TD 1 : Rappels de programmation C et orientée objet

Objectifs pédagogiques :

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

Introduction

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

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

1.1 Rappels chaîne du C, const.

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

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

On considère :

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

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

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

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

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

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

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

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

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

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

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

1.2 Une classe String

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

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

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

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

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

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

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

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

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

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

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

Qu'affiche actuellement par exemple le programme suivant ?

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

Citer d'autres problèmes que cela peut poser.

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

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

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

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

      String def = "def";
      def = abc;

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

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

Objectifs pédagogiques :

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

1.1 Plan des séances

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

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

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

1.2 Prise en main

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

Question 1. Lancement d'Eclipse

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

Question 2. Contruire un nouveau projet C++

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

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

Question 3. Compiler le projet

Sélectionner le menu "Project->Build Project".

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

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

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

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

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

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

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

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

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

Question 5. Découverte du debugger

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

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

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

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

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

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

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

Question 6. Visualisation de l'état sous debugger

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

Question 7. Valgrind et détection des fautes mémoire

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

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

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

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

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

1.3 Compilation, Link, Debug, Valgrind

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

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

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

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

List.h

```
#ifndef SRC_LIST_H_
#define SRC_LIST_H_

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

namespace pr {

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

```
};

class List {
public:

      Chainon * tete;

      List(): tete(nullptr) {}

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

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

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

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

      bool empty() ;

      size_t size() const ;
};


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

} /* namespace pr */

#endif /* SRC_LIST_H_ */
```

List.cpp

```

namespace pr {

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

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

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

```

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

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

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

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

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

} // namespace pr

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

main.cpp

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

int main () {

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

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

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

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

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

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

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

}
```

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

On recherche au total

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

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

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

1.4 Réalisation d'une classe String

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

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

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

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

On commencera par réaliser et tester :

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

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

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

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

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

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

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