INF3105 – Tableaux dynamiques et génériques

Jaël Champagne Gareau

Université du Québec à Montréal (UQAM)

Été 2024

http://cria2.uqam.ca/INF3105/

UQÀM

Sommaire

Tableaux natifs alloués automatiquement

- Généralement : la taille doit être fixe et calculable à la compilation.
- Contournable dans une fonction (sur la pile, avec VLA : pas standard).
- Incontournable pour les objets : il faut absolument une taille fixe.

Dans une fonction (sur la pile)

```
1  int main() {
2    int tableau1[10];
3    int tableau2[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
4
5    // VLA: pas dans le standard C++
6    int n = ...;
7    int tableau3[n];
8  }
```

Dans un objet

```
1  class A{
2    public:
3      ...
4    private:
5      int    tableau1[10];
6      double tableau2[10];
7  };
```

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 (sur la pile)

```
1  int main() {
2    int* tab1 = new int[5];
3    int* tab2 = new int[5] {0, 1, 2, 3, 4};
4    // ...
5    delete[] tab1;
6    delete[] tab2;
7  }
```

Dans un objet

```
1  class A {
2    public:
3      A(int n = 5) {tab = new int[n];}
4      ~A() {delete[] tab;}
5    private:
6      int* tab;
7  };
```

Non-vérification des indices

```
1   int tab1[5], tab2[5];
2   for(int i = 0; i < 5; i++)
3     tab1[i] = i;   tab2[i] = i + 10;
4   std::cout << "tab1[0..15] :";
5   for(int i = 0; i < 16; i++)       std::cout << " " << tab1[i];
6   std::cout << std::endl;
7   for(int i = 0; i < 15; i++)       tab1[i] = 99 - i;
8
9   std::cout << "tab1 :";
10  for(int i = 0; i < 5; i++)       std::cout << " " << tab1[i];
11  std::cout << std::endl;
12
13  std::cout << "tab2 :";
14  for(int i = 0; i < 5; i++)       std::cout << " " << tab2[i];
15  std::cout << std::endl;
```

Utilisation laborieuse des tableaux natifs (1)

```
1  #include <iostream>
2  int somme(int* tab, int n) { // 2 paramètres (objets) pour le tableau
3      int s = 0;
4      for(int i = 0; i < n; i++)
5          s += tab[i];
6      return s;
7  }
8  int main() {
9      int n = 0;
10     cin >> n;
11     int* tab = new int[n];
12     for(int i = 0; i < n; i++)
13         cin >> tab[n];
14     cout << somme(tab, n) << endl;
15     delete[] tab;
16 }
```

Utilisation laborieuse des tableaux natifs (2)

```
1  #include <iostream>
2  int* filtrerpairs(int* tab, int n, int& nbpairs) {
3      nbpairs = 0;
4      for(int i = 0; i < n; i++)
5          if(tab[i] % 2 == 0) nbpairs++;
6      int* pairs = new int[nbpairs];
7      nbpairs = 0;
8      for(int i = 0; i < n; i++)
9          if(tab[i] % 2 == 0)
10             pairs[nbpairs++] = tab[i];
11     return pairs;
12 }
```

```
1  int main() {
2      int n, m;
3      cin >> n;
4      int* tab = new int[n];
5      for(int i = 0; i < n; i++)
6          cin >> tab[n];
7      int* pairs = filtrerpairs(tab, n, m);
8      for(int i = 0; i < m; i++)
9          cout << pairs[i] << endl;
10     delete[] pairs;
11     delete[] tab;
12 }
```

Limites des tableaux natifs de C++

- **Taille fixe**.
- Pour contourner la limitation 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 : manipulation laborieuse des tableaux natifs.
- Besoin d'une structure de données abstraite de type **tableau**.

Objectifs

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

Utilisation conviviale et souhaitée (1)

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

```
1  int somme(const TableauInt& tab) {
2    int s = 0;
3    for(int i = 0; i < tab.taille(); i++) s += tab[i];
4    return s;
5  }
6  int main() {
7    TableauInt t;
8    int n, e;
9    cin >> n;
10   for(int i = 0; i < n; i++) {
11     cin >> e;
12     t.ajouter(e);
13   }
14   cout << somme(t);
15 }
```

Utilisation conviviale et souhaitée (2)

```
1  TableauInt filtrepairs(const TableauInt& tab) {
2    TableauInt pairs;
3    for(int i = 0; i < tab.taille(); i++)
4        if(tab[i] % 2 == 0) pairs.ajouter(tab[i];
5    return pairs;
6  }
7  int main() {
8    TableauInt t;
9    int n, e;
10   cin >> n;
11   for(int i = 0; i < n; i++) {
12     cin >> e;
13     t.ajouter(e);
14   }
15   t = filtrepairs(t);
16   for(int i = 0; i < t.taille(); i++)
17     cout << t[i] << endl;
18 }
```

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

```
1  class TableauInt {
2  public:
3    TableauInt();
4    ~TableauInt();
5    void ajouter(int nombre);
6    int& operator[](int index);
7  private:
8    int*  entiers;
9    int   capacite;
10   int   taille;
11 };
```

tableauint.cpp

```
1  TableauInt::TableauInt() {
2      capacite = 4;
3      taille = 0;
4      entiers = new int[capacite];
5  }
6
7  TableauInt::~TableauInt() {
8      delete[] entiers;
9  }
```

Suite tableauint.cpp

```
1  void TableauInt::ajouter(int nombre) {
2      assert(taille < capacite);
3      entiers[taille++] = nombre;
4  }
5
6  int& TableauInt::operator[](int index) {
7      assert(index >= 0 && index < taille);
8      return entiers[index];
9  }
```

Avec réallocation transparente / automatique

```
1  void TableauInt::ajouter(int nombre) {
2      if(taille == capacite) {
3          capacite++; // Méthode naïve : O(n)
4          int* nouveautab = new int[capacite];
5          for(int i = 0; i < taille; i++)
6              nouveautab[i] = entiers[i];
7          delete[] entiers;
8          entiers = nouveautab;
9      }
10     entiers[taille++] = nombre;
11 }
```

Avec réallocation transparente / automatique

```
1  void TableauInt::ajouter(int nombre) {
2      if(taille == capacite) {
3          capacite *= 2; // coût amorti : O(1)
4          int* nouveautab = new int[capacite];
5          for(int i = 0; i < taille; i++)
6              nouveautab[i] = entiers[i];
7          delete[] entiers;
8          entiers = nouveautab;
9      }
10     entiers[taille++] = nombre;
11 }
```

Nécessité de généricité

Besoin de plusieurs classes et fonctions similaires, mais légèrement différentes.

TableauPoints

```
1  class TableauPoints {
2    public:
3      void ajouter(const Point& p);
4      ...
5    private:
6      Point* points;
7      int    capacite;
8      int    nbpoints;
9  };
```

TableauStrings

```
1  class TableauStrings {
2    public:
3      void ajouter(const String& p);
4      ...
5    private:
6      String* strings;
7      int     capacite;
8      int     nbstrings;
9  };
```

Mécanisme de généricité dans C++

- Écriture d'un modèle générique (un *template*).
- Ce modèle a ≥ 1 variable(s) de type (souvent notée `T`).
- À chaque instanciation d'un modèle :
  - «copier-coller» du modèle ;
  - «rechercher-remplacer» pour attribuer un type précis à la variable de type.

Code écrit par le programmeur

```
1  template <class T>
2  class Point {
3      T x, y;
4    public:
5      Point(T x_, T y_);
6  };
7  template <class T>
8  Point::Point(T x_, T y_) : x(x_), y(y_) {}
9
10 int main() {
11   Point<float> p1;
12   Point<double> p2;
13 }
```

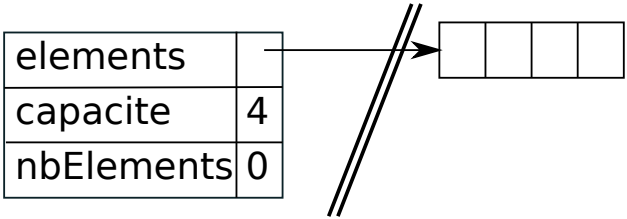
Instanciation par le compilateur

```
1  class Point<float> {
2      float x, y;
3    public: Point<float>(float x_, float y_);
4  };
5  Point<float>::Point<float>(float x_, float y_)
6          : x(x_), y(y_){}
7
8  class Point<double>{
9      double x, y;
10   public: Point<double>(double x_, double y_);
11 };
12 Point<double>::Point<double>(double x_, double y_)
13         : x(x_), y(y_) {}
14 int main() {
15   Point<float> p1;
16   Point<double> p2;
17 }
```

Déclaration

```
1  template <class T>
2  class Tableau {
3    public:
4      Tableau(int capacite_initiale = 4);
5      ~Tableau();
6      int taille() const {return nbElements;}
7      void ajouter(const T& item);
8      T& operator[] (int index);
9      const T& operator[] (int index) const;
10   private:
11     T*          elements;
12     int         capacite, nbElements;
13     void        redimensionner(int nouvCapacite);
14 };
15 // mettre la suite ici OU dans tableau.hcc qu'on inclut ici.
```

Représentation abstraite en mémoire



Constructeur et destructeur

```
1  template <class T>
2  Tableau<T>::Tableau(int initCapacite) {
3    capacite = initCapacite;
4    nbElements = 0;
5    elements = new T[capacite];
6  }
7
8  template <class T>
9  Tableau<T>::~Tableau() {
10   delete[] elements;
11   elements = nullptr; // optionnel
12 }
```

Ajout

```
1  template <class T> void Tableau<T>::ajouter(const T& item) {
2      if(nbElements >= capacite)
3          redimensionner(capacite * 2);
4      elements[nbElements++] = item;
5  }
6
7  template <class T> void Tableau<T>::redimensionner(int nouvCapacite) {
8      capacite = nouvCapacite;
9      T* temp = new T[capacite];
10     for(int i = 0; i < nbElements; i++)
11         temp[i] = elements[i];
12     delete[] elements;
13     elements = temp;
14 }
```

operator[]

```
1  template <class T>
2  T& Tableau<T>::operator[] (int index) {
3      assert(index < nbElements);
4      return elements[index];
5  }
6
7  template <class T>
8  const T& Tableau<T>::operator[] (int index) const {
9      assert(index < nbElements);
10     return elements[index];
11 }
```

operator[]

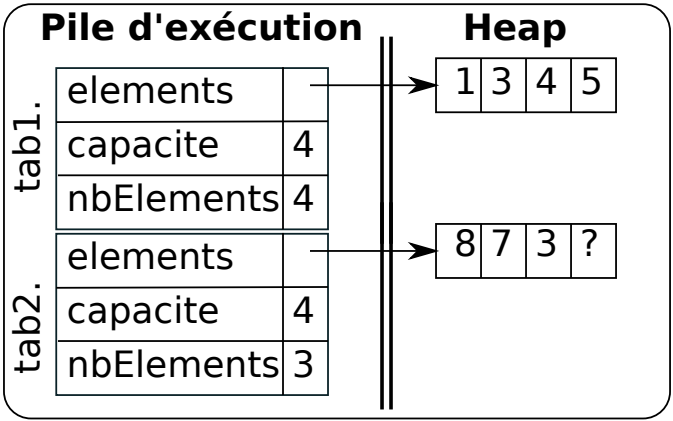
```
1  void affiche(const Tableau<int>& tab) {
2      for(int i = 0; i < tab.taille(); i++)
3          cout << tab[i] << endl;
4  }
5
6  int main() {
7      Tableau<int> tab;
8      for(int i = 0; i < 10; i++) tab.ajouter(i);
9      for(int i = 0; i < tab.taille(); i++) tab[i] *= 2;
10     affiche(tab);
11     return 0;
12 }
```

Affectation (operator=)

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

- 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.

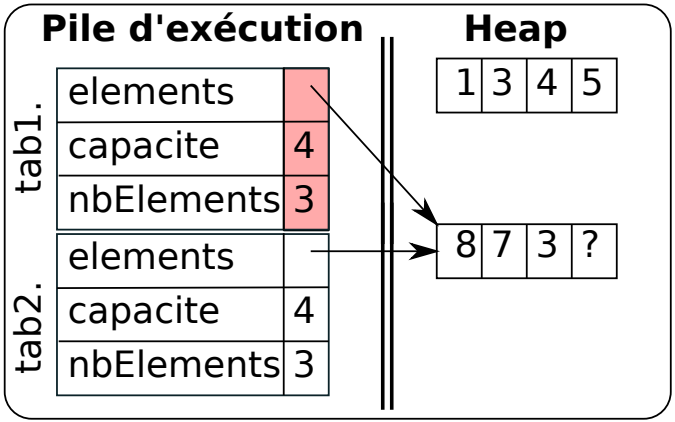
Représentation AVANT tab1 = tab2.



Code synthétisé par le compilateur pour Tableau : :operator =

```
1 template <class T>
2 Tableau& operator Tableau<T>::operator=(const Tableau<T>& autre) {
3     elements = autre.elements;
4     capacite = autre.capacite;
5     nbElements = autre.nbElements;
6     return *this;
7 }
```

Représentation APRÈS tab1 = tab2.



Bon code pour Tableau : :operator =

```
1  template <class T>
2  Tableau<T>& Tableau<T>::operator = (const Tableau<T>& autre) {
3      if(this == &autre) return *this; // Affectation à soi−même
4      nbElements = autre.nbElements;
5      if(capacite < autre.nbElements) {
6          delete[] elements;
7          capacite = autre.nbElements; // ou autre.capacite
8          elements = new T[capacite];
9      }
10     for(int i = 0; i < nbElements; i++)
11         elements[i] = autre.elements[i];
12     return *this;
13 }
```

Constructeur par copie d'un tableau

```
1  // passage par valeur
2  int fonction(Tableau<int> tab) {
3      int sommedouble=0;
4      for(int i = 0; i < tab.taille(); i++) {
5          tab[i] *= 2;
6          sommedouble += tab[i];
7      }
8      return sommedouble;
9  }
```

```
1  int main() {
2      Tableau<int> t;
3      t.ajouter(1);
4      t.ajouter(2);
5      t.ajouter(3);
6      cout << fonction(t) << endl;
7      cout << fonction(t) << endl;
8      return 0;
9  }
```

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

Constructeur par copie par défaut (par le compilateur)

```
1  template <class T>
2  Tableau<T>::Tableau(const Tableau<T>& autre)
3   : elements(autre.elements),
4     capacite(autre.capacite),
5     nbElements(autre.nbElements) {}
```

```
1  int fonction(Tableau<int> tab) { ... }
2
3  int main() {
4      Tableau<int> t;
5      t.ajouter(1);
6      t.ajouter(2);
7      t.ajouter(3);
8      cout << fonction(t) << endl;
9      cout << fonction(t) << endl;
10     return 0;
11 }
```

Bon code pour le constructeur par copie

```
1  template <class T>
2  Tableau<T>::Tableau(const Tableau& autre) {
3      capacite = autre.nbElements; // ou autre.capacite
4      nbElements = autre.nbElements;
5      elements = new T[capacite];
6      for(int i = 0; i < nbElements; i++)
7          elements[i] = autre.elements[i];
8  }
```

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://cria2.uqam.ca/INF3105/lab3/`