Semaine 15 et 16
Implémentation des types abstraits de données

Sources liées à ce TD :
- `PileStat.ps` : à distribuer aux étudiants
- `PileDyn_extraits.ps` : à distribuer aux étudiants

Implémentation : introduction

Pour implémenter un TAD, on déterminera en fonction :
- des constructeurs disponibles
- la taille mémoire utilisée (complexité en taille ou espace)
- la facilité de manipulation (complexité en temps)

Remarque : l'implémentation des types abstraits est une étape qui précède la programmation proprement dite. C'est pourquoi il est possible de parler abstraitement d'implémentation ! Par exemple :

```
Type TPile de TInfo =  entité ( corps : tableau[TAILLE_PILE] de TInfo ;
                                sommet : -1..TAILLE_PILE-1 )
```

Ceci permet de s'affranchir de la syntaxe, de garder encore une certaine généralité, etc... Cependant, cette approche, héritière du paradigme impératif, ne permet pas l'encapsulation des données et des méthodes : le type TPile ainsi défini n'interdit absolument pas des actions comme `x <-- P.corps[5] ; P.corps[sommet-3] <-- 12` etc...

Nous verrons donc l'implémentation des types abstraits de données comme leur traduction concrète en terme de classe.

1 Implémentation des Piles : allocation contiguë

On parle d'allocation contiguë lorsque chaque composante de la structure est placée physiquement (en mémoire) à côté de sa voisine logique ; autrement dit, le constructeur utilisé est le tableau. Les piles ne demandant pas d'insertion / suppression "au milieu", une allocation contiguë est bien adaptée.

1.1 Allocation statique

Une implémentation très simple mais finalement à peu près satisfaisante, à l'aide d'un tableau dont la taille est définie dès la compilation ; un entier (indice) permet de repérer le sommet.

Exercice 43 : Déclaration de la classe, la définition du constructeur et de empiler.

Guider / rafraichir la mémoire des étudiants, puis distribuer `PileStat.ps` et commenter.

Remarque : l'avantage de cette implémentation est sa simplicité d'écriture et son efficacité ; l'inconvénient est la limitation imposée à la taille de la pile.

1.2 Allocation dynamique

Cet inconvénient peut être supprimé si l'on procède par allocation dynamique du tableau ; le tableau n'est pas créé à la compilation, mais lorsqu'une variable Pile<T> est déclarée ; si, au cours de l'utilisation, la pile est saturée, on alloue un bloc plus grand, on y recopie l'ancienne pile, puis on libère l'ancien bloc et on fait pointer le tableau (pointeur) sur la nouvelle zone. ; de plus, cette solution permet de respecter la convention selon laquelle les objets sont déclarés dans .h et définis dans .cxx.

Exercice 44 : Ecrire la déclaration de cette classe.

```
template <class T>
class Pile<T>{
public :
 // idem sauf pas pilePleine()
private :
  static const int TAILLE_BLOC; //unité d'allocation
  int m_taille_pile;
  T * m_corps; // le tableau sera alloué dynamiquement
  int m_sommet;
  void agrandir(); // sera utilisé par empiler s'il le faut
};
```

Expliquer la stratégie d'allocation : on choisit une unité d'allocation (bloc), chaque fois qu'il faut agrandir on augmente d'un bloc.

Exercice 45 : Expliquer ce qui change dans les définitions (Pile.cxx). Ne pas détailler toutes les méthodes. On peut regarder la constante entière de classe `TAILLE_BLOC`, le constructeur, le destructeur, et empiler.

```
//ici on peut définir la constante dans .cxx
template <class T>
const int Pile<T>::TAILLE_BLOC = 10; //par exemple

template <class T>
Pile<T>::Pile()
{
  m_taille_pile = TAILLE_BLOC; // taille par défaut : 1 bloc
  m_corps = new T[m_taille_pile];
  m_sommet = -1;
}
template <class T>
Pile<T>::Pile(const Pile<T> & p)
```

```
{
  m_corps = new T[p.m_taille_pile]; // chaque pile a sa taille
  m_sommet = p.m_sommet;
  for(int i = 0; i <= m_sommet; i++)
    m_corps[i] = p.m_corps[i];
}
template <class T>
Pile<T>::~Pile()
{
  delete[] m_corps;
}
template <class T>
void Pile<T>::empiler( const T& elem)
{
  if(m_sommet == m_taille_pile-1) // pile pleine
    agrandir();
//empilement proprement dit.
  m_sommet++;
  m_corps[m_sommet] = elem;
}
```

On utilise une fonction agrandir : lorsque la pile est pleine, si l'on veut empiler, on agrandit l'espace : allocation d'un nouvel espace plus grand dans lequel on recopie l'ancien tableau.

Guider les étudiants avec un schéma, puis distribuer `PileDyn_extraits.ps`.

```
template <class T>
void Pile<T>::agrandir()
{
  m_taille_pile += TAILLE_BLOC ;
//allocation nouvelle zone
  T* tmp = new T[m_taille_pile];
//recopie de la pile dans cette zone
  for(int i=0; i<= m_sommet; i++)
    tmp[i] = m_corps[i];
//libération ancienne zone
  delete[] m_corps;
// nouvelle zone devient le corps de la pile
  m_corps = tmp;
}
```

La difficulté est sans doute ici de bien choisir la taille du bloc d'allocation en fonction de l'application et de l'environnement ; si le bloc est trop gros, il n'y a jamais de réallocation, autant valait un tableau statique ! si le bloc est trop petit, il y a sans cesse recopie, libération, allocation, d'où coût élevé.

2 Implémentation des Listes

On peut mentionner trois implémentations possibles : seule la troisième sera traitée en TD, les deux premières seront juste évoquées pour justifier la troisième.

2.1 Allocation contiguë : tableau de “taille variable”

Très simple, mais ... Quoi ? Faire réfléchir les étudiants...

Mais les insertions/suppressions en "milieu" de liste sont très coûteux. En revanche, la limitation à un parcours séquentiel paraît très artificielle.

Conclusion : convient assez mal.

2.2 Allocation contiguë à cellules non contiguës

Deux tableaux, l'un pour les éléments de la liste, l'autre pour les indices (adresses) ; plus proprement, un tableau d'entités, avec un champ info, et un champ suivant ; aujourd'hui cette implémentation ne se justifie plus beaucoup ; intéressant si on ne dispose ni de listes ni de pointeurs. A la rigueur, simplicité du mécanisme d'allocation, possibilité, pour des listes assez courtes, de manipuler des indices de faible taille (1 ou 2 octets) (? ? ?)... On peut regarder le principe sur un schéma

On aura par exemple un bloc de 10 cellules (entités) ; soit la liste d'entiers (5, 11, 3, 8, 17), elle pourrait être représentée ainsi :

info	17	3			11		5	8		
suivant	-1	7			1		4	0		
indices	0	1	2	3	4	5	6	7	8	9

Il faut connaître l'indice du début, donc une donnée appelée premier (ici = 6) ; une valeur impossible (-1 convient bien) signifie NULL. Evidemment, l'aspect un peu aléatoire du chaînage résulte de l'histoire de la liste. Enfin, pour faire une insertion, il faut prendre la première cellule disponible (on verra après ce que cela signifie), et la chaîner correctement ; mais comment savoir qu'une cellule est disponible ? on pourrait imaginer une autre valeur interdite pour suivant (par exemple const int LIBRE = -2) ; mais, premièrement, il n'est pas très logique de dire qu'une cellule libre est identifiée sur son champ suivant ; surtout, cela oblige à parcourir le tableau jusqu'à trouver une cellule libre. Une solution est donc de chaîner aussi les cellules libres, et d'avoir un point d'entrée nommé par exemple premierLibre. On peut avoir par exemple :

info	17	3			11		5	8		
suivant	-1	7	-1	2	1	8	4	0	9	3
indices	0	1	2	3	4	5	6	7	8	9

avec premier = 6 et premierLibre = 5

Insérons en tête la valeur 19 :

info	17	3			11	19	5	8		
suivant	-1	7	-1	2	1	6	4	0	9	3
indices	0	1	2	3	4	5	6	7	8	9

premier = 5 et premierLibre = 8

Il suffit de prélever la première cellule libre en tête de la liste des cellules libres, et de modifier le chaînage et les points d'entrée.

On aurait donc dans Liste.h (voir les sources complètes pour plus de précisions) :

```
// Liste.h
#ifndef _LISTEH_
#define _LISTEH_

template <class T>
struct Cellule
{
  T info;
  int suivant;
};

template <class T>
class Liste {
public:
  typedef int TAdresse;
  static const TAdresse NUL = -1;
  // + ....
  // toutes les méthodes de Liste
private:
  static const int TAILLE_BLOC_BASE;
  int m_tailleBloc;
  Cellule<T> * m_bloc;
  TAdresse m_premier, m_premierLibre;
  void dupliquer( const Liste<T> & liste );
  void agrandir();
};

#include "Liste.cxx"
#endif
```

2.3 Allocation non contiguë

C'est l'implémentation la plus importante ; c'est la forme classique de la liste chaînée ; très souple, la liste n'occupe que l'espace réellement nécessaire à l'instant t.

Expliquer sur des schémas et exemples la notion de `struct Cell` avec élément et pointeur sur suivant. (remarque : pas vu le struct, que class, leur dire ce que c'est)

Dans la suite, essayer de n'écrire que les fonctions sans la mention `//à faire`, car ces fonctions sont à faire pendant le TP.

On aura les déclarations suivantes :

```
// Liste.h
```

```
#ifndef _LISTE_H
#define _LISTE_H

template <class T>
struct Cellule
{
  T info;
  Cellule * suivant;
};


template <class T>
class Liste {
public:
  typedef Cellule<T>* TAdresse;

  Liste();
  ~Liste();

  TAdresse adressePremier() const;
  TAdresse adresseSuivant( TAdresse adr ) const;

  T valeurElement( TAdresse adr ) const;
  void modifierValeur( const T& elem, TAdresse adr );
  void insererEnTete( const T& elem );
  void insererApres( const T& elem, TAdresse adr );
  void supprimerEnTete();
  void supprimerApres( TAdresse adr );

private:
  TAdresse m_premier;
};

#include "Liste.cxx"

#endif
```

Et dans le fichier .cxx :

```
// Liste.cxx

#include <iostream>
#include <cassert>
#include "Liste.h"

using namespace std;
```

```
template <class T>
Liste<T>::Liste()
{
  m_premier = NULL;
}

template <class T>
Liste<T>::~Liste()
{
   while ( m_premier != NULL )
     supprimerEnTete();
}

template <class T>
typename Liste<T>::TAdresse
Liste<T>::adressePremier() const
{
  // à faire
  return m_premier;
}

template <class T>
typename Liste<T>::TAdresse
Liste<T>::adresseSuivant( TAdresse adr ) const
{
  // à faire
  assert(adr != NULL);
  return adr->suivant;
}

template <class T>
T
Liste<T>::valeurElement( TAdresse adr ) const
{
  // à faire
  assert(adr != NULL);
  return adr->info;
}

template <class T>
void
Liste<T>::modifierValeur( const T& elem, TAdresse adr )
{
  // à faire
  assert(adr != NULL);
  adr->info = elem;
}
```

```
template <class T>
void
Liste<T>::insererEnTete( const T& elem )
{
  TAdresse aux = new Cellule<T>;
  aux->info = elem;
  aux->suivant = m_premier;
  m_premier = aux;
}

template <class T>
void
Liste<T>::supprimerEnTete()
{
  assert ( m_premier != NULL );
  TAdresse aux = m_premier;
  m_premier = m_premier->suivant;
  delete aux;
}

template <class T>
void
Liste<T>::insererApres( const T& elem, TAdresse adr )
{
  // à faire
  assert(adr != NULL);
  TAdresse aux = new Cellule<T>;
  aux->info = elem;
  aux->suivant = adr->suivant;
  adr->suivant = aux;

}

template <class T>
void
Liste<T>::supprimerApres( TAdresse adr )
{
  // à faire
  assert(adr != NULL);
  TAdresse aux = adr->suivant;
  adr->suivant = aux->suivant;
  delete aux;
}
```

3 Si du temps... Implémentation des Files

Remarque : codes pas vérifiés ni mis à jour dans cette partie, le faire à la volée si besoin...

On serait tenté de transposer directement aux files ce qu'on vient de voir pour les piles. Il peut sembler en effet suffisant de remplacer l'unique sommet par premier et dernier. Cependant imaginons une file de 5 éléments. Au début premier et dernier valent -1 ; on enfile 5 fois : m_corps[++der] = elem ;
on défile 4 fois : prem++ ; on ne peut plus enfiler sans réallouer alors qu'il reste 4 places !

La solution va consister en l'utilisation d'un tableau "circulaire" en ce sens que les calculs d'indice se font modulo la taille du tableau.
Reprenons l'exemple ci-dessus on peut
enfiler en 0 : m_corps[(++der)défiler : (prem++)

Mais un nouveau problème se pose : comment distinguer entre file pleine et file vide ? Il suffit pour cela de "gaspiller" une composante du tableau que l'on nommera vide, et qui ne devra jamais rien contenir ; à la place de dernier, on utilisera cet indice nommé "vide", désignant la case suivant immédiatement le dernier.

Exercice 46 : Écrire l'en-tête, le constructeur, le destructeur...

```
//File.h
#ifndef _FILE_
#define _FILE_
template <class T>
class File{
public:
  File();
  File(const File&);
  ~File();
  bool fileVide() const;
  void enfiler(int);
  void defiler();
  const T& valeurPremier() const;
private:
  static const int TAILLE_BLOC;
  int m_taille_file;
  int m_premier, m_vide;
  T* m_corps;
  void agrandir() ;
};
#endif

//File.cxx
#include <iostream>
#include <cassert>
#include "File.h"

template <class T>
```

```
const int File<T>::TAILLE_BLOC=10;

template <class T>
File<T>::File(){
  m_taille_file=TAILLE_BLOC;
  m_premier=m_vide=0;
  m_corps=new T[m_taille_file];
}

template <class T>
File<T>::~File(){
  delete []  m_corps;
}
```

Exercice 47 : Ecrire le constructeur de copie.

```
template <class T>
File<T>::File(const File<T>& f){
  m_corps = new T[f.m_taille_file];
  m_taille_file=f.m_taille_file;
  m_premier=f.m_premier;
  m_vide=f.m_vide;
  for(int i=0;i<m_taille_file;i++)
    m_corps[i]=f.m_corps[i];
}

template <class T>
bool File::fileVide() const{
  return m_premier==m_vide;
}
```

Exercice 48 : Essayer de cerner les problèmes posés par enfiler.

Le 1° problème, c'est qu'on procède par allocation contiguë mais dynamique ; il faut donc comme pour les piles créer un nouvel espace si la file est pleine ; d'où la fonction membre privée agrandir. Le 2° problème, c'est que l'utilisation circulaire du tableau est perturbée par le changement de taille.

```
template <class T>
void File<T>::enfiler(const T& elem){
  if(m_premier == ((m_vide+1)%m_taille_file)){
    // file pleine
    agrandir() ;
//mise en file
  m_corps[m_vide]=elem;
  m_vide=(m_vide+1)%m_taille_file;
```

```
}

template <class T>
void File<T>::agrandir(){
//allocation nouveau bloc
  T * tmp=new T[m_taille_file+TAILLE_BLOC];

    //recopie ancienne file dans nouvel espace
    //en "décircularisant" pour éviter incohérences
  for(int i = 0, j=m_premier; i<m_taille_file-1; i++,j=(j+1)%m_taille_file)
    tmp[i]=m_corps[j];

  delete[] m_corps;//libération ancienne file

//mise à jour nouvelle file
  m_premier=0;
  m_vide=m_taille_file-1;
  m_taille_file+=TAILLE_BLOC;
  m_corps=tmp;
}

template <class T>
void File<T>::defiler(){
  assert(m_premier!=m_vide);
  m_premier=(m_premier+1)%m_taille_file;
}

template <class T>
const T& File<T>::valeurPremier() const{
  assert(m_premier!=m_vide);
  return m_corps[m_premier];
}
```

4 Si encore du temps... Proposer quelques exercices de révisions