UE MOBJ [4L103]

Jean-Paul CHAPUT
Jean-Paul.Chaput@lip6.fr

SESI

2020-2021

Table des matières

Concepts Présentés au Quatrième Cours

1. Templates de fonctions et de classes.
2. Conteneurs STL.
3. Iterateurs STL.

Pas de notes pour ce transparent.

III Introduction aux `template`

- Générateurs de fonctions ou de classes, ayant des types pour paramètres.
- Mécanisme à la base de la STL.
- Template Meta-Programming (TMP).

- Le mécanisme des `template` ne permet pas seulement de créer des modèles de fonctions et de classes, c'est en fait un langage de programmation complet exécutable par le compilateur. D'où les TMP.
- Le compilateur peut totalement réorganiser le code généré par un template. En particulier supprimer des portions jamais appelées.
- En pratique, cela signifie que l'on peut faire exécuter tout ou partie d'un programme *par le compilateur lui-même*. D'où des temps d'exécution du programme compilé très bons.

III.1 Exemple & Syntaxe (1)

```
template<typename T>
  T tableMax ( T* table, int size ) {
    T max = table[0];
    for ( int i=1 ; i<size ; i++ ) {
      if (table[i] > max) max = table[i];
    }
    return max;
  }
```

- Le mot clé `class` prête à confusion, utiliser de préférence `typename`.
- Le paramètre formel `T` doit être utilisé comme si cela était un type.

III.1 Instanciation – Types Simples (POD)

```
int main ( int argc, char* argv[] ) {
  int  t1[4] = { 0, 1, 2, 3 };
  int  t2[2] = { 4, 5 };
  char s1[5] = "abcd";

  cout << tableMax<int >(t1,4) << endl;
  cout << tableMax<char>(s1,4) << endl;

  cout << tableMax(t2,2) << endl;
}
```

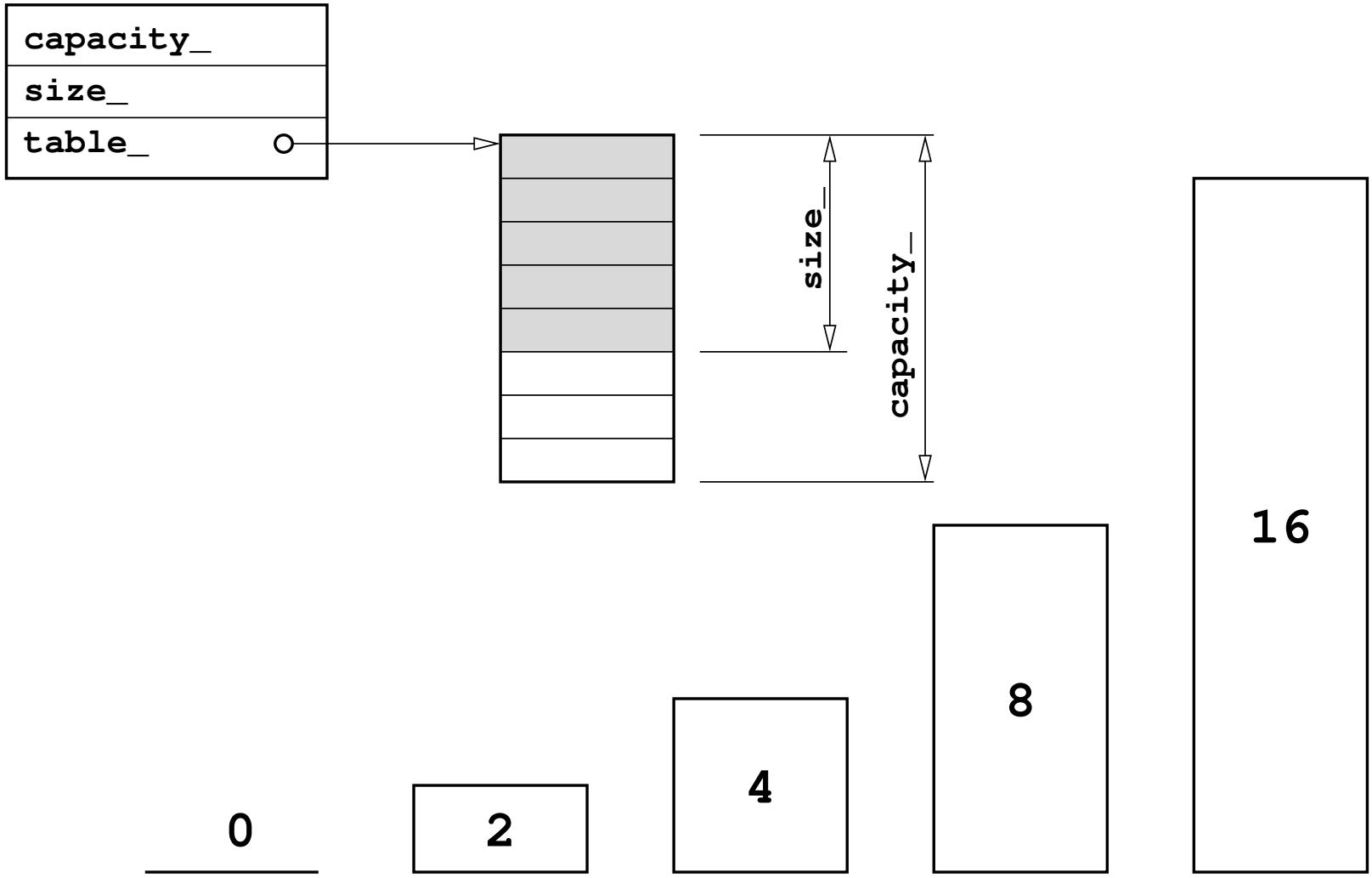
- Syntaxe analogue à un appel de fonction, sauf que le paramètre formel reçoit un type et non une valeur numérique ou un objet.
- La valeur du type `T` peut être fournie explicitement ou bien le compilateur peut la déduire du type des arguments passés à la fonction elle-même.

III.1 Instanciation – Types Classes

```
class Element {
  private:
    long  value_;
  friend bool operator> ( Element& lhs, Element& rhs );
};

bool operator> (Element& lhs, Element& rhs)
{ return lhs.value_ > rhs.value_; }

int main ( int argc, char* argv[] ) {
  Element t1[4] = { 0, 1, 2, 3 };

  cout << tableMax(t1,4) << endl;
}
```

- Les opérateurs ou fonctions membres auxquels on fait appel dans le `template` doivent exister dans le type concerné.
- Une fois instanciée, une classe template se comporte, et est utilisable exactement comme une classe ordinaire.
- Remarque, lorsqu'une fonction non-`template` de type compatible est disponible, celle-ci sera prioritaire. Elle sera considérée comme plus spécialisée.

III.2.1 Spécification de la classe `Vector`



Politique de réallocation du tableau

- Trois attributs:
  - `capacity_` : la taille *totale* du tableau alloué. Que les éléments soient utilisés ou non.
  - `size_` : le nombre d'éléments *effectivement* utilisés.
  - `table_` : un pointeur sur le tableau, alloué dynamiquement, contenant les éléments.
- Politique d'allocation mémoire du tableau: l'approche naïve consisterait à dimensionner le tableau de façon à ce qu'il contienne le nombre exact d'éléments. Ce qui obligerait à le réallouer à chaque insertion ou retrait. Ce serait extrêmement coûteux en temps, et aussi en mémoire, car on devrait gérer beaucoup d'objets de tailles disparates.

  Au lieu de cela, on choisit, à chaque fois que le tableau est plein, de le réallouer *en doublant* sa taille. On aura donc les tailles 0, 2, 4, 8, 16, 32, 64, … Cela correspond à une observation statistique. Si on passe de 8 à 9, il a beaucoup de chance qu'on arrive à 16. Bien sûr, les cas pathologiques sont toujours possibles.

  Une fois qu'elle a grandi, la `table_` ne rétrécit plus, même si tous les éléments en sont retirés. La mémoire ne sera rendue au système qu'à la destruction de l'objet.

III.2.1 Template de la classe Vector

```
template<typename T>
class Vector {
  private:
          T*          table_;
          size_t      size_;
          size_t      capacity_;
  private:
          void        resize_      ( size_t newcapacity );
  public:
                      Vector       ();
                      Vector       ( const Vector& );
                     ~Vector       ();
  public:
    inline size_t     size         ();
    inline size_t     capacity     ();
          void        reserve      ( size_t );
          void        push_back    ( T );
          void        pop_back     ();
          T&          back         ();
    const T&          back         () const;
          T&          operator[]   ( size_t );
};
```

- Méthodologie de développement : d'abord créer un objet ordinaire, le vérifier puis le transformer en `template`.
- Remplacement du type `POD int` par le paramètre formel de type `T`.
- `template` sert à déclarer un paramètre formel. Partout où l'on aura besoin d'un paramètre formel, il faudra utiliser cette déclaration.
- Ici encore, tout dans le `.h`.
- Le conteneur *possède* ses éléments. On a bien un tableau d'objets `T`, et la fonction `push_back()` prend bien un argument de type `T` en argument.
- Corollaire : le type `T` doit avoir un constructeur par copie.

III.2.2 Template des fonctions membres – 1

```
template<typename T>
Vector<T>::Vector () : table_    (NULL)
                     , size_     (0)
                     , capacity_(0)
{ }

template<typename T>
Vector<T>::~Vector ()
{ if (table_) delete [] table_; }

template<typename T>
T& Vector<T>::operator[] (size_t index)
{
  static T notFound;
  if (index < size_) return table_[index];
  return notFound;
}
```

- Noter l’endroit où s’applique le paramètre formel : à la classe, pas au membre.
- Problème du type de retour de l’opérateur d’accès indexé: référence sur un objet qui ne doit pas disparaître quand la fonction se termine. On utilise un objet statique.

III.2.2 Template des fonctions membres – 2

```
template<typename T>
void  Vector<T>::resize_ ( size_t newcapacity )
{
  if (newcapacity <= capacity_) {
    cerr << "[ERROR] Vector::resize_() cowardly refusing to shrink ("
         << capacity_ << " to " << newcapacity << ")" << endl;
    return;
  }

  T* newtable = new T [newcapacity];
  for ( size_t i=0 ; i<size_ ; ++i ) newtable[i] = table_[i];

  if (table_) delete [] table_;

  table_    = newtable;
  capacity_ = newcapacity;
}
```

Pas de notes pour ce transparent.

III.2.2 Template des fonctions membres – 3

```
template<typename T>
void  Vector<T>::push_back ( T element ) {
  if (size_ == capacity_) {
    size_t newcapacity = (capacity_)?(capacity_*2):2;
    resize_( capacity_ );
  }
  table_[ size_++ ] = element;
}

template<typename T>
void  Vector<T>::pop_back () { if (size_) --size_; }

template<typename T>
T&  Vector<T>::back () {
  static T notFound;
  return (size_) ? table_[size_-1] : notFound;
}
```

Pas de notes pour ce transparent.

III.2.3 Syntaxe des templates

```
#include "Vector.h"
void printVectorInt ( const Vector<int>& v ) {
  for ( size_t i=0 ; i<v.size() ; ++i )
    cout << "v[" << i << "] = " << v[i] << endl;
}

int main (int argc, char* argv[]) {
  Vector<int> v;
  for ( size_t i=0 ; i<10 ; ++i )
    v.push_back(i);
  printVectorInt( v );
  return 0;
}
```

- Faire le lien avec System C.
- `const Vector<int>` est un type. Paramétré, certes, mais c'est un type. Il peut donc apparaître partout où un type est légal. Déclaration de variables, `cast`.
- En première approche, on peut faire une analogie avec le préprocesseur et les `define`. Les `template` suppriment un autre pan d'utilisation du préprocesseur (l'autre: `const`).
- La compilation «expanse» les `template`.
- On peut aussi le comprendre comme une fonction dont les arguments sont des types et qui est exécutée par le compilateur (et pas à l'exécution du programme).

III.3 containers – Généralités

- Ensembles organisés d'objets.
- Chaque type de `conteneur` est optimisé pour un type d'usage (insertion/accès/effacement).
- Principaux conteneurs :
  `list`, `vector`, `stack`, `queue`, `map`
- Bien entendu, ce sont des `templates`.

Pas de notes pour ce transparent.

III.3 containers – Critères de Choix

- Temps d'insertion/d'accès/d'effacement.
- Comment évalue-t-on les temps : en fonction du nombre d'éléments de la collection.
- Les types de temps : linéaire, logarithmique, quadratique, exponentiels...
- On souhaite limiter cette croissance.
- Encombrement mémoire : le bon sens...

- Soit un conteneur ayant n éléments, les temps classiques seront:

Constant	$\mathcal{O}(1)$	Ne dépend pas du nombre d'éléments
Linéaire	$\mathcal{O}(n)$	
Logarithmique	$\mathcal{O}(\log n)$	
Linearithmique	$\mathcal{O}(n \log n)$	
quadratique	$\mathcal{O}(n^2)$	Mauvais
exponentiel	$\mathcal{O}(n!)$	Très Mauvais

- Pour un même conteneur, les différentes opérations pourront avoir des temps différents.

III.3 Typologie des containers

- ✏ `list` : insertion en tout endroit.
- ✏ `vector` : accès indexé rapide.
- ✏ `map`, `set` : recherche rapide d'un élément, avec ou sans clé.

- Les principaux conteneurs, ainsi que leurs points forts.
- Présenter succintement les structures de données sous-jacentes.
- La `map` est aussi appelé dictionnaire.
- Le `set` rapelle le concept mathématique d'ensemble.

III.3.1 Exemple : vector

```
#include <vector>
#include "Box.h"

int main ( int argc, char* argv[] ) {
  vector<Box> boxes;

  Box b1 ( "b1", 0, 0, 10, 10 );
  Box b2 ( "b2", 5, 5, 20, 20 );

  boxes.push_back( b1 );  // L'element est copie.
  boxes.push_back( b2 );
  for ( size_t i=0 ; i<boxes.size() ; ++i )
    cout << "boxes[" << i << "] = " << boxes[i] << endl;
}
```

- Les conteneurs «possèdent» leurs éléments.
- Les objets que gère le conteneur doivent avoir un constructeur par copie et supporter l'affectation (`operator=()`).
- Les objets dans le conteneur sont détruits à la destruction de celui-ci.
- **Conséquence** : dans la majorité des cas on utilise des pointeurs.

III.3.1 Exemple : vector

```
#include <vector>
#include "Box.h"

int main ( int argc, char* argv[] ) {
  vector<Box*> boxes;

  Box b1 ( "b1", 0, 0, 10, 10 );
  Box b2 ( "b2", 5, 5, 20, 20 );

  boxes.push_back( &b1 );  // L'element *n'est pas*
                           // copie.
  boxes.push_back( &b2 );
  for ( size_t i=0 ; i<boxes.size() ; ++i )
    cout << "boxes[" << i << "] = " << *boxes[i] << endl;
}
```

- Version «pointeur» du code précédent.
- Attention à la désallocation de b1 et b2, elle ne doit pas intervenir avant que le vecteur ne soit lui-même détruit.

III.3.2 Fonctions Membres Communes

```
  bool     empty       ();
  size_t   size        ();
  void     resize      ( size_t size );
  size_t   capacity    ();
  size_t   max_size    ()
  void     clear       ();
  T&       front       ();
  T&       back        ();
  void     push_back   ( const T& element );
  void     pop_back    ();
// Pas dans <vector>.
  void     push_front  ( const T& element );
  void     pop_front   ();
```

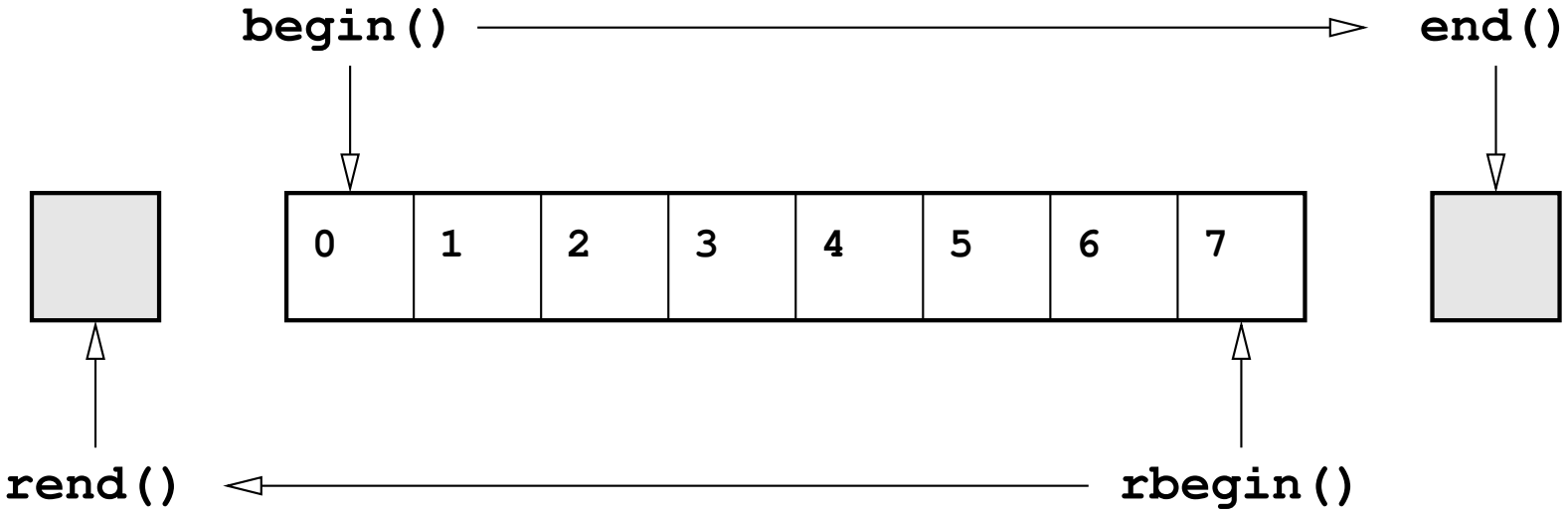
Pas de notes pour ce transparent.

III.4 Itérateurs

```
int    i      = 0;
char* table = new char [10];
for ( char* p=table ; p!=table+10 ; ++p ) // Remplissage
  (*p) = '0'+i;
for ( char* p=table ; p!=table+10 ; ++p ) // Affichage
  std::cout << *p;
std::cout << std::endl

// Le meme code, avec des iterateurs.
vector<char> v;
for ( int j=0 ; j<10 ; ++j )                  // Remplissage
  v.push_back('0'+j);
vector<char>::iterator iv = v.begin();
for ( ; iv != v.end() ; ++iv )                // Affichage
  std::cout << (*iv);
std::cout << std::endl
```

- Remarque: rappel de l'arithmétique des pointeurs...
- Analogie avec les pointeurs, **mais** le conteneur comme ses itérateurs sont des objets.
- Accès uniforme aux éléments de tous les types de conteneurs.
- iterator est un sous-type défini dans chaque conteneur, d'où la syntaxe avec opérateur de résolution de portée.
- `begin()` et `end()` sont des fonctions membres de chaque conteneur qui renvoient un itérateur sur le premier élément et sur un élément virtuel situé au-delà du dernier élément réel.
- Nécessité du end(), dérouler la boucle.
- Accès à l'élément pointé. Forme théorique (accès par indirection) forme sûre (déréférence).
- Les itérateurs font un usage intensif des opérateurs.
- Nous allons pouvoir introduire les fonctions manquantes des conteneurs.

III.4 Itérateurs `begin()` & `end()`



- Les itérateurs `end()` et `rend()` renvoient sur des éléments fictifs situés *après* le dernier élément réel ou *avant* le premier.
- Ils sont utiles pour initialiser les itérateurs ou faire des comparaisons, mais on ne doit **jamais** essayer d'accéder à l'élément lui-même (cela finira très probablement en *core dump*).

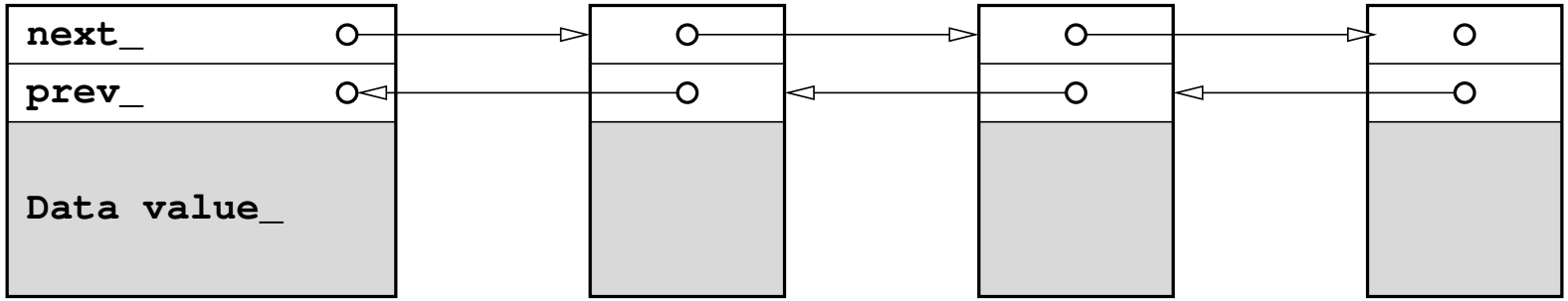
III.4.1 Fonctions des Conteneurs (itérateurs)

```
vector<char>::iterator  beg = v.begin();
vector<char>::iterator  end = v.end();
vector<char>::iterator  pos = v.insert(beg,'R');
vector<char>::iterator  pos = v.erase (beg);

// Parcours inverse.
vector<char>::reverse_iterator iv = v.rbegin();
for ( ; iv != v.rend() ; ++iv )
  std::cout << (*iv);
std::cout << std::endl
```

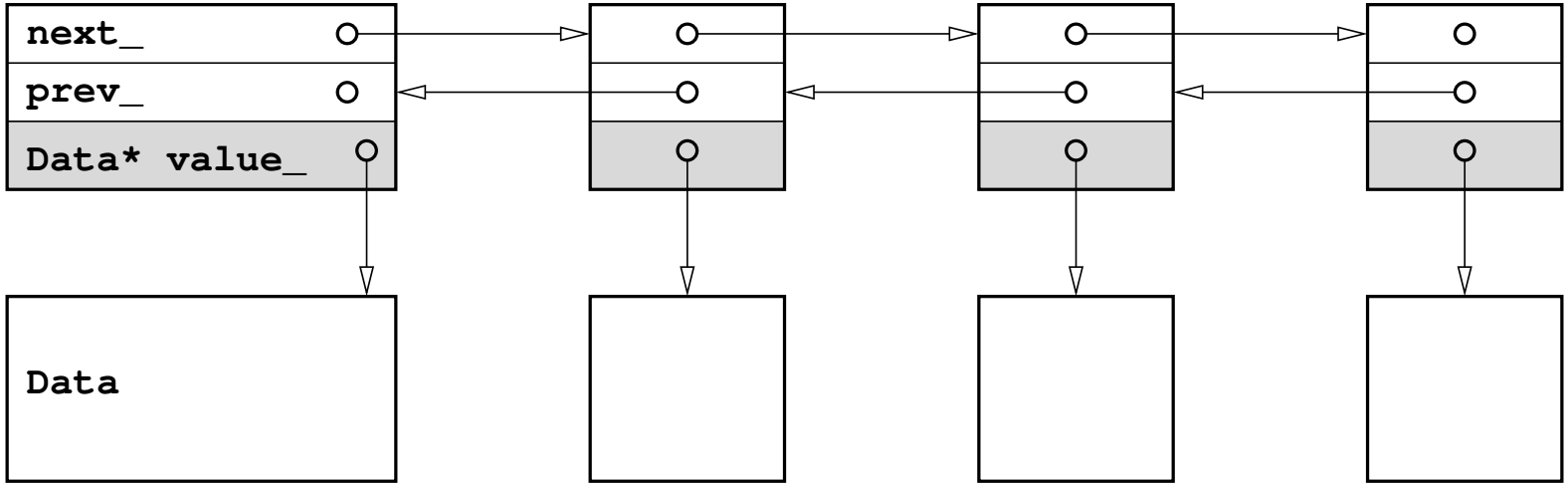
- Parcours inverse disponible avec reverse_iterator.
- insert() insère avant l’itérateur.
- erase() efface à la position demandée.
- Problème de l’invalidation des itérateurs.

III.4.2 Conteneur `list`

`list<Data> dataList;`

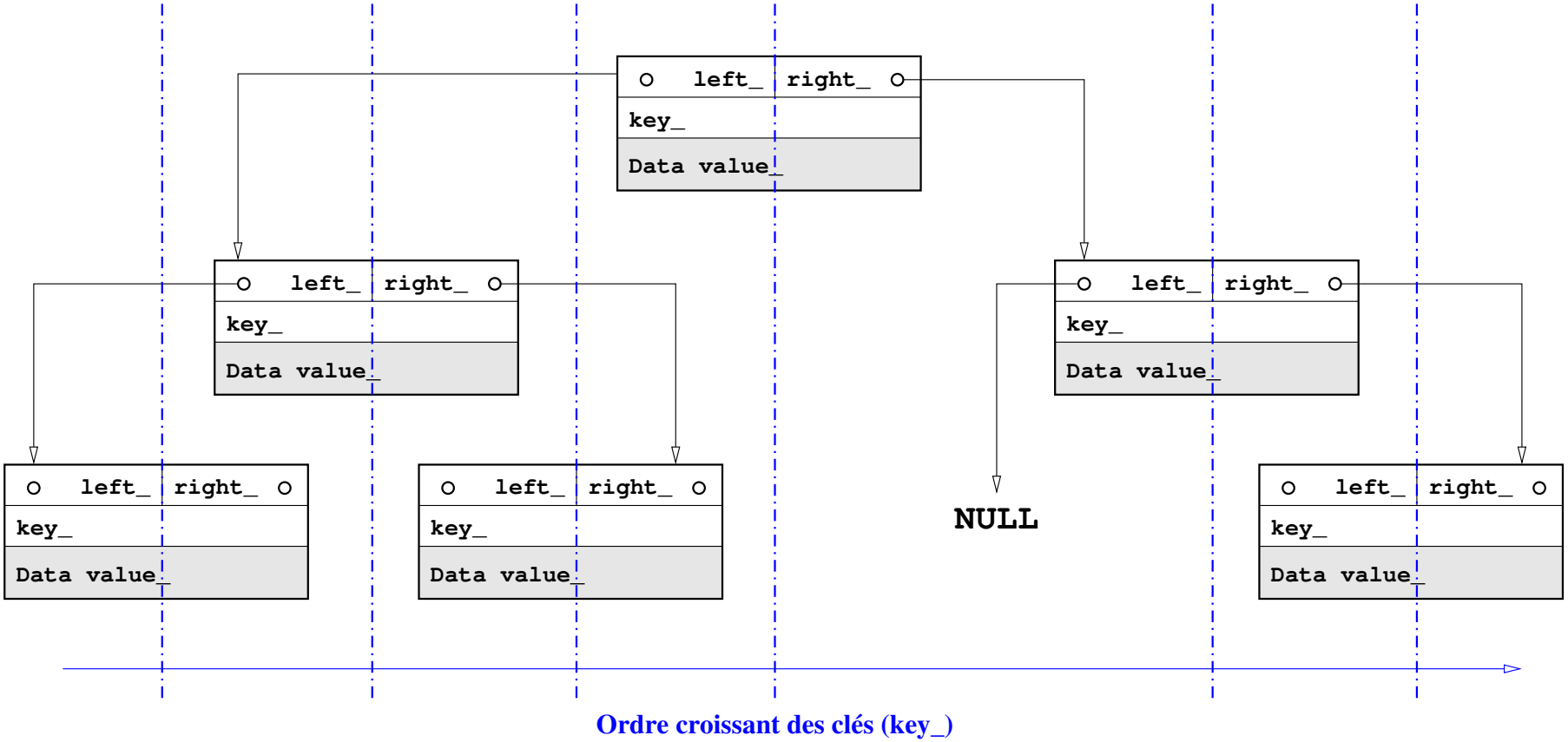


`list<Data*> dataList;`



- Il s’agit de la liste doublement chaînée.
- Insertion et retrait à n’importe quel point de la liste en temps linéaire.
- Empreinte mémoire nettement supérieure à celle du tableau (`vector`) : deux pointeurs *par élément*.
- Les itérateurs restent valides en cas de modifications.

III.4.3 Conteneur `map`



- Accès, insertion et effacement en temps logarithmique.
- Tri sur la clé. Dans un `vector` la clé ne peut être qu'un entier. Dans une `map`, le type de la clé peut être quelquonque, avec la contrainte qu'elle doit posséder une fonction d'ordre.
- Un élément d'une `map` est donc un couple (clé,valeur). C'est ce que renvoie l'itérateur.
- Implémentation classique en utilisant des arbres binaires rouge/noir (red/black tree). Ce n'est pas une obligation. Organisation de l'arbre pour un choix dichotomique rapide. L'arbre est réorganisé à chaque modification pour rester de profondeur minimale.

III.4.4 Map et Itérateurs

```
std::map<std::string,Box> m;
m["machin"] = Box(0,0,1,1);
m["bidule"] = Box(0,0,2,2);
m["truc"  ] = Box(0,0,3,3);
std::map<std::string,Box>::iterator im = m.find("truc");
if (im != m.end()) {
  std::cout << "Key:"   << (*im).first
            << "␣value" << (*im).second << endl;
  m.erase(im);
}
for ( im = m.first() ; im != m.end() ; ++im )
  std::cout << "Key:"   << (*im).first
            << "␣value" << (*im).second << endl;
```

- Lorsque l'on utilise l'opérateur d'accès indexé, l'élément d'index demandé est *automatiquement* créé, même s'il n'existait pas auparavant.
- La valeur pointée par l'itérateur de map est une paire clé/ valeur. L'accès par itérateur est donc rendu un peu plus compliqué.
- Itérateurs non-invalidés en cas d'insertion/effacement.

III.4.5 Itérateurs et const

```
void myFind ( const std::map<std::string,Box>& m ) {
  std::map<std::string,Box>::const_iterator im = m.find("truc");
  if (im != m.end())
    std::cout << "Key:"   << (*im).first
              << "␣value" << (*im).second << endl;
  else
    std::cout << "Not␣found" << std::endl;
}

std::map<std::string,Box> m;
m["machin"] = Box(0,0,1,1);
m["bidule"] = Box(0,0,2,2);
m["truc"  ] = Box(0,0,3,3);
myFind ( m );
```

- Pour utiliser des itérateurs sur un conteneur const, utiliser const_iterator.

III.4.6 Algorithme – Tri

```
class CompareByY2 {
  public:
    bool operator() ( const Box& lhs, const Box& rhs )
    { return lhs.getY2() < rhs.getY2(); }
};


std::vector<Box> v;
v.push_back( Box(0,0,3,3) );
v.push_back( Box(0,0,2,2) );
v.push_back( Box(0,0,1,1) );

CompareByY2 cmp;       // cmp est un *objet* ...
if ( cmp(v[0],v[1]) ) // qui peut etre appele comme une *fonction*.
  cout << "v[0] est inferieur a v[1]" << endl;

sort( v.begin(), v.end(), CompareByY2() );
```

- Bien distinguer l'instance de la classe `CompareByY2`, et son appel «comme si c'était» une fonction (opérateur fonctionnel).
- Relation d'ordre faible (strict weak ordering).
- Relation `lhs < rhs`.
- Le tri est lent. Map triée par construction.