MODELES.

MODELES DE FONCTION.

introduction.

Lorsqu'on désire écrire la définition d'une fonction *max* qui a pour valeur le maximum de deux valeurs, on est amené à écrire le même code pour différents types d'arguments :

```
int max ( int a, int b){ return a>b?a:b; }
float max ( float a, float b) {return a>b?a:b;}
duree max (duree a, duree b){return a>b?a:b;}
...
```

C++ offre la possibilité de définir des modèles de fonction qui évitent la réécriture des diverses fonctions.

syntaxe.

Une définition de modèle de fonction se fait de la façon suivante :
template < *liste de paramètres formels séparés par des* **,** *> définition de la fonction.*

Un paramètre formel a la syntaxe suivante :

- **class** *nom de classe* pour un modèle paramétré par une classe
- **nom de classe** *nom de variable* pour un modèle paramétré par une variable

exemples

Pour la fonction *max* définie précédemment nous obtenons la définition de modèle suivante :

```
template <class T> T max ( T a, T b) { return a>b?a:b;}
```

L'utilisation du modèle de fonction *max* se fait, par exemple, de la façon suivante :

```
void main ()
{  int a, b;
             .....
             cout<< max(a,b);
}
```

Lors de l'appel *max(a, b)*, on dit qu'il y a i**nstanciation** du modèle *T max (T , T)* en la fonction *int max (int, int).*

Nous pourrions également définir un modèle de fonction qui accepte des tableaux à nombre et type variables d'éléments en paramètres :

```
template <class T, int n> void f( T t[n], int nb)
{  ...
}
```

L'appel de la fonction pouvant se faire de la façon suivante :

```
float t1[10]; int t2[20];
...
f(t1, 10);
f(t2, 20);
```

Surcharge de modèles de fonctions.

1. Les modèles de fonctions peuvent être surchargées exactement comme les fonctions ordinaires : supposons qu'on veuille écrire une définition de fonction *max* qui calcule la valeur maximum parmi les éléments d'un tableau. Nous écririons la surcharge de *max* de la façon suivante :

```
template <class T> T max ( T a [], int taille)
{      T min; int i;
       min = a[0];
       for (i=1; i<taille; ++i) if(a[i]<min)min=a[i];//hypothèse : < existe
                                                     // pour la classe T
       return min;
}
```

2. Si on utilise la fonction *max* définie précédemment avec des arguments de type *char ** , la valeur maximum retournée par la fonction sera la valeur de la chaîne de caractères dont l'adresse est la plus grande : en effet ce qui est comparé, ce ne sont pas les chaînes mais les valeurs des pointeurs sur les chaînes. Pour résoudre ce problème nous sommes donc amenés à surcharger la définition de la fonction *max* pour les chaînes de caractères :

```
char * max ( char *a, char *b){ return strcmp(a,b)<0?b:a;}
```

un problème.

Supposons que l'on veuille écrire un modèle de fonction *somme* qui a pour valeur la somme des deux arguments qui sont d'un type arithmétique quelconque :

```
template <class T> T somme (T a, T b){ return a+b;}
```

Cette définition permettra de faire des sommes avec des arguments de même type, mais pas avec des arguments de types différents :

```
int a=1, b=2;
unsigned int c=1, d=2;
float x=1.1, y=2.2;
```

Nous avons alors :

```
somme(a, b)  //     O.K.
somme(a,c)   //     le compilateur répond par le message :
somme(a,x)   //     could not find a match for
somme(c,x)   //     somme(     ,     )
```

Nous avons alors envie d'écrire :

```
template<class T, class U>  X  somme(Ta, U b){ return a+b;}
```

Mais se pose alors la question du type ***X*** de la fonction : deux choix possibles :

1. Choisir le plus grand possible (*double*).

2. Laisser l'utilisateur choisir le type de retour :

```
template <class T, class U, class RT>
             RT somme (T a, U b, RT bidon){return a+b;}
```

le dernier paramètre n'est là que pour indiquer le type du résultat, qui est donc choisi par l'utilisateur du modèle de fonction. Nous avons :

```
int a=-10; unsigned c=1; float x;
somme(a, x, 0.0)    // O.K. résultat flottant.
somme(a, c, 0u)     // a pour valeur 65527
somme(a, c, 0)      // a pour valeur -9
```

exercice.

Ecrire un modèle de fonction qui permet de trier un tableau à une dimension, dans l'ordre croissant. On fera l'hypothèse que la classe *T*, type des éléments du tableau possède les opérateurs d'affectation = et de comparaison >.

MODELES DE CLASSE.

Les ***templates*** (***modèles***, ***patrons***) ou encore ***types paramétrés*** sont utilisés pour implanter des structures de données qui sont indépendantes du type des données qu'elles stockent Par exemple, lorsqu'on utilise une structure de tableau pour stocker un ensemble d'éléments, le type des éléments n'a aucun sens par rapport à la définition des actions de stockage, suppression ou recherche. En revanche il est nécessaire que les opérations de construction et destruction, d'affectation, d'égalité et non égalité, voire de comparaison soient définies pour le type des éléments stockés. Pour ne pas avoir à réécrire la définition de toutes ces fonctions pour chacun des types qui peut être stocké dans le tableau, le langage C++ offre la possibilité de déclarer une classe paramétrée par un ou plusieurs types.

Les templates ne peuvent pas être compilés séparément : le compilateur a besoin de connaître le ou les types paramètres pour générer le code objet correspondant. On parle d'*instanciation* du template.

syntaxe.

Une définition de modèle de classe se fait de la façon suivante :

template < *liste de paramètres formels séparés par des* **,** *> définition de la classe.*

Un paramètre formel a la syntaxe suivante :

- **class** *nom de classe* pour un modèle paramétré par une classe
- **nom de classe** *nom de variable* pour un modèle paramétré par une variable

la classe tableau.

Nous désirons définir une classe *Tableau* qui est une classe permettant de stocker des objets d'une classe *T* quelconque, mais qui possède les fonctions membres suivantes :

- constructeurs par défaut et copie, et destructeurs.
- affectation
- égalité et non égalité

La classe *Tableau* doit permettre de représenter des tableaux *statiques* dont la taille ne varie pas pendant l'exécution du programme ou *dynamiques*, dont la taille peut augmenter pendant l'exécution du programme. Les opérations que nous désirons pouvoir effectuer sur cette structure sont les suivantes :

- écriture de déclarations de la forme : *tableau de T [binf..bsup]*.
- accès direct au i^{ème} élément du tableau *x* en écrivant : *x[i]*.
- ajout d'éléments dans le tableau, soit en fin de tableau, soit à l'indice *i*.
- suppression soit d'un élément du tableau, soit du i^{ème} élément du tableau.
- obtention du rang d'un élément, s'il est présent dans le tableau.
- obtention du nombre d'éléments présents dans le tableau.
- ...

La déclaration de la classe *Tableau* paramétrée par le type *T* des éléments qui sont stockés dans les objets de cette classe est la suivante :

```
template <class T> class Tableau
```

```
{
  public :
    Tableau ( int bs, int bi = 0, int del = 0);
    ~Tableau();
    Tableau<T> & operator =( const Tableau<T> &) ;
    void Ajout ( const T & );
    void AjoutA( const T &, int );
    void Supp  ( const T &);
    void Supp  ( int);
    int Chercher( const T &);
    bool Appartient(const T &);
    void Vider();
    bool EstVide();
    bool EstPlein();
    int NbElements();
    int Taille();
    T& operator [] (int);
    typedef void (*TAction)( T & , void*);
    void PourChaque( TAction, void* );
    typedef bool (*TCondition)( const T & , void*);
    T * PremierQui( TCondition, void* );
    T * DernierQui( TCondition, void* );

  private :

    int binf, bsup, delta, nbmaxelem;
    T *t;
    int nbelem;
    int BaseZero( int pos){ return pos - binf;}
    int BaseBInf( int pos){ return pos + binf;}
};
```

Le préfixe **`template <class T>`** signifie qu'on déclare une classe dépendant du type ***T***. En d'autres termes la classe tableau est paramétrée par le type ***T***.

Deux types d'erreur peuvent survenir pendant l'utilisation du tableau :

- des erreurs d'utilisation qui peuvent être gérés par l'utilisateur du tableau, par exemple l'ajout d'un élément qui ne peut pas se réaliser : alors la fonction renvoie la valeur faux si l'ajout ne peut pas se faire, et 1 sinon.
- des erreurs qui ne peuvent pas être traitées par l'utilisateur, et nécessitent l'arrêt du programme : c'est le cas d'un tableau qui ne peut pas être créer (borne supérieure < à la borne inférieure ou manque de mémoire pou représenter le tableau). Dans ce cas on peut traiter de plusieurs façons :
-

  ➲ utiliser la fonction *assert(expression booléenne)* qui interrompt le programme avec l'expression, le nom du fichier, et la ligne : ceci ne veut rien dire pour l'utilisateur, et doit donc être utilisé pour la mise au point uniquement.

  ➲ lever une exception : on utilisera *try*, *catch*, et *throw* ... qui lève et gère des exceptions de type *TableauExcept* ...

```
template <class T> class TableauExcept
{
     char message [100];
     Tableau<T> * pt;
   public :
     TableauExcept( char * m, Tableau<T>* t)
     :pt(t)
     { strcpy(message, m); }
     void affiche()
     { cout <<typeid(*pt).name()
              <<" : "
              <<message;
     }
} ;
```

Le programme appelant une fonction qui risque de lever une exception, le fera de la façon suivante :

```
try
{
    partie de code pouvant lever une exception
} catch( TableauExcept<T> & e) { e.affiche();}
```

Pour lever une exception lors d'une erreur on écrira :

```
if ( la condition d'erreur )
throw( TableauExcept<T> (" le message ", this));
```

➲ écrire une fonction *erreur(char * mess)* qui affiche un message et arrête le programme :

```
void erreur ( char * mess)
{ cout<<mess; abort(); }
```

Constructeur, destructeur.

Le constructeur a trois arguments :

- **bs** : la borne supérieure du tableau.
- **bi** : la borne inférieure du tableau. Si la borne inferieure n'est pas présente, 0 est pris par défaut.
- **del** : le nombre d'éléments qu'on va ajouter au tableau, si le besoin s'en fait sentir. La valeur par défaut est 0, ce qui signifie que la taille du tableau ne peut pas augmenter.

```
template <class T> Tableau<T>::Tableau ( int bs, int bi = 0, int del = 0)
: binf(bi), bsup(bs), nbmaxelem(bs-bi+1), delta(del), nbelem(0)
{
   if(bi>bs) throw TableauExcept<T>("borne sup inférieure à borne inf", this);
   if(del<0) throw TableauExcept<T>("delta négatif", this);
   try{
       t = new T [nbmaxelem];
   } catch(bad_alloc ){
        throw TableauExcept<T>("plus de mémoire disponible", this);
   }
}
```

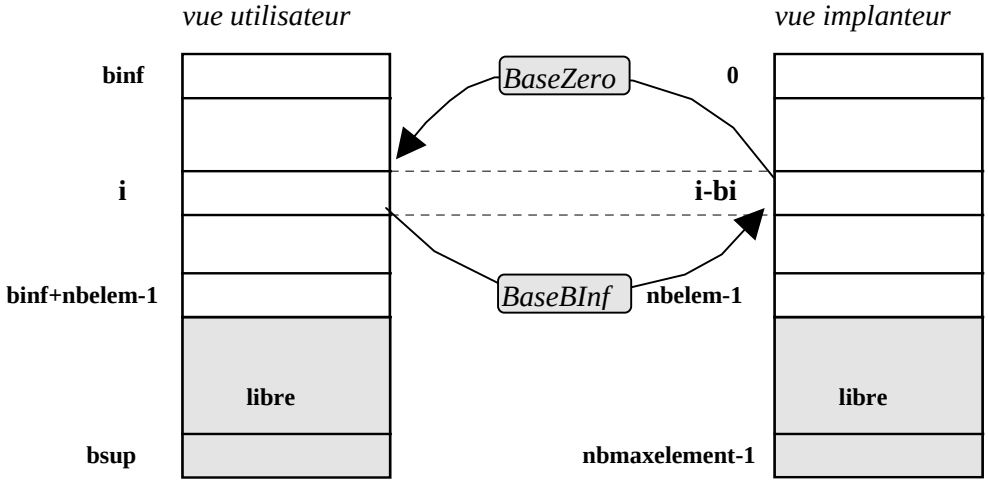
```
template <class T> Tableau<T>::Tableau const Tableau<T> & t)
: binf(t.binf), bsup(t.bsup), nbmaxelem(bsup-binf+1), delta(t.delta),
  nbelem(t.nbelem);
{
   try{
       t = new T [nbmaxelem];
   } catch(bad_alloc ){
       throw TableauExcept<T>("plus de mémoire disponible", this);
   }
}
```

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

Remarques :

1. au moment de la réservation de place pour tous les éléments du tableau, `t = new T [nbaxelement]` il y a appel du constructeur implicite associé au type *T* des éléments du tableau. Ce constructeur doit donc être défini correctement pour les objet du type *T*.
2. au moment de la libération de la place occupée par les éléments du tableau `delete [] t` il y a appel du destructeur associé a la classe *T* pour tous les éléments du tableau *t*.

Les indices des éléments du tableau *X* de la classe *tableau* ont des valeurs qui dépendent du point de vue :



Les fonctions *int BaseZero(int pos){ return pos - binf;}* et *int BaseBInf (int pos){ return pos + binf;}* ont pour rôle de passer d'une représentation à l'autre.

Opérateur [].

L'opérateur *[]* renvoie une référence à l'élément de rang *i* dans le tableau : cet élément doit exister sinon c'est une erreur. On ne peut permettre à l'utilisateur de la classe d'aller écrire n'importe où en mémoire ! Une autre façon de traiter l'erreur serait d'agrandir le tableau de façon qu'il contienne l'élément de rang i.

```
template <class T> T& Tableau<T>::operator [] (int i)
{
  if((i<binf)||(i-binf>=nbmaxelem))
       throw(TableauExcept<T>("indice en dehors des bornes",this));
  return t[BaseZero(i)];
}
```

Nombre d'éléments.

La fonction renvoie le nombre d'éléments stockés dans le tableau : la valeur contenue dans le champ *nbelem*.

```
template <class T> int Tableau<T>::NbElements()
{
      return nbelem;
}
```

La fonction précédente est différente de la fonction qui renvoie la taille du tableau :

```
template <class T> int Tableau<T>::Taille()
{
      return nbmaxelem;
}
```

Opérateur d'affectation

```
Tableau<T> & int Tableau<T>::operator =( const Tableau<T> &t){
    if(this !=&t){
       nbelem = t.nbelem ;
       binf= t.binf ;  bsup  = t.bsup ;
       nbmaxelem = t.nbmaxelem ; delta = t.delta ;
       delete [] rep ;
       try{
          rep = new T [ nbmaxelem] ;
          for(int i = 0 ; i<nbelem ; ++i ) rep[i]=t.rep[i] ;
```

```
      } catch( bad_alloc) {
         throw TableauExcept<T>(" plus de mémoire disponible", this);
      }
    }
    return *this ;
}
```

Chercher.

La fonction *chercher* retourne pour valeur le rang de l'élément *e* si cet élément se trouve dans le tableau et le rang du dernier élément + 1 sinon : L'opérateur == du type *T* est utilisé par cette fonction.

```
template <class T> int Tableau<T>::Chercher( const T & e)
{   int i;
    for(i=0; i<nbelem; ++i) if(e==t[i]) break;
    return BaseBInf(i);
}
```

La fonction *Appartient* renvoie vrai si l'élément *e* appartient au tableau et faux sinon :

```
template <class T> int Tableau<T>::Appartient( const T & e)
{   int i;
    for(i=0; i<nbelem; ++i) if(e==t[i]) break;
     return i!=nbelem;
}
```

Ajouts.

L'ajout en fin de tableau se fait de façon évidente : si le tableau est plein et que *delta* est différent de 0 alors la taille du tableau est augmentée de *delta* :

```
template <class T> void Tableau<T>::Ajout (const T & e)
{
      int i;
      if (nbelem=nbmaxelem)
      if (delta == 0) throw TableauExcept<T>("tableau plein", this);
      else         // extension du tableau
      {
         T * p=t;
         try {
             t = new T [nbmaxelem+delta];
         } catch bad_alloc) {
            throw TableauExcept<T>("plus de mémoire disponible", this);
         }
         nbmaxelem+=delta;
         for(i=0;i<nbelem;++i)t[i]=p[i];
         delete [] p;
      }
      t[nbelem]= e;
      ++nbelem;
}
```

Pour l'ajout à l'indice *p*, il faut commencer par libérer la place occupée par l'élément en place, avant d'insérer le nouvel élément. Cette libération peut se faire en respectant l'ordre des éléments en place, ce qui implique un décalage, ou plus simplement en rangeant en fin de tableau l'élément qui se trouve au rang *p*. Si *p* est en dehors des indices d'éléments du tableau, la fonction renvoie 0.

```
template <class T> int Tableau<T>::AjoutA (const T & e, int p)
```

```
{       // p doit être >= binf et <=nbelem
       int i;
       if ((p<binf)||(p>nbelem)) throw TableauExcept<T>("indice en dehors
                                                        des bornes", this);
       if (nbelem>=nbmaxelem)
       if (delta == 0) throw TableauExcept<T>("tableau plein", this);
       else                                    // extension du tableau
       {
          T * p=t;
          try {
              t = new T [nbmaxelem+delta];
          } catch bad_alloc) {
              throw TableauExcept<T>("plus de mémoire disponible", this);
          }
          for(i=0;i<nbelem;++i)t[i]=p[i];
          nbmaxelem+=delta;
          delete [] p;
       }
       for( i= nbelem; i>p; --i) t[i]=t[i-1];        // décalage
       t[p]= e;
       ++nbelem;
}
```

Les deux fonctions précédentes suggèrent la création d'une fonction *ReTaille* qui redimensionne le tableau si c'est possible, et lève une exception sinon :

```
template <class T> void Tableau<T>::ReTaille()
{
          T * p=t;
          try {
              t = new T [nbmaxelem+delta];
          } catch bad_alloc) {
              throw TableauExcept<T>("plus de mémoire disponible", this);
          }
          for(i=0;i<nbelem;++i)t[i]=p[i];
          nbmaxelem+=delta;
          delete [] p;
}
```

Suppressions.

Pour la suppression de l'élément à l'indice *p*, il nous faut « retasser » le tableau de façon à ce que les éléments du tableau restent représentés de façon consécutive, ou plus simplement le remplacer par le dernier élément.

```
template <class T> void  Tableau<T>::Supp  ( int p)
{       int i;
        if ((p<binf)||(BaseZero(p)>=nbelem))
              throw TableauExcept<T>("indice en dehors des bornes", this);
        for(i=BaseZero(p); i<nbelem-1;++i) t[i]=t[i+1];
        T e; t[nbelem] = e; // pour récupérer l'espace occupé par
                            // le dernier élément
        t[nbelem].~T();     // autre possibilité
        nbelem--;
}
```

Remarques :

1.Lorsque les objets qui sont stockés dans le tableau contiennent des données dynamiques, la libération de l'espace occupé par l'élément à supprimer ne se fait que s'il est remplacé par un autre élément (et dans ce cas à condition que l'affectation définie pour le type *T* le réalise). La libération de l'espace ne se fait donc pas pour la suppression du dernier élément, ou pour le dernier élément transféré lors de la suppression de l'élément de rang *i*. Nous sommes alors amenés à appeler le destructeur

pour cet élément s'il existe, c'est à dire si le type *T* en paramètre est une classe (on ne pourra pas avoir de tableau d'entiers ou de réels...). Il faut également que ce constructeur ait été défini de façon à pouvoir être appelé plusieurs fois...

2.Sinon, on peut remplacer l'élément dont on veut récupérer l'espace dynamique par un élément qui n'occupe pas d'espace dynamique : si on fait l'hypothèse que le constructeur par défaut ne réserve pas d'espace dynamique, on peut affecter un élément qui n'occupe pas de place au dernier élément.

La suppression associative, qui supprime la première occurrence d'un élément dans un tableau, peut se programmer en utilisant la fonction membre *chercher*, puis la suppression d'un élément connaissant son rang, ou programmer directement la suppression :

```
template <class T> void Tableau<T>::Supp  ( const T &e)
{
       int i;
       for(i=0; i<nbelem; ++i) if(e==t[i]) break;
       if (i==nbelem) throw TableauExcept<T>("élément absent", this);
       for( ; i<nbelem-1;++i) t[i]=t[i+1];
       nbelem--;
}
```

Remarque :

Si le type *T* est le type *int*, il y a ambiguïté entre la fonction membre de suppression associative et la fonction membre de suppression du $i^{ème}$ élément.

divers

```
template <class T> void Tableau<T>::Vider ()
{
      for(i = nbelem-1; i>=0; --i) supp(i+binf);
}
```

```
template <class T> bool Tableau<T>::EstVide()
{
      return nbelem==0;
}
```

```
template <class T> bool Tableau<T>::EstPlein()
{
      return (nbelem==nbmaxelem)&&(delta==0);
}
```

PourChaque et PremierQui

La fonction *PourChaque* permet d'exécuter une action avec ses arguments pour tous les élément stockés dans le tableau. La fonction *action* est du type : `typedef void (*TAction)( T & , void* args);` ou

```
typedef void TAction ( T & , void* args);
```

Le paramètre args est un paramètre qui sert à transmettre des arguments à la fonction action. Il peut y avoir un ou plusieurs arguments de types quelconques.

```
template <class T> void Tableau<T>::PourChaque( TAction action, void * args)
{     int i;
      for (i=0; i<nbelem;++i) action(t[i], args);
}
```

La fonction *PremierQui* renvoie un pointeur sur le premier élément du tableau satisfaisant la condition, s'il existe, et *NULL* sinon. Le type Tcondition est défini comme suit :

```
typedef bool TCondition( const T & , void* args); ou
      typedef bool (*TCondition)( const T & , void* args);
```

```
template <class T> T * Tableau<T>::PremierQui( TCondition condition, void *
args)
{     int i;
      for (i=0; i<nbelem;++i) if (condition(t[i], args)) break;
      if (i==nbelem) return NULL;
      else return &t[i];
}
```

```
template <class T> T * Tableau<T>::DernierQui( TCondition condition, void *
args)
{     int i;
      for (i=nbelem-1; i>=0;--i) if (condition(t[i], args)) break;
      if (i==-1) return NULL;
      else return &t[i];
}
```

Utilisation.

Un exemple de programme utilisant la classe tableau définie précédemment, pourrait être :

```
class Element
{
       int val;
      public :
        Element(){}
        Element(const Element & ):val(e.val){}
        Element( int v):val(v){}
        ~Element(){}
        Element & operator = ( const Element & e)
          { val=e.val;return *this;}
        int operator == ( const Element & x)const{ return val==x.val;}
         friend ostream& operator<<( ostream& o, Element & x){o<<x.val;return
o; }

};
```

```
void Imp ( Element & e, void * args)
{
      cout<<e<<(char*) args;
}
```

```
int egal( const Element & e, void * args)
{
      return e ==*(Element*)args;
}
```

```
void main()
{
      Tableau<Element> t(19, 10, 2);
      for(int i = 2;i<16; ++i)
      {
            try{
                  t.Ajout(i));
                  t.ecrire();cout<<"\n";
            }catch( TableauExcept & e) {
                  cout<< i<<" : ajout non fait\n";
            }
```

```
        }
        ...
         cout<<t.PremierQui(egal, Element(5);
        ...
}
```

Exercices.

1. On surchargera l'opérateur << des objets de la classe *ostream* pour afficher un tableau sous la forme : <*e1, e2, ... , en*>.

2. On surchargera les opérateurs << et >> des objets de la classe *fstream* de façon a pouvoir sauvegarder et restaurer les valeurs stockées dans un tableau de la classe *tableau*.

la classe tableau ordonné.

Afin d'accélérer la recherche d'un élément dans la structure de tableau, on peut être amené à faire en sorte que les éléments du tableau soient rangés dans l'ordre. Pour réaliser la classe *tableau ordonné*, nous supposerons que la classe *T* est munie de l'opérateur <, en plus des opérateurs d'affectation et d'égalité. Dans le cas d'un tableau ordonné, l'ajout dans le tableau n'a plus de sens : il s'agit d'un ajout d'un élément à sa place dans le tableau.

```
template <class T> class OTableau
{
  public :
    OTableau ( int bs, int bi = 0, int del = 0);
    ~OTableau();
    void Ajout ( const T & );
    // void AjoutA( const T &, int ); devient illégal
    void Supp  ( const T &);
    void Supp  ( int);
    int Chercher( const T &);
    bool Appartient(const T &);
    void Vider();
    bool EstVide();
    bool EstPlein();
    int NbElements();
    int Taille();
    T& operator [] (int);
    typedef void (*TAction)( T & , void*);
    void PourChaque( TAction, void* );
    typedef int (*TCondition)( const T & , void*);
    T * PremierQui( TCondition, void* );
    T * DernierQui( TCondition, void* );

  private :

    int binf, bsup, delta, nbmaxelem;
    T *t;
    int nbelem;
    int BaseZero( int pos){ return pos - binf;}
    int BaseBInf( int pos){ return pos + binf;}
};
```

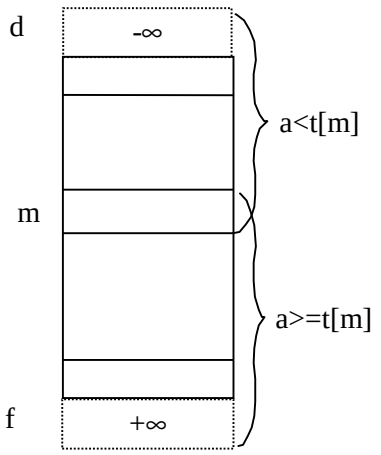
Les fonctions membres étant définies comme suit :

Constructeur, destructeur, Nombre d'éléments, Opérateur [], etc...

Rien ne change ...

Chercher, Appartient

La fonction *chercher* retourne pour valeur le rang de l'élément *a* si cet élément se trouve dans le tableau et le nombre d'éléments + 1 sinon : dans le cas d'un tableau ordonné on peut utiliser une recherche dichotomique plus rapide qu'une recherche séquentielle :



- Invariant de l'itération : *t[d]<=a & t[f]>a*
- Arrêt si *d=f-1*
- Donc quand l'itération s'arrête :
  - *t[f]>a* ⇒ *f* est l'indice du premier élément supérieur à *a*, et c'est là qu'il faut insérer *a*.
  - *t[d]<=a* ⇒Si *d!=-1* & *t[d]==a*, on a trouvé

```
template <class T> int OTableau<T>::chercher( const T &a)
{   int m, d, f;
    d=-1; f=nbelem;
    while (d<f-1){m=(d+f)/2; if (a<t[m]) f=m ; else d=m;}
    if (d==-1) return bi-1;
    if (t[d]==a) return d+bi; else return bi-1;
}
```

S'il est préférable que la fonction *chercher* retourne l'indice où il faudrait insérer l'élément cherché lorsque celui-ci ne se trouve pas dans le tableau, nous écririons :

```
template <class T> int OTableau<T>::chercher( const T &a)
{   int m, d, f;
    d=-1; f=nbelem;
    while (d<f-1){m=(d+f)/2; if (a<t[m]) f=m ; else d=m;}
    if (d==-1) return f+bi;
    if (t[d]==a) return d+bi; else return f+bi;

}
```

Ajouts

La recherche de la place où insérer l'élément *a* et la libération de la place pour ranger *a* se font dans la même itération :

```
template <class T> void OTableau<T>::ajout( const T &a)
{      int i;
       if (nbelem>=nbmaxelem)
       if (delta == 0) throw TableauExcept<T>("tableau plein", this);
       else         // extension du tableau
       {
           ReTaille();
       }
       // déplacement et recherche de place
       for(j=nbelem-1;((j>=0)&&(a<t[j]));--j) t[j+1] = t[j];
       nbelem+=1;
       t[j+1] = a;
}
```

Suppressions.

La suppression de l'élément à l'indice *i*, se fait de la même façon que pour un tableau non ordonné, en revanche pour la suppression associative, on peut commencer à faire une recherche dichotomique pour trouver le rang le l'élément à supprimer, puis le supprimer par ma suppression précédente.

Exercices.

1. On surchargera l'opérateur << des objets de la classe *ostream* pour afficher un tableau sous la forme : < , , ... , >.
2. On surchargera les opérateurs << et >> des objets de la classe *fstream* de façon a pouvoir sauvegarder et restaurer les valeurs stockées dans un tableau de la classe *tableau*.

introduction à l'héritage.

Dans les deux classes précédentes *Tableau* et *0Tableau* nous constatons que de nombreuses fonctions membres ont le même code. Pour éviter la redéfinition de ces fonctions membres, nous pourrions utiliser le concept d'***héritage*** :

Lorsqu'une classe ***B*** hérite d'une classe ***A,*** elle hérite de toutes ses données membres ou fonctions membres. En revanche les accès aux fonctions dépendent des mécanismes suivants (si le modificateur d'héritage est *public*) :

- Celles qui sont définies dans les rubriques *private* ne sont pas accessibles.
- Celles qui sont définies dans les rubriques *protected* sont accessibles de puis les fonctions membres de la classe dérivée, mais pas par l'utilisateur de la classe dérivée.
- Celles qui sont définies dans les rubriques *public*, sont accessibles depuis les fonctions membres de la classe et par l'utilisateur de la classe dérivée.

Ce mécanisme est valable sauf, évidemment si une fonction est redéfinie dans la classe dérivée.

Il y a plusieurs façons d'envisager l'héritage pour les classes *tableau* et *tableau ordonné* :

1. la classe *tableau ordonné* hérite de la classe *tableau* :

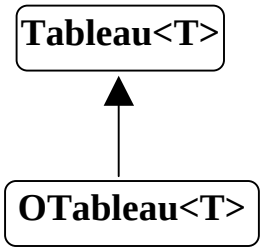


Tableau protected : int binf, bsup, nbmaxelem, nbelem, delta;
T *t;
BaseZero, BaseBInf

Tableau public : Tableau, ~ Tableau, ajouts, suppressions, [], Chercher, NbElements

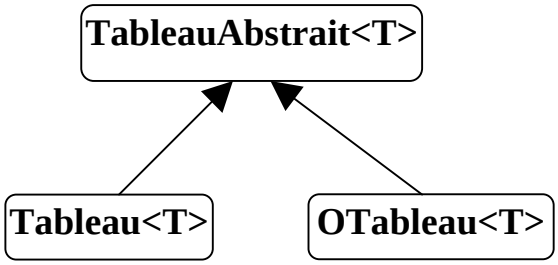
Otableau public : Otableau, ajouts, Chercher

Dans ce schéma d'héritage, la classe *OTableau* hérite de toutes les données membres de la classe *Tableau*.

- Elle hérite également des fonctions membres ~Tableau, des suppressions, ainsi que de la fonction nombre d'éléments.
- Le constructeur tableau ne peut être hérité, car **les constructeurs ne sont jamais hérités**.
- Les ajouts sont forcément redéfinis car ils doivent maintenir la propriété du tableau d'être ordonné.
- La fonction chercher est surchargée par une fonction chercher plus rapide lorsque le tableau est ordonné.

Pour un tableau ordonné, les ajouts à l'indice *i* et en fin de tableau sont dépourvus de sens. il faudra les surcharger : l'ajout en fin est surchargé par l'ajout d'un élément à sa place, ajout qui, du coup, masque l'ajout en *i*.

2. La deuxième façon d'organiser l'héritage est la suivante : on regroupe dans une classe *TableauAbstrait* les fonction et données communes aux deux classes :



Pour interdire la création d'instances de la classe **TableauAbstrait**, il suffit de ranger le constructeur dans la zone protégée.

```
template <class T> class TableauAbstrait
{
  public :
        ~TableauAbstrait();

        int EstVide();
        int EstPlein();
        int NbElements();
        int Taille();
        T& operator [] (int);
        void Vider();
        void Supp  ( int );
        typedef void (*TAction)( T & , void*);
        void PourChaque( TAction, void* );
        typedef int (*TCondition)( const T & , void*);
        T * PremierQui( TCondition, void* );
        T * DernierQui( TCondition, void* );

  protected :
        TableauAbstrait( int bs, int bi = 0, int del = 0);

        int binf, bsup, delta, nbmaxelem;
        T *t;
        int nbelem;
        int BaseZero( int pos)const{ return pos - binf;}
        int BaseBInf( int pos)const{ return pos + binf;}
        void ReTaille();

};
```

La classe dérivée *Tableau* est définie ainsi :

```
template <class T> class Tableau : public TableauAbstrait<T>
{
  public :
         Tableau( int bs, int bi = 0, int del = 0);
        ~Tableau();
        void Ajout ( const T & );
        void AjoutA ( const T & , int);
        void Supp  ( const T &);
        int Chercher( const T &);
        bool Appartient(const T &);
};
```

```
template <class T> Tableau<T>::Tableau( int bs, int bi = 0, int del = 0)
: TableauAbstrait<T>(bs, bi, del)
{}
```

La classe dérivée *OTableau* est définie ainsi :

```
template <class T> class OTableau : public TableauAbstrait<T>
{
  public :
         OTableau( int bs, int bi = 0, int del = 0);
        ~OTableau();
```

```
        void Ajout ( const T & );
       void Supp  ( const T &);
        int Chercher( const T &);
        bool Appartient(const T &);
};
```

```
template <class T> Tableau<T>::OTableau( int bs, int bi = 0, int del = 0)
: TableauAbstrait<T>(bs, bi, del)
{}
```

Les modèles et la modularité.

Les modules contenant des modèles de classe ou de fonction ne peuvent pas être précompilés de la même façon que les modules contenant des classes : il faut que les types en paramètre du modèle de classe soient instanciés pour que la compilation soit autre chose qu'une simple vérification syntaxique et produise un fichier ***.obj***.

Considérons le fichier *x.h* constitué de la façon suivante :

```
template <class T> class X
{ private : ...
   public : fx();
};
```

Le fichier *x.cpp* contient la définition de la fonction membre *fx* :

```
#include "x.h"
template <class T> X<T>::fx(){}
```

le fichier *essai.cpp* contient :

```
#include "x.h"
void main()
{ X<int> x;
   x.fx();
```

}

L'éditeur de liens génère alors le message suivant :

undefined symbol x<int>::fx() in module essai.cpp