```
// Pile.h
#ifndef _PILE_H
#define _PILE_H

template <class T>
class Pile {
public:
  Pile(); // constructeur
  Pile(const Pile<T> & p); // constr. de copie
  ~Pile(); // destructeur

  // les primitives classiques :
  T valeurSommet() const;
  void empiler(const T& elem);
  void depiler();
  bool pileVide() const;
  // une primitive "spéciale" à cause de cette implémentation
  // pas obligatoire, mais bon
  bool pilePleine() const;
private:
  static const int TAILLE_PILE = 20;
  T m_corps [TAILLE_PILE];
  int m_sommet;
};

#include "Pile.cxx"

#endif
```



```
// Pile.cxx

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

template <class T>
Pile<T>::Pile()
{
  m_sommet = -1; // pile vide : l'indice est un "non indice"
}

template <class T>
Pile<T>::Pile<T>(const Pile<T> &p)
{
  m_sommet=p.m_sommet;
  for(int i=0; i<=p.m_sommet; i++)
    m_corps[i] = p.m_corps[i];
}

template <class T>
Pile<T>::~Pile<T>()
{}  //peu à faire...!!!

template <class T>
bool
Pile<T>::pileVide() const
{
  return (m_sommet == -1);
}

template <class T>
T
Pile<T>::valeurSommet() const
{
  assert(m_sommet != -1);
  return (m_corps[m_sommet]);
}
template <class T>
void
Pile<T>::empiler(const T& elem)
{
  assert(m_sommet != TAILLE_PILE-1);
  m_sommet++;
  m_corps[m_sommet] = elem;
}

template <class T>
void
Pile<T>::depiler()
{
  assert(m_sommet != -1);
  m_sommet--;
}

template <class T>
bool
Pile<T>::pilePleine() const
{
  return (m_sommet == TAILLE_PILE-1);
}
```