TD 7 - Templates

Programmation Orientée Objet

Objectif

- Comprendre le principe des templates C++

1 Modèle de fonction 1

Créez un modèle de fonction permettant de calculer la somme d'un tableau d'éléments de type quelconque, le nombre d'éléments du tableau étant fourni en paramètre. Ecrivez un programme utilisant ce patron.

```
#include<iostream>
using namespace std;

////////////////////////////////////////////////////////////////////////////////
// Remarque: le template somme n a de sens que si :                           //
// - l operation d addition a un sens; il ne peut donc pas s agir d'un type de pointeur //
// il peut s'agir d'une classe Ã  condition que cetet derniere ait une operation //
// surchargee d'addition.                                                     //
// - La declaration T est correcte; cela signifie que si T est un type clase, il est //
// necessaire qu'il dispose d'un constructeur sans argument                   //
// - L affectation som=0 est correcte; cela signifie que si T est un type classe, il //
// est nÃ©cessaire qu'il ait surdefini l'affectation.                          //
////////////////////////////////////////////////////////////////////////////////
template <class T> T somme (T *tab, int iNbElt){
        T som;
        som=0;
        for (int i=0; i < iNbElt; i++)
                som = som + tab[i];
        return som;
}

int main(){
        int ti[] = {3, 5, 2, 1};
        float tf[] = {2.5, 3.2, 1.8};

        cout << "La somme de ti est egale a " << somme (ti,4) << endl;
        cout << "La somme de tf est egale a " << somme (tf,3) << endl;

        return (EXIT_SUCCESS);
}
```

2 Modèle de fonction 2

Créez un modèle de fonction permettant de calculer le carré d'une valeur de type quelconque (le résultat possédera le même type. Ecrivez un programme utilisant ce patron.

```
#include<iostream>
using namespace std;

template <class T> T carre (T a){
        return a * a ;
}

int main(){
        int n = 5;
        float x = 1.5;

        cout << "Le carre de " << n << " est egal a " << carre (n) << endl;
        cout << "Le carre de " << x << " est egal a " << carre (x) << endl;
```

```
        return (EXIT_SUCCESS);
}
```

3 Classe de fonction

Créez une classe ensemble générique. Vous pouvez pour cela vous appuyer sur le TD précédent traitant des ensembles.

```
#include<iostream>
using namespace std;

/*************************************************/
template <class T>
class Ensemble{
        T * tab;
        int taille;
        int appartient(T);
  public :
        Ensemble();
        int card();
        void operator<<(T);
        void suppr(T);
        Ensemble<T> operator+(Ensemble<T>);
        Ensemble<T> inter(Ensemble<T>);
        Ensemble<T> operator[] (int offset);

////////////////////////////////////////////////////////////////////////////////////////////////////
// REMARQUE
////////////////////////////////////////////////////////////////////////////////////////////////////
// En fait, la declaration de fonctions amies est vicieuse avec les templates.
// Lorsqu'on instancie une classe avec un parametre int par exemple, celle-ci sera amie de operator<< avec
// comme parametre un ensemble<int>, et seulement lui.
// Pour resumer, il faudrait ecrire une surcharge de operator<< pour chaque type que l'on utilise pour
// instancier ensemble, pour que ca compile.
//
// La solution est de declarer la fonction template operator<< en tant qu'amie (et non sa surcharge
// pour le type T seulement), ainsi le code compilera.
//
        template <class U> friend ostream& operator<< (ostream& os, const Ensemble<U>& s);
////////////////////////////////////////////////////////////////////////////////////////////////////
};
/*************************************************/

/*************************************************/
// ostream& operator<<
/*************************************************/
// Solution d affichage avec surcharge de l operateur
// operator<<
/*************************************************/
template <class T>
ostream& operator<<(ostream &os, const Ensemble<T> & e){
  for (int i=0 ; i<e.taille ; i++)
    os << e.tab[i] << " ";
  return os;
}

/*************************************************/
// operator[]
/*************************************************/
template <class T>
Ensemble<T> Ensemble<T>::operator[] (int offset){
        return tab[offset];
}
/*************************************************/
// Ensemble()
/*************************************************/
template <class T>
Ensemble<T>::Ensemble(){
  taille=0;
  tab=new T;
}
/*************************************************/
// card
/*************************************************/
template <class T>
int Ensemble<T>::card(){
  return taille;
```

```
}
/*************************************************/
// appartient
/*************************************************/
template <class T>
int Ensemble<T>::appartient(T el){
  int i=0;
  while (tab[i]!=el && i<card())
    i++;
  return (i<taille);
}
/*************************************************/
// void operator<<
/*************************************************/
template <class T>
void Ensemble<T>::operator<<(T el){
    if (!appartient(el)){
      T*nouv;
      nouv=new T[taille+1];
      for (int i=0 ; i<card() ; i++)
        nouv[i]=tab[i];
      taille++;
      nouv[taille-1]=el;
      delete[] tab;
      tab=nouv;
    }
}

/*************************************************/
// suppr
/*************************************************/
template <class T>
void Ensemble<T>::suppr(T el){
  int i=0;
  while (tab[i]!=el && i<taille)
    i++;
  if (i!=card()){
    taille--;
    tab[i]=tab[card()];
  }
}
/*************************************************/
template <class T>
Ensemble<T> Ensemble<T>::operator+(Ensemble<T> e2){
  Ensemble<T> e=*this;
  for (int i=0 ; i<e2.card() ; i++)
    e << e2.tab[i];
  return e;
}
/*************************************************/
// inter
/*************************************************/
template <class T>
Ensemble<T> Ensemble<T>::inter(Ensemble<T> e2){
  Ensemble<T> e3;
  int j=0;
  for(int i=0 ; i<card() ; i++)
    if (e2.appartient(tab[i])){
      e3.tab[j]=tab[i];
      j++;
        }
  e3.taille=j;
  return e3;
}


/*************************************************/
// main
/*************************************************/
int main(){
        Ensemble<int> ens1;
        int a=1;
        int el;

        system("clear");

        /* Creation d un premier ensemble */

        cout << "CREATION ENSEMBLE 1" << endl;
        cout << "-------------------" << endl<<endl;
        while(a){
```

```
        cout << "0 fin ; 1 affiche ; 2 ajoute ; 3 supprime" << endl;
        cin >> a;
        if (a==1){
          cout << ens1;
        } else if (a==2){
          cin >> el;
          ens1 << el;}
        else if (a==3){
          cin >> el;
          ens1.suppr(el);}
        cout << endl;
      }

      /* Creation d un second ensemble */
      cout << endl << "CREATION ENSEMBLE 2" << endl;
      cout << "--------------------" << endl;
      Ensemble<int> ens2;
      a=1;
      while(a){
        cout << "0 fin ; 1 affiche ; 2 ajoute ; 3 supprime" << endl << endl;
        cin >> a;
        if (a==1)
          cout << ens2;
        else if (a==2){
          cin >> el;
          ens2 << el;}
        else if (a==3){
          cin >> el;
          ens2.suppr(el);}
        cout << endl;
      }

      /* Test intersection des 2 ensembles */
      cout << endl;
      cout << "INTERSECTION DES 2 ENSEMBLES" << endl;
      cout << "----------------------------" << endl;

      cout << (ens1.inter(ens2));
      cout << endl;

      /* Test union des 2 ensembles */
      cout << endl;
      cout << "UNION DES 2 ENSEMBLES" << endl;
      cout << "---------------------" << endl;

      cout << (ens1+ens2);
      cout << endl;

      cout << "Bye bye" << endl;
      return EXIT_SUCCESS;
}
```