INF3105 – Tableaux dynamiques et génériques

Éric Beaudry

Université du Québec à Montréal (UQAM)

2017A

Sommaire

1. Tableau natif C++
2. Tableau dynamique
3. Généricité en C++ (Templates)
4. Tableau abstrait générique
5. Lab3

Allocation automatique

- La taille doit être connue (déductible) lors de la compilation.
- La taille doit être fixée dans le code source.
- Généralement, impossible de spécifier la taille à l'exécution.

Sur la pile

```
int main(){
  int tableau1[10];
  int tableau2[10] = { 0, 1, 2, 3, 4, 5,
       6, 7, 8, 9};
  int tableau3[10] = { 0, 1, 2, 3, 4, 5};
}
```

Dans un objet

```
class A{
  public:
    ...
  private:
    int    tableau1[10];
    double tableau2[10];
};
```

Allocation dynamique

- La taille peut être inconnue lors de la compilation.
- Allocation sur le tas (*heap*) explicite par l'opérateur `new`.
- La taille demeure fixe après l'allocation.
- Libération de la mémoire par l'opérateur `delete []`.

Dans une fonction

```
int main(){
  int* tableau1 = new int[10];
  int* tableau2 = new int[10] { 0, 1,
       2, 3, 4, 5, 6, 7, 8, 9};
  ...
  delete[] tableau1;
  delete[] tableau2;
}
```

Dans un objet

```
class A{
  public:
    A(int n=10) {tableau=new int[n];}
    ~A(){delete[] tableau;}
  private:
    int*   tableau;
};
```

Que fait ce programme ?

```
#include <iostream>
int main() {
    int tab1[5], tab2[5];
    for(int i=0;i<5;i++){
      tab1[i] = i;      tab2[i] = i + 10;
    }
    std::cout << "tab1[0..15] :";
    for(int i=0;i<16;i++)        cout << " " << tab1[i];
    std::cout << std::endl;
    for(int i=0;i<15;i++)        tab1[i] = 99 − i;

    std::cout << "tab1 :";
    for(int i=0;i<5;i++)       std::cout << " " << tab1[i];
    std::cout << std::endl;

    std::cout << "tab2 :";
    for(int i=0;i<5;i++)       std::cout << " " << tab2[i];
    std::cout << std::endl;

    return 0;
}
```

Limites des tableaux natifs de C++

- **Taille fixe**.
- Pour contourner le problème de taille fixe, il faut :
  1. allouer un nouveau tableau ;
  2. copier les éléments de l'ancien vers le nouveau ;
  3. libérer l'ancien tableau.
- Un tableau est accessible via un pointeur.
- La **taille du tableau est dissociée du pointeur**.
- **Non-vérification des indices** lors de l'accès aux éléments.
- Conclusion : pas possible de manipuler un tableau comme un objet abstrait.
- Besoin d'une structure de données abstraite de type **tableau**.

Objectifs

- Manipuler des tableaux comme des objets.
- Faire abstraction des pointeurs et de la gestion de mémoire.
- Modifier la taille d'un tableau (taille dynamique).
- Vérifier les indices lors de l'accès afin de déceler les bogues le plus tôt possible.
- Comment : Créer un type abstrait de données de type `Tableau<T>`.

Utilisation souhaitée

Nous aimerions manipuler les tableaux de la même façon que des objets. Exemple :

```
int main(){
  TableauInt tab1;
  tab1.ajouter(1); tab1.ajouter(2); tab1.ajouter(3);
  int a = tab1[1];
  TableauInt tab2 = tab1; // fait une copie de tab1
  tab1.ajouter(1); tab1.ajouter(2); tab1.ajouter(3);
  tab2[1] = 34;
  assert(a == tab1[1]); // le tableau tab1 doit ne pas etre modifie
  for(int i=0;i<tab1.taille();i++)
    cout << tab1[i] << endl;
  return 0;
}
```

Solution

- Encapsulation d'un tableau natif à l'intérieur d'un objet Tableau.
- La classe `Tableau` **encapsule** :
  - un pointeur vers un tableau natif ;
  - la capacité du tableau natif ;
  - le nombre d'éléments dans le tableau abstrait ($\leq$ capacité) ;
- Les accès au tableau passent par une interface publique.
- Accès indirect au tableau natif.

La classe TableauInt

tableauint.h

```
class TableauInt{
public:
  TableauInt();
  ~TableauInt();
  void ajouter(int nombre);
  int& operator[](int index);
private:
  int*   entiers;
  int    capacite;
  int    taille;
};
```

tableauint.cpp

```
TableauInt::TableauInt(){
    capacite = 4;
    taille = 0;
    entiers = new int[capacite];
}
TableauInt::~TableauInt(){
    delete[] entiers;
}
```

Suite tableau.cpp

```
void TableauInt::ajouter(int nombre){
    assert(taille<capacite);
    entiers[taille++]=nombre;
}
int& TableauInt::operator[](int index){
    assert(index>=0 && index<taille);
    return entiers[index];
}
```

Avec réallocation transparente / automatique

```
void TableauInt::ajouter(int nombre){
    if(taille==capacite){
        capacite++; // Methode naive : O(n)
        int* nouveautab = new int[capacite];
        for(int i=0;i<taille;i++)
            nouveautab[i] = entiers[i];
        delete[] entiers;
        entiers = nouveautab;
    }
    entiers[taille++]=nombre;
}
```

Avec réallocation transparente / automatique

```
void TableauInt::ajouter(int nombre){
    if(taille==capacite){
        capacite *= 2; // cout amortie : O(1)
        int* nouveautab = new int[capacite];
        for(int i=0;i<taille;i++)
            nouveautab[i] = entiers[i];
        delete[] entiers;
        entiers = nouveautab;
    }
    entiers[taille++]=nombre;
}
```

Problématique

On peut avoir besoin de plusieurs classes et fonctions similaires, mais légèrement différentes au niveau des types.

TableauPoints

```
class TableauPoints{
  public:
    ...
    void ajouter(const Point& p);
    ...
  private:
    Point* points;
    int    capacite;
    int    nbpoints;
};
```

TableauStrings

```
class TableauStrings{
  public:
    ...
    void ajouter(const String& p);
    ...
  private:
    String* strings;
    int    capacite;
    int    nbstrings;
};
```

Mécanisme de généricité dans C++

- Écriture d'un modèle générique (un *template*).
- Variables de type (nous la noterons souvent `T`).
- À chaque instanciation d'un modèle :
  - «copier-coller» du modèle ;
  - «recherche-remplacer» pour affecter la variable de type à une type précis.

Code écrit par le programmeur

```
template <class T>
class Point{
    T x, y;
  public:
    Point(T x_, T y_);
};
template <class T>
Point::Point(T x_, T y_)
: x(x_), y(y_){}

int main(){
  Point<float> p1;
  Point<double> p2;
}
```

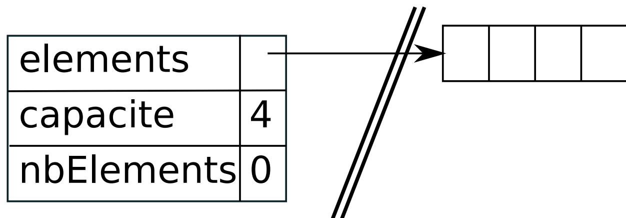
Instanciation par le compilateur

```
class Point<float>{
    float x, y;
  public:
    Point<float>(float x_, float y_);
};
Point<float>::Point<float>(float x_, float y_)
: x(x_), y(y_){}

class Point<double>{
    double x, y;
  public:
    Point<double>(double x_, double y_);
};
Point<double>::Point<double>(double x_, double
        y_)
: x(x_), y(y_){}

int main(){
  Point<float> p1;
  Point<double> p2;
}
```

Déclaration

```
template <class T>
class Tableau {
 public:
  Tableau(int capacite_initiale=4);
  ~Tableau();
  int taille() const {return nbElements;}
  void ajouter(const T& item);
  T& operator[] (int index);
  const T& operator[] (int index) const;
 private:
  T*          elements;
  int         capacite;
  int         nbElements;
  void        redimensionner(int nouvCapacite);
};
// mettre la suite ici OU dans tableau.hcc qu'on inclut ici.
```

Représentation abstraite en mémoire



Constructeur et destructeur

```
template <class T>
Tableau<T>::Tableau(int initCapacite) {
  capacite = initCapacite;
  nbElements = 0;
  elements = new T[capacite];
}
template <class T>
Tableau<T>::~Tableau() {
   delete[] elements;
   elements = NULL; // optionnel
}
```

Ajout

```
template <class T> void Tableau<T>::ajouter(const T& item)
{
    if(nbElements >= capacite)
        redimensionner(capacite*2);
    elements[nbElements++] = item;
}
template <class T> void Tableau<T>::redimensionner(int nouvCapacite)
{
    capacite = nouvCapacite;
    T* temp = new T[capacite];
    for(int i=0;i<nbElements;i++)
        temp[i] = elements[i];
    delete [] elements;
    elements = temp;
}
```

operator[]

```
template <class T>
T& Tableau<T>::operator[] (int index){
    assert(index<nbElements);
    return elements[index];
}

template <class T>
const T& Tableau<T>::operator[] (int index) const{
    assert(index<nbElements);
    return elements[index];
}
```

operator[]

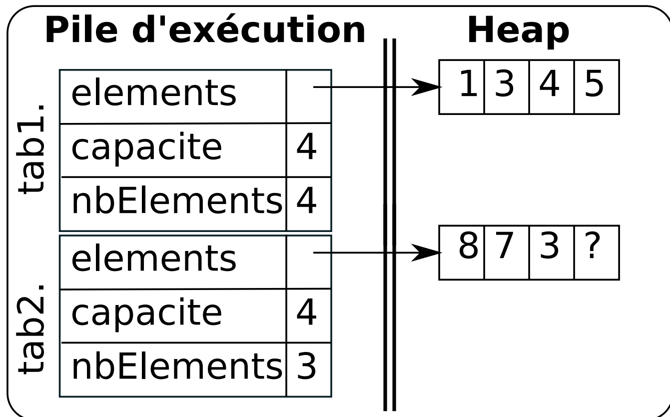
```
void affiche(const Tableau<int>& tab){
    for(int i=0;i<tab.taille();i++)
        cout << tab[i] << endl;
}
int main(){
    Tableau<int> tab;
    for(int i=0;i<10;i++)
        tab.ajouter(i);
    for(int i=0;i<tab.taille();i++)
        tab[i] *= 2;
    affiche(tab);
    return 0;
}
```

Affectation (operator=)

```
void fonction(){
    Tableau<int> tab1();
    tab1.ajouter(1); tab1.ajouter(3); tab1.ajouter(4); tab1.ajouter(5);
    Tableau<int> tab2();
    tab2.ajouter(8); tab2.ajouter(7); tab2.ajouter(3);
    tab1 = tab2; // cela devrait copier tab2 vers tab1
}
```

- Que se passe-t-il ?
- L'opérateur égal n'ayant pas été surchargé, le compilateur en génère un.
- Ce dernier ne fait qu'appeler l'opérateur = sur les variables de Tableau.

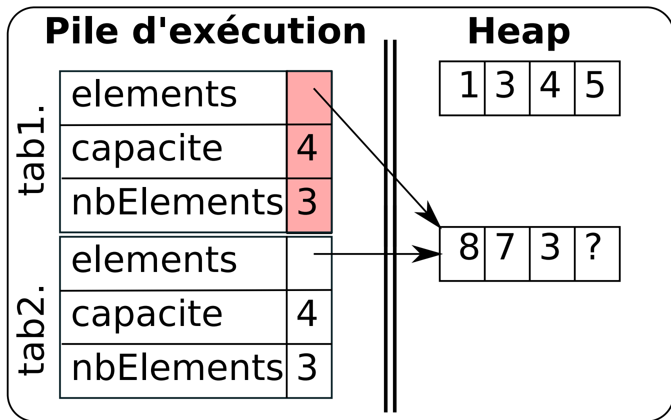
Représentation AVANT tab1=tab2.



Code synthétisé par le compilateur pour Tableau : :operator =

```
template <class T>
Tableau& operator Tableau<T>::operator=(const Tableau<T>& autre){
    elements = autre.elements;
    capacite = autre.capacite;
    nbElements = autre.nbElements;
}
```

Représentation APRÈS tab1=tab2.



Bon code pour Tableau : :operator =

```
template <class T>
Tableau<T>& Tableau<T>::operator = (const Tableau<T>& autre){
    if(this==&autre) return *this;//cas special lorsqu'on affecte un objet a
      lui—meme
    nbElements = autre.nbElements;
    if(capacite<autre.nbElements){
        delete[] elements;
        capacite = autre.nbElements; //ou autre.capacite
        elements = new T[capacite];
    }
    for(int i=0;i<nbElements;i++)
        elements[i] = autre.elements[i];
    return *this;
}
```

Constructeur par copie d'un tableau

```
int fonction(Tableau<int> tab){
// tab par valeur et non pas par
      reference
   int sommedouble=0;
   for(int i=0;i<tab.taille();i++){
      tab[i]*=2;
      sommedouble += tab[i];
   }
   return sommedouble;
}
```

```
int main(){
   Tableau<int> t;
   t.ajouter(1);
   t.ajouter(2);
   t.ajouter(3);
   cout << fonction(t) << endl;
   cout << fonction(t) << endl;
   return 0;
}
```

- Que se passera-t-il ?
- Indice : similaire à l'opérateur =.

Code synthétisé par le compilateur pour le constructeur par copie

```
template <class T>
Tableau<T>::Tableau(const Tableau<T>& autre)
 : elements(autres.elements),
   capacite(autre.capacite),
   nbElements(autre.nbElements)
{
}
```

Bon code pour le constructeur par copie

```
template <class T>
Tableau<T>::Tableau(const Tableau& autre) {
    capacite = autre.nbElements; // ou autre.capacite
    nbElements = autre.nbElements;
    elements = new T[capacite];
    for(int i=0;i<nbElements;i++)
        elements[i] = autre.elements[i];
}
```

Création d'une classe générique Tableau<T> en C++

- Lectures préalables : Sections 2 et 4 des notes de cours.
- Tâches :
  1. Prendre connaissance des fichiers source dans lab3.zip.
  2. Compléter la classe générique `Tableau` tel que présentée dans les notes de cours.
  3. Tester avec `test_tab.cpp`.
  4. Résoudre un problème simple (nuage de points) en appliquant un tableau.
- `http://ericbeaudry.ca/INF3105/lab3/`