TD de Programmation Orientée Objets

Corrigé du TD 3 : Classe pile d'entiers

M. Bosc, J-M. Dischler et J. Lamy

Utilisation d'un tableau statique

Ecrire une classe "pileent" permettant d'implémenter la structure de données "pile d'entiers" en utilisant un tableau statique, limité à une taille constante (de 100 par exemple). On proposera les méthodes suivantes :

```
void empiler(int x);
int depiler();
int sommet();
int hauteur();
bool estVide();
bool estPleine();
int tailleMax();
void afficher();
```

Quel résultat est fourni par le programme suivant :

```
#include "pile_ent.h"

int main()
{

pile_ent pi;

pi.empiler(2);
pi.empiler(6);
pi.empiler(1);
pi.afficher();
pi.depiler();
cout<<"sommet actuelle :"<<pi.sommet()<<endl;
return 0;
}
```

Correction :

```
// pile_ent.h: interface for the pile_ent class.
//
//////////////////////////////////////////////////////////////////////

#ifndef PILE_ENT_H
#define PILE_ENT_H

#include <iostream.h>
```

```
const int PILE_MAX=100;

class pile_ent
{
int valeurs[PILE_MAX];
int n_elem;

public:
pile_ent() { n_elem=0; }
void empiler(int x);
int depiler();
int sommet();
int hauteur() { return n_elem; }
bool estVide() { return n_elem==0; }
bool estPleine() { return n_elem==PILE_MAX; }
int tailleMax() { return PILE_MAX; }
void afficher();

};

#endif

// pile_ent.cpp: implementation of the pile_ent class.
//
//////////////////////////////////////////////////////////////////////

#include "pile_ent.h"


//////////////////////////////////////////////////////////////////////
// Modificateurs
//////////////////////////////////////////////////////////////////////

void pile_ent::empiler(int x)
{
if (estPleine()) { cout<<"impossible d'empiler"<<endl; }
else
{
valeurs[n_elem++]=x;
}
}

int pile_ent::depiler()
{
if (estVide()) {  cout<<"impossible de depiler"<<endl; return -1; }
else return valeurs[--n_elem];
}

//////////////////////////////////////////////////////////////////////
// Observateurs (selecteurs, convertisseurs, etc.)
//////////////////////////////////////////////////////////////////////

int pile_ent::sommet()
{
if (n_elem==0) { cout<<"pas de sommet!"<<endl; return -1; }
else return valeurs[n_elem-1];
```

```
}

////////////////////////////////////////////////////////////////////
// entree/sortie
////////////////////////////////////////////////////////////////////

void pile_ent::afficher()
{
cout<<"pile d'entiers, taille "<<hauteur()<<endl;
for (int i=0; i<n_elem; i++) cout<<valeurs[i]<<endl;
}
```

Utilisation d'un tableau dynamique

Réécrire la classe "pileent" en utilisant cette fois ci un tableau alloué de manière dynamique et dont la taille est fournie en argument du constructeur. On proposera les mêmes méthodes.

Correction :

```
// pile_ent.h: interface for the pile_ent class.
//
////////////////////////////////////////////////////////////////////

#ifndef PILE_ENT_H
#define PILE_ENT_H

#include <iostream.h>


class pile_ent
{
int *valeurs;
int n_elem;
int max_elem;

public:
pile_ent(int max);
~pile_ent();
void empiler(int x);
int depiler();
int sommet();
int hauteur() { return n_elem; }
bool estVide() { return n_elem==0; }
bool estPleine() { return n_elem==max_elem; }
int tailleMax() { return max_elem; }
void afficher();
};

#endif


// pile_ent.cpp: implementation of the pile_ent class.
//
////////////////////////////////////////////////////////////////////

#include "pile_ent.h"
```

```
////////////////////////////////////////////////////////////////////////
// Constructeurs et destructeur
////////////////////////////////////////////////////////////////////////

pile_ent::pile_ent(int max)
{
cout<<"dans le constructeur de base"<<endl;
valeurs = new int [max_elem=max];
n_elem=0;
}

pile_ent::~pile_ent()
{
cout<<"dans le destructeur de base"<<endl;
delete valeurs;
}

////////////////////////////////////////////////////////////////////////
// Modificateurs
////////////////////////////////////////////////////////////////////////

void pile_ent::empiler(int x)
{
if (estPleine()) { cout<<"impossible d'empiler"<<endl; }
else
{
valeurs[n_elem++]=x;
}
}

int pile_ent::depiler()
{
if (estVide()) {  cout<<"impossible de depiler"<<endl; return -1; }
else return valeurs[--n_elem];
}

////////////////////////////////////////////////////////////////////////
// Observateurs (selecteurs, convertisseurs, etc.)
////////////////////////////////////////////////////////////////////////

int pile_ent::sommet()
{
if (n_elem==0) { cout<<"pas de sommet!"<<endl; return -1; }
else return valeurs[n_elem-1];
}

bool pile_ent::egal(const pile_ent pi)
{
if (n_elem!=pi.n_elem) return false;
for (int i=0; i<n_elem; i++) if (valeurs[i]!=pi.valeurs[i]) return false;
return true;
}

////////////////////////////////////////////////////////////////////////
// entree/sortie
////////////////////////////////////////////////////////////////////////
```

```
void pile_ent::afficher()
{
cout<<"pile d'entiers, taille "<<hauteur()<<endl;
for (int i=0; i<n_elem; i++) cout<<valeurs[i]<<endl;
}
```

Ajouter à présent une méthode permettant de comparer deux piles. Le profil est le suivant :

```
bool egal(const pile_ent);
```

Correction :

```
bool pile_ent::egal(const pile_ent pi)
{
if (n_elem!=pi.n_elem) return false;
for (int i=0; i<n_elem; i++) if (valeurs[i]!=pi.valeurs[i]) return false;
return true;
}
```

Que se passe-t-il si l'on exécute le programme suivant :

```
#include "pile_ent.h"

int main()
{

pile_ent pi(20), pp(10);

pi.empiler(2);
pi.empiler(6);
pi.empiler(1);
pi.afficher();
pi.depiler();
cout<<"sommet actuelle :"<<pi.sommet()<<endl;
pp.empiler(2);
pp.empiler(6);
if (pp.egal(pi))
{
cout<<"les piles sont egales"<<endl;
}
else
{
cout<<"les piles ne sont pas egales"<<endl;
}
return 0;
}
```

Proposer une solution.
Correction : il faut ajouter un constructeur de recopie à la classe.

```
pile_ent::pile_ent(const pile_ent &pi)
{
cout<<"dans le constructeur de recopie"<<endl;
valeurs = new int [max_elem=pi.max_elem];
n_elem=pi.n_elem;
for (int i=0; i<n_elem; i++) valeurs[i]=pi.valeurs[i];
}
```