0) {
            tab = new T [alloc_sz];
      }
      virtual ~Vector() {
            delete [] tab;
      }

      const T & operator[] (size_t index) const {
            return tab[index];
      }
      T& operator[] (size_t index) {
            return tab[index];
      }

      void push_back (const T& val) {
            ensure_capacity(sz+1);
            tab[sz++]=val;
      }

      size_t size() const { return sz ; }

      // Declarations et implantations non-const
      typedef T* iterator;
      iterator begin() {
            return tab;
      }
      iterator end() {
            return tab + sz;
      }

      // Declarations et implantations const
      typedef const T* const_iterator;
      const_iterator begin() const {
            return tab;
      }
      const_iterator end() const {
            return tab+sz;
      }
};

} /* namespace pr */

#endif /* SRC_VECTOR_H_ */
```

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.

Cf le code fourni. On présente la version non const dans un premier temps.

Donc on commence par se définir une classe au sein de List, qu'on appelle "iterator", et qui a en attribut un pointeur de Chainon. L'itérateur stocke donc un pointeur de `Chainon *cur;`.

`T & operator*()` rend le *data* stocké dans le chaînon actuel. On le rend par référence, de manière à pouvoir le modifier e.g. `* list.begin() = 42 ;` .

On peut aussi fournir `operator->` cohérent avec `operator*` pour plus de confort, mais ce n'est pas obligatoire pour faire un for-each.

`iterator & operator++()` décale notre itérateur d'un chaînon, donc fait `cur = cur->next`. On rend `*this`, par convention pour permettre de chaîner les appels, mais on pourrait rendre *void* ça ne changerait pas grand chose.

Pour l'inégalité on compare les pointeurs stockés dans les deux itérateurs : si on pointe le même chaînon, on est égaux, sinon pas.

Au niveau de la liste elle même, il faut offrir begin et end.

Begin rend un itérateur qui stocke un pointeur qui pointe la tete de la liste. Niveau syntaxe, on pourrait se contenter de faire "return tete", vu la le type de retour dans la signature de begin, et la présence d'un ctor a un seul argument dans la classe "iterator".

End rend un itérateur dont le pointeur stocké vaut "nullptr".

On voit que par construction, après un $++$ sur le dernier chaînon de la liste qui se résoud en $cur = cur- > next$, le pointeur *cur* stocké dans l'itérateur vaudra *nullptr*. On va donc obtenir quand on dépasse la fin de liste un itérateur qui stocke "nullptr", donc c'est la bonne condition d'arrêt.

List.h

```
#ifndef SRC_LIST_H_
#define SRC_LIST_H_

#include <cstddef>

namespace pr {

template <typename T>
class List {

      class Chainon {
      public :
            T data;
            Chainon * next;
            Chainon (const T & data, Chainon * next=nullptr):data(data),next(next) {};
      };
      Chainon * tete;


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

      void push_front (const T& val) {
```

```
            tete = new Chainon(val,tete);
      }

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

      class iterator {
            Chainon * cur;
      public:
            // default ctor = end()
            iterator(Chainon * cur=nullptr) : cur(cur) {}

            T & operator* () const {
                  return cur->data;
            }
            // pas strictement necessaire
            T * operator-> () const {
                  return & cur->data;
            }

            iterator & operator++ () {
                  cur = cur->next;
                  return *this;
            }
            bool operator!= (const iterator &o) const {
                  return cur != o.cur;
            }
      };

      iterator begin() {
            return iterator(tete);
      }
      iterator end() {
            return iterator(nullptr);
      }

      class const_iterator {
            const Chainon * cur;
      public:
            // default ctor = end()
            const_iterator(const Chainon * cur=nullptr) : cur(cur) {}

            const T & operator* () const {
                  return cur->data;
            }
            const T * operator-> () const {
                  return & cur->data;
            }

            const_iterator & operator++ () {
                  cur = cur->next;
                  return *this;
            }
            bool operator!= (const const_iterator &o) const {
                  return cur != o.cur;
            }
      };
```

```
      const_iterator begin() const {
            return const_iterator(tete);
      }
      const_iterator end() const {
            return const_iterator(nullptr);
      }
};

} /* namespace pr */

#endif /* SRC_LIST_H_ */
```

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;
}
```

Donc ce code invoque implicitement un "begin" membre sur vec.

On recherche donc à la compilation une fonction "begin" avec une signature compatible.

Actuellement, on trouve la signature membre : `iterator begin()`, donc une fonction au final `iterator begin(Vector<T> * this)`.

(Rappelons qu'une fonction membre de Vector, possède implicitement un premier argument *this* typé $Vector*$ si la méthode n'est pas const, et $constVector*$ si l'on suffixe la déclaration de la fonction par le mot clé "const".)

On a l'expression `vec.begin()` à évaluer, où `vec` est un `const Vector<T> &`. On cherche donc une signature `XXX begin(const Vector * this)` et il n'y en a pas.

On ne peut pas convertir un $constT*$ (read only) vers $T*$ (read AND write) (par contre l'inverse est automatique et normal), donc faute de compilation.

Autrement dit, l'expansion du foreach recherche : begin et end membres, mais qui soient de plus déclarés *const*, car l'argument *vec* dans cette fonction d'affichage est passé via une *const&*.

Et donc on n'en trouve pas; nos begin et end ne sont pas const.

Si on ajoute simplement "const" après la déclaration de *begin* et *end*, le code de notre fonction d'affichage compile, mais par contre la classe Vector ne compile plus !

Le code de `begin` fait un accès dans `this->tab`. Comme `this` est const, tous les champs pointés par `this` le sont aussi ; donc `this->tab` est un `const T*` dans le contexte de `iterator begin() const`.

Autrement dit, le fait de rendre un $T*$ (non const) qui vise le contenu de *tab*, stocké dans *this* qui est maintenant *const* induit une faute de compilation. Il est illégal dans une fonction membre *const* de construire des références ou pointeurs non const vers le contenu (en déréférencant *this* qui est $const*$ on ne tombe que sur des champs *const*).

Il faut donc également que ces opérations rendent des pointeurs *const* pour pouvoir compiler.

Morale : les signatures const et non const interfèrent entre elles et ne sont pas compatibles au final. On est forcés en pratique à dupliquer les opérations begin/end et les itérateurs liés dans une version const

et une version non const.

Par contre, une fois que c'est fait, le compilateur sélectionnera automatiquement la bonne variante, pour le client de la classe c'est transparent normalement le fait que les signatures soient dupliquées.

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.

Donc `operator*` est celui dont la signature change : au lieu de rendre `T &` on va rendre `const T &` non modifiable.

On note un élément assez embêtant, une même classe ne peut avoir ces deux méthodes, les signatures ne différant que par le type de retour ce qui est illégal.

Donc on est forcés de dupliquer le code des itérateurs, en déclinant une version const et une version non const (comme on a fait au TD 2 pour `operator[]`).

Sur Vector c'est juste rajouter un typedef de plus `typedef const T * const_iterator`, et dupliquer les begin/end en versions const essentiellement.

Sur Liste il faut aussi dupliquer la classe itérateur entière malheureusement.

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>` ?

Donc on va itérer et tester avec ==. La fonction `std::find` existe et fait ça.

```
template< typename iterator, typename T>
iterator find (iterator begin, iterator end, const T& target) {
   while (begin != end) {
      if (*begin == target) {
         break;
      }
      ++begin;
   }
   return begin;
}

void foo() {
   List<int> list; list.push_front(10);
   Vector<string> vec; vec.push_back("toto");
   if (find(list.begin(),list.end(),12) != list.end()) {
      cout << "trouve";
   }
   auto it = find(vec.begin(),vec.end(),"toto");
   if (it != vec.end()) {
      *it = "tata"; // en modification
   }
}
```

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 ?

Version pointeur de fonction :

```
bool matches (const string & s) {
   return s.length() == 3;
}
void foo() {
   Vector<string> vec; vec.push_back("toto"); //... autres insertions
   auto it = find_if(vec.begin(),vec.end(),matches);
}
```

Version foncteur (pas un grand intérêt ici) :

```
struct Matcher {
 bool operator() (const string & s) {
   return s.length() == 3;
 }
}
void foo() {
   Vector<string> vec; vec.push_back("toto"); //... autres insertions
   Matcher m;
   auto it = find_if(vec.begin(),vec.end(),m);
}
```

Version lambda :

```
void foo() {
   Vector<string> vec; vec.push_back("toto"); //... autres insertions

   auto it = find_if(vec.begin(),vec.end(),
      [](const string & s) {
         return s.length() == 3;
      });
}
```

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

Version pointeur de fonction, on est mal : à moins d'une variable globale n on ne s'en sort pas. La version variable globale n'est pas réentrante ce qui est un (gros) défaut, surtout dans un cadre concurrent.

```
// global
int n;
bool matches (const string & s) {
   return s.length() == n;
}
void foo(int longueur) {
```

```
   Vector<string> vec; vec.push_back("toto"); //... autres insertions
   n=longueur;
   auto it = find_if(vec.begin(),vec.end(),matches);
}
```

Version foncteur (prend tout son intérêt ici) :

```
struct Matcher {
 int n;
 Matcher(int n):n(n){}
 bool operator() (const string & s) {
   return s.length() == n;
 }
}
void foo(int longueur) {
   Vector<string> vec; vec.push_back("toto"); //... autres insertions
   Matcher m (longueur);
   auto it = find_if(vec.begin(),vec.end(),m);
}
```

Version lambda :

```
void foo(int longueur) {
   Vector<string> vec; vec.push_back("toto"); //... autres insertions

   auto it = find_if(vec.begin(),vec.end(),
      [longueur](const string & s) {
         return s.length() == longueur;
      });
}
```

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 ==.

Corrigé Q1 et Q2.

utils.h

```
#ifndef UTILS_H_
#define UTILS_H_

#include <string>

namespace pr {

template<typename iterator>
size_t count (iterator begin, iterator end) {
      size_t sz = 0;
      while (begin != end) {
            ++begin;
            ++sz;
      }
      return sz;
}

template<typename iterator, typename T>
size_t count_if_equal (iterator begin, iterator end, const T& target) {
      size_t sz = 0;
      while (begin != end) {
            if (*begin == target) {
                  ++sz;
            }
            ++begin;
      }
      return sz;
}

}

#endif /* UTILS_H_ */
```

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.

main.cpp

```cpp
#include "utils.h"
#include <iostream>
#include <vector>
#include <string>
#include <cassert>
using namespace std;



int main2() {
      vector<string> vec;
      vec.push_back("toto");
      vec.push_back("tata");
      vec.push_back("toto");

      assert(vec.size() == pr::count(vec.begin(), vec.end()));
      assert(2 == pr::count_if_equal(vec.begin(), vec.end(),"toto"));
      assert(1 == pr::count_if_equal(vec.begin(), vec.end(),"tata"));

      return 0;
}
```

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*

On a ajouté les itérateurs + pas mal de commentaires

HashMap.h

```cpp
#ifndef SRC_HASHMAP_H_
#define SRC_HASHMAP_H_
```

```cpp

#include <cstddef>
#include <ostream>

#include <forward_list>
#include <vector>

namespace pr {

template <typename K, typename V>
class HashMap {

public:
	// on pourrait aussi typedef pair<K,V> Entry;
	// avec .first =.key et .second=.value
	// i.e. le choix fait dans std::unordered_map
	class Entry {
	public :
		// on pourrait utiliser const K ici, mais c'est génant pour les copies
		K key;
		V value;
		Entry(const K &k,const V& v):key(k),value(v){}
	};
private :

	typedef std::vector<std::forward_list<Entry> > buckets_t;
	// storage for buckets table
	buckets_t buckets;
	// total number of entries
	size_t sz;

public:
	HashMap(size_t size): buckets(size),sz(0) {
		// NB : le ctor buckets(size) => size éléments construits par défaut => listes vide
	}

	V* get(const K & key) {
		// std::hash<K>() => instancier un struct possédant : size_t operator()(const K &)
		// attention aux doubles parentheses
		size_t h = std::hash<K>()(key);
		size_t target = h % buckets.size();
		// iteration seulement sur la liste cible
		for (Entry & ent : buckets[target]) {
			if (ent.key==key) {
				return & ent.value;
			}
		}
		return nullptr;
	}

	bool put (const K & key, const V & value) {
		// début idem get
		size_t h = std::hash<K>()(key);
		size_t target = h % buckets.size();
		for (Entry & ent : buckets[target]) {
			if (ent.key==key) {
				// hit : mettre à jour la valeur associée
```

```
                    ent.value=value;
                    return true;
                }
            }
            // miss : faire une insertion
            sz++;
            // la liste n'est pas triée, donc insertion en tete O(1)
            buckets[target].emplace_front(key,value);
            return false;
        }

        size_t size() const {
            // maintenu à jour dans put
            return sz ;
        }

        void grow () {
            HashMap other = HashMap(buckets.size()*2);
            // on profite des itérateurs de vector et list
            for (auto & list : buckets) {
                for (auto & entry : list) {
                    other.put(entry.key, entry.value);
                }
            }
            // affectation : libère buckets actuel, copie other
            // buckets = other.buckets;
            // NB : il vaut mieux faire move ici, other est un temporaire
            buckets = std::move(other.buckets);
        }

        class iterator {
            // les typename ici sont obligatoires ; buckets_t générique n'est pas pleinement connu à ce stade
            // du coup buckets_t::identifiant pourrait être n'importe quoi, e.g. un nom de fonction, un attribut
            // on doit aider le compilo en précisant que c'est un nom de type dit "nested".

            // pour borner les itérations sur vit (itérateur fixe).
            // On pourrait a la place détenir un pointeur ou une ref à buckets, et utiliser buckets.end()
            typename buckets_t::iterator buckend;
            // le curseur qui indique le bucket (case du vector) qu'on est en train d'itérer
            typename buckets_t::iterator vit;
            // le curseur qui indique le chainon de liste dans ce bucket où on en est
            typename std::forward_list<Entry>::iterator lit;
        public :
            // ctor : copie des arguments
            iterator(const typename buckets_t::iterator & buckend, const typename buckets_t::iterator & vit, const typename std::forward_list<Entry>::iterator & lit)
              :buckend(buckend), vit(vit),lit(lit){}

            iterator & operator++() {
                // on se décale dans la liste courante
                ++lit;
                // test : faut il changer de liste/bucket
                if (lit == vit->end()) {
```

```
                        // on se décale d'un bucket pour commencer
                        ++vit;
                        // on continue de se décaler tant que les buckets sont vides
                        while (vit->empty() && vit != buckend) {
                              ++vit;
                        }
                        // si on déborde, c'est fini, on sortira avec un vit==buckend
                        if (vit != buckend) {
                              // sinon, on positionne lit au début de la liste dans ce
                                  bucket
                              lit = vit->begin();
                        }
                  }
                  // par convention
                  return *this;
            }
            Entry & operator*() {
                  // on déreference simplement lit, iterateur de liste de Entry
                  // donc on obtient un Entry
                  return *lit;
            }
            bool operator!=(iterator other) {
                  return vit != other.vit || lit != other.lit;
            }
      };
      iterator begin() {
            // on doit chercher le premier bucket non vide s'il existe
            typename buckets_t::iterator vit = buckets.begin();
            while (vit->empty() && vit != buckets.end()) {
                  ++vit;
            }
            if (vit != buckets.end()) {
                  // on a trouve au moins un bucket non vide
                  return iterator(buckets.end(),vit,vit->begin());
            } else {
                  // ce map est vide ! TODO : on aurait du tester size pour s'en
                      apercevoir tt de suite
                  return end();
            }
      }
      iterator end() {
            // la valeur du troisième itérateur ne change pas grand chose (c'est un
                iterator qui porte nullptr)
            return iterator(buckets.end(),buckets.end(),buckets.front().end());
      }
};

} /* namespace pr */

#endif /* SRC_HASH_H_ */
```

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.

main.cpp

```cpp
#include "HashMap.h"
#include <string>
#include <algorithm>
#include <iostream>
using namespace std;
using namespace pr;

int main3 () {
      HashMap<string, int> map(100);
      map.put("tutu",12);
      map.put("toto",3);
      map.put("tata",5);

      vector<HashMap<string,int>::Entry> vec;
      // copie dans le vecteur
      for (auto & e : map) {
            vec.push_back(e);
      }
      // tri
      sort(vec.begin(),vec.end(),[](const auto &a,const auto &b) { return a.value > b.
          value;});

      for (auto & e : vec) {
            cout << "mot : " << e.key << " freq : " << e.value << endl;
      }
      return 0;
}
```

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.

main.cpp

```cpp
#include "HashMap.h"
#include <iostream>
```

```cpp
#include <fstream>
#include <regex>
#include <algorithm>
#include <vector>
#include <chrono>
#include <unordered_map>

using namespace std;
using namespace pr;

int main_myhash() {
      HashMap<std::string,int> map(2<<18); // 256 k word capacity
      ifstream input = ifstream("/tmp/WarAndPeace.txt");

      std::chrono::steady_clock::time_point start = std::chrono::steady_clock::now();
      std::cout << "Parsing War and Peace" << endl;
      std::string s;
      regex re( R"([^a-zA-Z])");
      while (input >> s) {
            s = regex_replace ( s, re, "");
            std::transform(s.begin(),s.end(),s.begin(),::tolower);
            auto ret = map.get(s);
            if (ret != nullptr) {
                  (*ret)++;
            } else {
                  map.put(s,1);
            }
      }
      input.close();
// std::cout << "Counts :" << map << endl;
      std::cout << "Finished Parsing War and Peace" << endl;

      std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
   std::cout << "Parsing with my hash took "
         << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()
         << "ms.\n";
   return 0;
}

int main_stdhash() {
      unordered_map<std::string,int> map(2<<18); // 256 k word capacity
      ifstream input = ifstream("/tmp/WarAndPeace.txt");

      std::chrono::steady_clock::time_point start = std::chrono::steady_clock::now();
      std::cout << "Parsing War and Peace" << endl;
      std::string s;
      regex re( R"([^a-zA-Z])");
      while (input >> s) {
            s = regex_replace ( s, re, "");
            std::transform(s.begin(),s.end(),s.begin(),::tolower);
            auto ret = map.find(s);
            if (ret != map.end()) {
                  ret->second++;
            } else {
                  map.insert(make_pair(s,1));
            }
      }
      input.close();
// std::cout << "Counts :" << map << endl;
```

```
      std::cout << "Finished Parsing War and Peace" << endl;

      std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();
   std::cout << "Parsing with std hash took "
         << std::chrono::duration_cast<std::chrono::milliseconds>(end - start).count()
         << "ms.\n";
   return 0;
}

int main4 () {
      main_myhash();
      main_stdhash();

   return 0;
}
```

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 $push\_front(k)$.

invert.h

```
#ifndef INVERT_H_
#define INVERT_H_

#include <unordered_map>
#include <forward_list>
#include <vector>
#include <functional>

template <typename K, typename V>
std::unordered_map<V, std::forward_list<K> > invert (const std::unordered_map<K,V> & src
    ) {
      std::unordered_map<V, std::forward_list<K> > dest;
      for (auto & e : src) {
            // NB operator[](key) => crée une entrée avec la valeur construite par dé
                faut si k n'existe pas
            // donc dest[e.second] => soit une belle liste vide, soit la liste cible.
            dest[e.second].push_front(e.first);
      }
      return dest;
}

// group by prenom
template <typename T>
std::unordered_map<std::string, std::forward_list<T> > groupBy (const std::vector<T> &
    src) {
      std::unordered_map<std::string, std::forward_list<T> > dest;
      for (auto & e : src) {
            // NB operator[](key) => crée une entrée avec la valeur construite par dé
                faut si k n'existe pas
            // donc dest[e.second] => soit une belle liste vide, soit la liste cible.
```

```
            dest[ e.getPrenom() ].push_front(e);
      }
      return dest;
}


#endif /* INVERT_H_ */
```

group.cpp

```
#include "invert.h"
#include <string>
#include <iostream>

using namespace std;

class Person {
      string nom;
      string prenom;
public :
      Person(const string & nom, const string & prenom):nom(nom),prenom(prenom){}
      const string & getPrenom() const { return prenom ;}
      const string & getNom() const { return nom ;}
};

int main() {
      vector<Person> pop;
      pop.emplace_back("dupon","toto");
      pop.emplace_back("duron","toto");
      pop.emplace_back("cudon","tata");

      for (auto & e : groupBy(pop)) {
            cout << "prenom :" << e.first << endl;
            for (auto & p : e.second) {
                  cout << p.getNom() << ", ";
            }
            cout << endl;
      }
}
```

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.

cf corrigé précédent.