Projet de programmation générique.

Julien Allali : allali@univ-mlv.fr

Cache Générique et Smart Pointer.

1 Introduction.

Ce sujet comporte deux objectifs:

1. La mise en place d'un concept (ou type abstrait) de cache et l'implémentation de celui-ci.

2. L'écriture d'un programme utilisant le cache mis en place précédemment.

Nous verrons plus tard que l'implémentation de certains types de caches imposent l'utilisation des *Smart Pointer*.

2 Qu'est-ce qu'un cache?

L'objectif d'un cache est de fournir une interface de chargement pour des objets d'un même type. Il est aussi chargé de détruire les objets en fonction de certaines contraintes, la taille totale du cache en mémoire par exemple.

Le buffer cache du système est un objet de ce type. Il fournit une interface pour le chargement de donnée depuis le disque. Le type de donnée dans ce cache est un bloc de bits. Lorsque l'on demande un bloc, s'il est déjà chargé, le cache le renvoie tout de suite. Sinon, le cache fait appel à un chargeur externe qui lui renvoie le bloc. Dans le cas où l'on atteint la limite du cache (taille totale qu'il est autorisé à prendre en mémoire), si il y a des blocs non utilisés, on libère le plus ancien, sinon on indique qu'il n'y a plus de place.

Un autre exemple de cache, est le cache de pages Web dans les navigateurs. Lorsqu'on demande à mozilla la page `http://www.google.com`, mozilla demande cette page à son cache interne qui la retourne tout de suite si elle est présente ou passe par un chargeur internet sinon.

Dans ces deux exemples, il faut noter que le cache dispose d'une fonction pour *rendre* un objet au cache. Ceci permet de distinguer les objets "libérables" (chargés mais pas utilisés) de ceux qui ne le sont pas (chargés et utilisés).

3 Qu'est-ce qu'un Smart Pointer

Un *Smart Pointer* est un objet fournissant les mêmes fonctionnalités qu'un pointeur plus certaines autres. Parmi ces fonctionnalités, il y a par exemple, la gestion de ressource mémoire (désallocation automatique), la copie paresseuse, une meilleure gestion du pointeur *NULL* etc...

Afin de mieux comprendre ce qu'est un *Smart Pointer*, voici un exemple basique:

```
class Exemple
{
public:
  void f(){}
};

class NullPointerException{};

class SPExemple // also named auto_ptr
{
  Exemple *pointed_object;
public:
  SPExemple(Exemple *v=0):pointed_object(v){}
  SPExemple(SPExemple &sp){
    pointed_object = sp.pointed_object;
    sp.pointed_object = 0;
  }

  SPExemple &operator=(SPExemple &sp){
    if (this==&sp)
      return *this;
    if (pointed_object)
      delete pointed_object;
    pointed_object = sp.pointed_object;
    sp.pointed_object = 0;
    return *this;
  }

  ~SPExemple() {
    if (pointed_object)
       delete pointed_object;
  }

  Exemple &operator*(){
    if (!pointed_object)
      throw NullPointerException();
    return  *pointed_object;
```

```
  }
  Exemple *operator->() {
    if (!pointed_object)
      throw NullPointerException();
    return pointed_object;
  }
  const Exemple &operator*() const {
    if (!pointed_object)
      throw NullPointerException();
    return  *pointed_object;
  }
  Exemple *const operator->() const {
    if (!pointed_object)
      throw NullPointerException();
    return pointed_object;
  }
};



int main(void)
{
  SPExemple sp(new Exemple);
  SPExemple sp2(sp); // Transfert
  try {
    sp2->f();
    sp->f(); // Exception
  }
  catch(const NullPointerException &e) {
    //....
  }
}
```

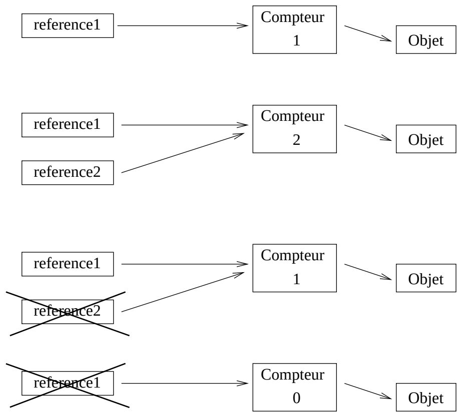
La classe SPExemple se comporte comme la classe *auto_ptr* de la librarie standard C++. Un objet SPExemple pointe sur un objet Exemple alloué dynamiquement. Lorsqu'il est détruit, il désalloue l'objet sur lequel il pointe. Lorsque'on copie un SPExemple dans un autre, l'ojet interne est transferé, ainsi il n'y a qu'un seul SPExemple par objet Exemple. Dans cet exemple, l'utilisation des *auto_ptr* permet une gestion des tentatives d'accès à un pointeur *null*.

Mais si les *Smart Pointer* sont connus, c'est surtout pour permettre la mise en place de systèmes de désallocation automatique de la mémoire (ramasse miettes). Ceci ce fait de la manière suivante:

1 A chaque objet dynamique, on attache un entier indiquant le nombre de références sur l'objet.

2 Si le nombre de références sur un objet tombe à 0, alors l'ojet est libéré.

```
void f(SmartPointer ref2)
{
...
}
...
int main(void)
{
SmartPointer ref1(new Objet());
.....
f(ref1);
.....
return 0;
}
```



Pour obtenir un tel système, il faut une classe servant de compteur de références. La classe *SmartPointer* a alors comme membre interne une référence (ou pointeur) sur le compteur de référence. De plus, la classe *SmartPointer* doit se comporter comme un pointeur, c'est à dire surcharger les opérateurs `*` et `->`.

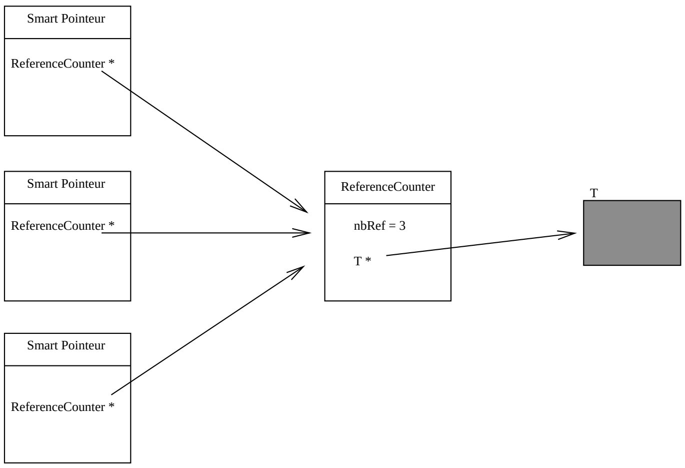


Figure 1: Relations entre les classes.

4 Ce qu'il faut faire.

Dans ce projet vous allez écrire plusieurs caches génériques. Ils auront tous la particularité de ne plus renvoyer un objet, mais un *Smart Pointer* sur un objet. L'avantage de cette démarche étant que le cache peut détruire sa référence sur l'objet le plus vieux, sans avoir à se soucier du fait que le programme l'utilise encore.

4.1 au niveau des *Smart Pointer*

Comme nous l'avons vu, l'un des intêrets des *Smart Pointer* est la possibilité d'ajouter des traitements au niveau des opérateurs `*` et `->`. Ainsi vous devez implémenter les 2 "types" de *Smart Pointer* suivant:

> *StandardSmartPointer*: Pas de traitement lors d'un accès à l'objet pointé.
>
> *NullProtectedSmartPointer*: Les accès à l'objet pointé sont protégés pour le cas où l'objet est null.

plus 1 au choix parmis:

> *MutexSmartPointer*: Les accès à l'objet pointé sont protégés par des mutex (comme un *synchronize* en java). (`man pthread_mutex_init`)
>
> *TabularSmartPointer*: L'objet pointé est un pointeur sur le début d'un tableau. Les opérateurs `+` et `[]` sont surchargés et peuvent lever une exception en cas de débordement. Attention : plutôt difficile !!!
>
> *WhatEverSmartPointer*: un *Smart Pointer* de votre cru `:)`.

4.2 au niveau des Caches

Un cache aura au minimum 3 arguments templates:

> *les objets:T* Le type des objets dans le cache est passé par template.
>
> *Le chargeur:Loader* Le type du chargeur est lui aussi passé par template. Un cache devra avoir un constructeur prenant le chargeur en argument. Ce chargeur possede une méthode `T *operator()(Key k)`, qui correspond à la demande de l'objet de clef `k` (note: peut-être devrait-elle renvoyer une exception ...).
>
> *la clef:Key* La demande d'objet au cache se fait par l'intermédiaire de clefs. Le type de la clef sera un argument template du cache mais aura comme valeur par défaut *int*.

Enfin, pour chacun des *Smart Pointer* choisis, votre *concept* de cache devra contenir une methode de demande d'objet, par exemple:

```
MutexSmartPointer<T> getMutexSmartPointer(const Key &k)
```

Un certain nombre d'autres fonctions devront être présentes, par exemple:

- La possibilité d'itérer sur les éléments chargés dans le cache.
- Un fonction permettant de savoir si un objet est dans le cache.

* ...

Vous devez implémenter 2 caches aux propriétés différentes (de même que les *vector* et les *list* de la STL ont des propriétés différentes sur le temps d'insertion par exemple). Voici quelques exemples de propriétés:

* La stratégie de libération:

  premier chargé, premier libéré: Au niveau de la libération, c'est le premier objet chargé (par le chargeur) qui est détruit en premier.

  le plus anciennement accédé, premier libéré: Au niveau de la libération, c'est le plus anciennement objet demandé qui est détruit en premier.

* *complexité:* Le temps de recherche d'un objet se fait en $O(log(n))$ si n est le nombre d'objet indexé dans le cache.

* *originalité:* L'idée est de trouver tout seul une ou plusieurs contraintes interessantes (idée: penser au cache web par exemple).

Enfin, on doit pouvoir à tout moment modifier la taille du cache. Ceci doit pouvoir se faire à l'aide de 2 fonctions, une où la limite est le nombre d'objet maximum, l'autre où la limite est la taille (en octet) du cache en mémoire.

5 Utilisation

Enfin, vous devez écrire un petit visualisateur d'image utilisant vos caches. Vous êtes complétement libre sur les outils que vous utilisez. La seule contrainte étant qu'il faut que le programme indique d'une manière ou d'une autre quels sont les fichiers actuellement dans le cache. Par exemple, un visualisateur en mode texte donnera:

```
$view_picture toto/
action: list

pictures founded in directory toto/:
------------
1: maison.jpg
2: chien.jpg
3: chat.bmp
4: PIC0009.png
------------
(*) = file is loaded in cache.

action: view chien.jpg

action: list
```

```
pictures founded in directory toto/:
------------
1: maison.jpg
2: chien.jpg      *
3: chat.bmp
4: PIC0009.png
------------
(*) = file is loaded in cache.

....
```

6 Planning et recommendations

Ce projet doit être réalisé par groupe de 4 personnes maximum, 3 minimum. Vous devez passer 2 soutenances. La première **doit** se faire **avant le 21 Mars**. Pour ce, vous devez prendre rendez vous avec moi par mail (allali@univ-mlv.fr), en me fournissant un bref rapport (4 pages max) qui devra contenir:

- Les noms des membres du groupe (pas de modifications possibles après).
- La norme de nommage choisie.
- Les noms des espaces de nommage prévus, ainsi qu'un bref aperçu des classes figurant dedans.
- Le détail de votre *concept* de cache.
- Une description de vos *Smart Pointer*.

Le but de cette pré-soutenance étant de vous aidez dans votre reflexion, donc n'oubliez pas de faire la liste de vos questions `:)`. Vous pouvez proposer d'autres utilisation de vos caches, en remplacement du visualisateur d'image, on en discutera à la soutenance.

Les sources de votre projet devront être envoyées à allali@univ-mlv.fr, sous la forme d'un seul fichier compressé (.tar.gz ou .zip ou .tar.bz2) avant le **12 Mai 2003**, cachet du serveur mail faisant fois. Ces sources seront les seules prises en compte dans la notation et pour la soutenance.

Le rapport de votre projet devra être envoyé toujours à la même adresse et sous les mêmes conditions avant le **19 Mai 2003**. Vous pouvez l'écrire dans le format de votre choix sauf *doc* (html, ps, pdf et tex sont les biens venus).

La soutenance finale aura lieu la dernière semaine de Mai, en même temps que la soutenance de *Génie Logiciel*.

Pour toute question n'hésitez pas à me contacter par mail, ou venir directement au bureau 4B070. Codez bien,

Julien Allali.

7 Annexe:

Voici quelques remarques et conseils en vrac:

- A priori, le cache contiendra un ensemble de compteurs de référence.
- *doxygen* est un outil très pratique pour faire la documentation de votre code source (un peu comme javadoc).
- La notation prendra principalement en compte le côté professionel de votre code. Ce qui implique:
  - L'utilisation des namespace.
  - L'utilisation des const.
  - L'utilisation des const références.
  - Les commentaires judicieux dans le code (c'est à dire devant chaque fonction et chaque classe).
  - L'utilisation d'un outil de compilation avancé: Makefile, Imakefile, automake etc... (vous trouverez à l'adresse `http://monge.univ-mlv.fr/~allali/Makefile` un exemple de makefile).
  - Un code final **éprouvé**: penser à vous réserver une (ou plusieurs) phase(s) de teste. Vous pouvez par ailleurs consacré une petite partie de votre rapport là dessus.
- Pour le visualisateur, vous pouvez utilisez la librairie `http://www.imagemagick.org/www/Magick++/index.html`
- Si vous voulez faire une visualisateur entierement graphique, regardez du côté de la librairie *Qt*, et de l'outil *Qt-Designer*.
- Pour l'écriture de vos concepts, inspirez vous de la *STL* (`/usr/share/doc/stl-manual/html/index.html`).
- N'oublier pas donner dans la doc, la liste des contraintes que vous posez sur vos templates.
- Pourquoi pas utiliser des Arbres Binaires de Recherche ...
- Tout *new* mérite un *delete*.
- Pour libérer volontairement un *Smart Pointer* on peut faire:

  ```
  SP<T> sp(new T);
  ......
  sp=0;
  ```

- Un forum de discussion sera mis en place pour ce projet.