TP Semaine 14

Nous allons aujourd'hui implémenter et manipuler une classe nommée `Pile<T>`, dont l'interface correspond au type abstrait Pile. Dans les premiers exercices, cette classe sera implémentée en utilisant un tableau de taille dynamique. Puis dans un second temps, nous nous intéresserons à la gestion des exceptions, nous utiliserons une Pile de taille maximale fixe pour illustrer cette notion.

Recopiez chez vous le répertoire `/net/Bibliotheque/AP2/TP_par_Semaine/Semaine14`. Vous trouverez dans ce répertoire un répertoire `PileDyn` (dans lequel vous implémenterez une classe Pile de taille variable) et un répertoire `PileStat` (dans lequel vous trouverez une classe Pile de taille fixe).

Exercice 1 : implémentation de la classe Pile (dynamique)

Commencez par vous placer dans le répertoire `PileDyn`. Vous trouverez dans ce répertoire un fichier `Makefile` ainsi qu'un fichier `main.cc`.

Jusqu'à présent, nous passions aux fonctions des paramètres donc les types étaient définis dans les prototypes de ces fonction. Le concept de template (patron en français) nous permet de nous affranchir de cette contrainte et ainsi de définir des fonctions et classes génériques.

L'avantage des classes génériques est que le même code fonctionne "pour n'importe quel type", ce qui évite de "copier coller du code" et qui simplifie évidemment sa maintenance.

L'utilisation des classes génériques est donc particulièrement adaptée pour l'implémentation des types abstraits de données.

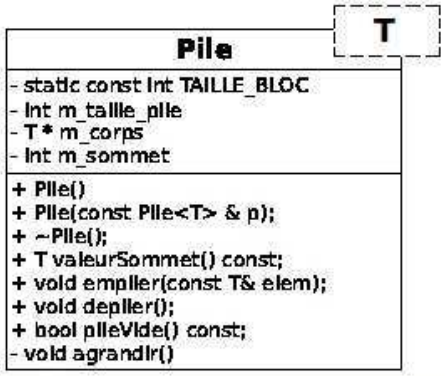
Exemple :

```
template < class T >
class MaClasseTemplate {
   public :
        MaClasseTemplate();
        ...
        void ajouterElement(const T& elem);
        ...
   private :
        ...

};
```

Dans cet exercice, nous souhaitons mettre en place une classe générique Pile n'ayant pas de taille maximale (hormis évidemment la quantité de mémoire que possède l'ordinateur).

Exercice 48 : Donner une interface de la classe générique Pile respectant le diagramme suivant :



Exercice 49 : Donner l'implémentation de la classe générique Pile dans un fichier `Pile.cxx` (ne pas oublier d'inclure ce fichier dans `Pile.h`).

Exercice 50 : Ajouter dans la classe Pile une méthode permettant de connaitre la taille de la Pile.

Exercice 51 : Ajouter dans `main.cc` un exemple d'utilisation de la classe Pile (tester des piles de plusieurs types).

Exercice 52 : Ajouter dans `main.cc` une méthode générique non-membre `template < class T > void clonerPile(Pile<T> p, Pile<T> &p_clone)` permettant de copier la pile `p` dans `p_clone` (attention l'ordre doit rester le même).

Exercice 53 : Ajouter dans `main.cc` une méthode générique non-membre `template < class T > void inverserPile(Pile<T> &p)` permettant d'inverser la pile `p`.

Exercice 54 : Ajouter dans `main.cc` une méthode non-membre `void supprimerNegatif(Pile<int> &p)` permettant de supprimer les entiers négatifs de la pile `p` sans modifier l'ordre de la pile.

0.1 Comparaison avec la classe stack de la STL

La STL propose elle aussi une implémentation des piles. Nous souhaitons maintenant comparer les performances de notre pile avec celle de la STL.

Exercice 55 : Ecrire dans `main.cc` une méthode non-membre `void testPerformancePile(Pile<int> &p)` ajoutant dans la pile p les entiers de 1 à 100000 puis supprimant tout ces entiers.

Exercice 56 : La classe correspondante à Pile dans la STL s'appelle `stack` (pile en anglais). Les *stacks* possèdent entre autres les mêmes méthodes que celles que nous avons implémentées : `push` (`empiler`), `pop` (`depiler`), `top` (`valeurSommet`) et `empty` (`pileVide`).

Ecrire dans `main.cc` une méthode non-membre
`void testPerformanceStack(stack<int> &p)` ajoutant dans la pile p les entiers de 1 à 100000 puis supprimant tout ces entiers. Comparer les performances de ces deux classes. Que pouvez vous en conclure ?

Gestion des exceptions

Placez vous maintenant dans le répertoire `PileStat`. Vous y trouverez les fichiers `Pile.h`, `Pile.cxx`, `main.cc` et `Makefile`. `Pile.h` et `Pile.cxx` sont une implémentation de la Pile de taille fixe (au maximum la pile peut contenir 20 éléments).

Lisez le fichier `Pile.cxx`, vous pourrez remarquer que nous y utilisons la fonction `assert` afin de vérifier que l'opération qui est faite sur la pile est valide. L'utilisation d'assertions convient bien pour les besoins de la mise au point : le programme s'arrête dans une situation critique où une assertion, qui devait être normalement vraie, se trouve être fausse, ce qui peut révéler une erreur de programmation. Cependant, dans un cas d'utilisation réel, ceci peut se révéler catastrophique puisque, par exemple des données non sauvegardées peuvent alors être perdues.

Les exceptions permettent de signaler des situations indésirables (par exemple le fichier dont on vient de demander le nom n'existe pas) mais qui sont récupérables pendant l'exécution (on peut signaler une erreur et revenir au menu). En c++, il faut pour gérer les exceptions utiliser les mots clefs `try`, `throw` et `catch`. Le mot clef `try` permet de définir une zone de code qui pourrait générer une erreur. Si c'est le cas, alors le mot clef `throw` permet de lever l'exception correspondante. Enfin, le mot clef `catch` permet de récupérer l'exception et de définir un bloc dans lequel cette exception sera gérée. Par exemple :

```
void traiter_fichier(char nom[]) {
   ifstream f(nom, ios::in);
    if ( f.fail() ) {
      throw string("erreur");
   }
}

void menu () {
   bool encore = true;
   while (encore) {
      try {
        // code qui pourrait g\'en\'erer une erreur
        ...
        traiter_fichier("xy");
        ...
      }
      catch (string e) {
       //traitement pour g\'erer l'exception
       cout << "oups : " << e << endl;
      }
   }
}
```

Dans cet exemple, si le fichier ne peut pas être ouvert (par exemple, s'il n'existe pas ou s'il est endommagé) alors la fonction `traiter_fichier` va lever une exception (`throw string(``erreur'')`). Cette erreur est alors récupérée dans la fonction `menu` (`catch(string e)`) et le traitement approprié est effectué. Ici, c'est une chaîne de caractères qui est utilisée mais nous pourrions utiliser n'importe quel type (par exemple, un entier `911`).

La bibliothèque standard du c++ contient bien évidemment une classe permettant de gérer ces exceptions (il faut pour l'utiliser inclure `exception`). Cette classe est définie comme suit :

```
class exception
{
public:
    //constructeur
    exception() throw(){ }

    //destructeur
    virtual ~exception() throw();

    // retourne une cha\^ine contenant l'erreur.
    virtual const char* what() const throw();
};
```

Cette classe est prévue pour pouvoir être dérivée afin d'avoir une gestion des exceptions dédiée à notre programme.

Exercice 57 : Ajouter dans le fichier `Pile.h`, une classe `ExceptionPile` héritant de la classe `exception`.

Exercice 58 : Ajouter dans la classe `ExceptionPile`, un constructeur avec paramètre `ExceptionPile(const char * raison) throw()` (ce tableau de caractères sera utilisé pour retourner l'information). Ajouter dans la classe `ExceptionPile`, un attribut `const char * m_raison` pour stocker ce tableau de caractères.

Exercice 59 : Implémenter dans la classe `ExceptionPile`, la méthode `const char* what() const throw();` qui retourne sous la forme d'un tableau de caractères la "raison" de la levée de l'exception.

Exercice 60 : Dans chaque méthode de la classe `Pile`, lever une exception en lieu et place d'une assertion. Pour cela, il suffit en cas d'erreur d'ajouter `throw ExceptionPile("Raison de l'exception")`.

Exercice 61 : Dans le fichier `main.cc`, ecrire trois fonctions `void testEmpiler()`, `void testdepiler()` et `void testValeurSommet()` permettant de générer et de gérer les exceptions respectivement liées à l'ajout d'un élément dans une pile pleine, la suppression d'un élément dans une pile vide et l'accès au sommet de la pile dans une pile vide.