

# **INF1010**

## **Programmation orientée objet**

Introduction à la STL  
Standard Template Library

# À quoi sert la STL ?

---

- Multiple types de données en C++
  - Copier/coller des bouts de code pour toujours faire les mêmes actions ?
- Non ! On a les classes génériques !
  - Oui, mais comment faire pour que des types répandus soit réutilisés à travers les programmes ?
- La STL = réutiliser encore et encore

# Éléments de la STL

---

- Les conteneurs (ou *containers*)
  - Pour stocker des objets
- Les itérateurs (ou *iterators*)
  - Pour parcourir les conteneurs
- Les algorithmes, foncteurs et introducteurs
  - Pour ordonner, chercher, manipuler les conteneurs en utilisant des itérateurs

# Prenons un exemple...

## Version 1: C++ standard

---

```
#include <cstdlib> // qsort
#include <iostream> // cin, cout, ..
using namespace std;

int compare_int (const void *a, const void *b) {
    int aa = *(int *)a;
    int bb = *(int *)b;
    return (aa < bb) ? -1 : (aa > bb) ? 1 : 0;
}

int main (int argc, char *argv[]) {
    int size;
    cout << "Nombre d'entiers? " << endl;
    cin >> size;

    int *array = new int[size];
    for (int i = 0; i < size; i++) {
        cout << "Valeur de l'entier " << i
              << " de " << size << "?" << endl;
        cin >> array[i];
    }
}
```

```
// Ordonner les éléments
qsort(array, size, sizeof(int), compare_int);

cout << "Entiers ordonnés:" << endl;
for (int i = 0; i < size; i++)
    cout << array[i] << endl;

// Afficher juste ceux entre 10 et 30
cout << "Entiers avec une valeur entre 10 et 30:"
      << endl;
for (int i = 0; i < size; i++) {
    if (array[i] >= 10 && array[i] <= 30)
        cout << array[i] << endl;
}

delete [] array;

return 0;
```

# Prenons un exemple...

## Version 1: C++ standard

```
#include <cstdlib> // qsort
#include <iostream> // cin, cout, ..
using namespace std;

int compare_int (const void *a, const void *b) {
    int aa = *(int *)a;
    int bb = *(int *)b;
    return (aa < bb) ? -1 : (aa > bb) ? 1 : 0;
}

int main (int argc, char *argv[]) {
    int size;
    cout << "Nombre d'entiers? " << endl;
    cin >> size;

    int *array = new int[size];
    for (int i = 0; i < size; i++) {
        cout << "Valeur de l'entier " << i
             << " de " << size << "?" << endl;
        cin >> array[i];
    }
```

```
// Ordonner les éléments
qsort(array, size, sizeof(int), compare_int);

cout << "Entiers ordonnés:" << endl;
for (int i = 0; i < size; i++)
    cout << array[i] << endl;

// Afficher juste ceux entre 10 et 30
cout << "Entiers avec une valeur entre 10 et 30:"
     << endl;
for (int i = 0; i < size; i++) {
    if (array[i] >= 10 && array[i] <= 30)
        cout << array[i] << endl;
}

delete [] array;

return 0;
```

Ce code présente un ensemble d'opérations communément réalisées, comme **lire des données depuis l'entrée standard** ou **afficher/trier/filtrer le contenu d'un tableau**

La **STL**, via ses classes et fonctions génériques, peut ici simplifier l'utilisation de structures de données, éliminer les boucles et ainsi réduire le nombre de lignes et augmenter la clarté du code.

# Comment améliorer notre code?

## De la version 1 à la version 2

---

- Utiliser un conteneur de la STL
  - On verra plus en détail les différents types de conteneurs, mais on en connaît déjà deux: **vector** et **list**

Avant:

```
int *array = new int[size];
for (int i = 0; i < size; i++) {
    cout << "Valeur de l'entier " << i
         << " de " << size << "?" << endl;
    cin >> array[i];
}
```

Après:

```
vector<int> array;
int input;
for (int i = 0; i < size; i++) {
    cout << "Valeur de l'entier " << i
         << " de " << size << "?" << endl;
    cin >> input;
    array.push_back(input);
}
```

- **vector** est un conteneur... on peut faire mieux!

# Comment améliorer notre code?

## De la version 1 à la version 2

---

- En fait, on peut même se passer de demander le nombre de valeurs que l'on veut entrer!

Avant:

```
int size;
cout << "Nombre d'entiers? " << endl;
cin >> size;

int *array = new int[size];
for (int i = 0; i < size; i++) {
    cout << "Valeur de l'entier " << i
        << " de " << size << "?" << endl;
    cin >> array[i];
}
```

Après:

```
vector<int> array;
int input;

cout << "Utiliser Ctrl+D (Linux) ou"
    << "Ctrl+Z (Windows) pour finir"
    << endl;
while (cin >> input)
    array.push_back(input);
```

- Mais **qsort** ne marche pas avec un conteneur!

# Comment améliorer notre code?

## De la version 1 à la version 2

---

- Remplacer **qsort** par **sort**
  - `void qsort (void* base, size_t num, size_t size, int (*compar)(const void*,const void*));`
  - `template <class RandomAccessIterator>  
void sort (RandomAccessIterator first,  
RandomAccessIterator last);`
- Ok... Mais c'est quoi un objet “RandomAccessIterator” ?!





# Parenthèse: Qu'est-ce qu'un itérateur ?

---

- C'est un moyen de spécifier une position dans un conteneur
- Il peut être:
  - Incrémenté (*passage à l'objet suivant*)
  - Déréférencé (*comme pour un pointeur: on récupère l'objet pointé*)
  - Comparé avec un autre itérateur (*pointent-ils vers le même objet du même conteneur?*)
- C'est une généralisation des pointeurs

# Parenthèse: Qu'est-ce qu'un itérateur ?

---

- Deux itérateurs peuvent servir à définir un intervalle dans un conteneur. L'intervalle complet d'un conteneur est défini par:
    - Un itérateur qui pointe sur le premier objet
    - Un itérateur qui pointe sur la valeur spéciale “past-the-end”
- Pourquoi “past-the-end” ?

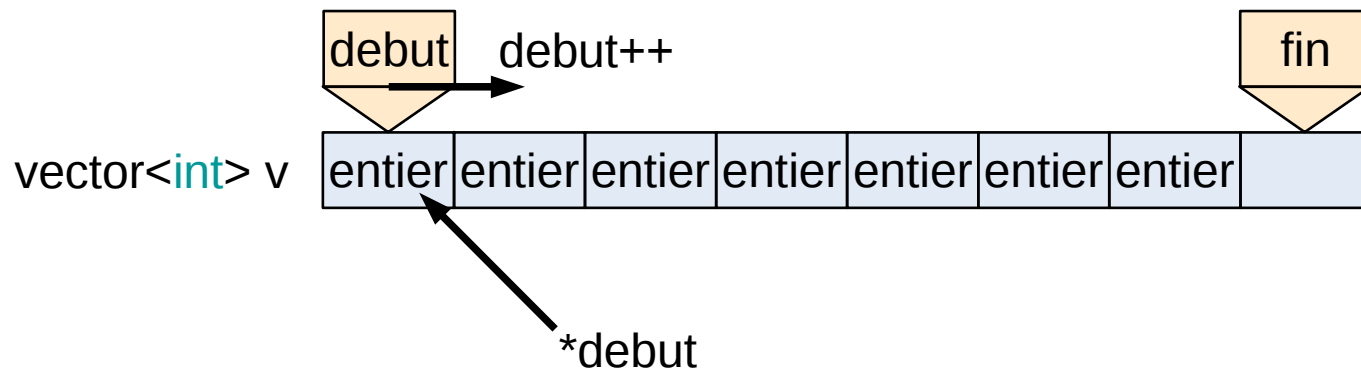
  - Pour savoir qu'on a fait le tour des valeurs
  - Pour pouvoir définir un intervalle vide
- Ces deux itérateurs peuvent être récupérés par les méthodes “begin()” et “end()” d'un conteneur, respectivement

# Parenthèse: Qu'est-ce qu'un itérateur ?

- Sur un vecteur d'entiers par exemple:

```
vector<int> v
// Ajouter des entiers avec push_back par exemple
vector<int>::iterator debut = v.begin();
vector<int>::iterator fin = v.end();
```

- Représentation graphique:



# Parenthèse: Qu'est-ce qu'un itérateur ?

---

- Avec **sort**, on a besoin de spécifier l'intervalle de travail, on aura donc besoin de ces deux itérateurs.
- On a donc plus qu'à changer **qsort** par l'appel à **sort** sur l'intervalle complet de notre nouveau vecteur:

```
qsort(array, size, sizeof(int), compare_int);
```

devient:

```
sort(array.begin(), array.end());
```

# Prenons un exemple...

## Version 2: conteneur, sort

```
#include <cstdlib> // qsort
#include <vector> // vector
#include <algorithm> // sort
#include <iostream> // cin, cout, ..
using namespace std;

int compare_int (const void *a, const void *b) {
    int aa = *(int *)a;
    int bb = *(int *)b;
    return (aa < bb) ? -1 : (aa > bb) ? 1 : 0;
}

int main (int argc, char *argv[]) {
    int size;
    cout << "Nombre d'entiers? " << endl;
    cin >> size;

    int *array = new int[size];
    vector<int> array;
    int input;
    for (int i = 0; i < size; i++) {
        cout << "Valeur de l'entier " << i
              << " de " << size << " ? " << endl;
        cin >> array[i];
    }
}
```

```
cout << "Utiliser Ctrl+D (Linux) ou "
      << "Ctrl+Z (Windows) pour finir"
      << endl;
while (cin >> input)
    array.push_back(input);

// Ordonner les éléments
qsort(array, size, sizeof(int), compare_int);
sort(array.begin(), array.end());

cout << "Entiers ordonnés:" << endl;
for (int i = 0; i < sizearray.size(); i++)
    cout << array[i] << endl;

// Afficher juste ceux entre 10 et 30
cout << "Entiers avec une valeur entre 10 et 30:"
      << endl;
for (int i = 0; i < sizearray.size(); i++) {
    if (array[i] >= 10 && array[i] <= 30)
        cout << array[i] << endl;
}

delete [] array;

return 0;
```

# Prenons un exemple...

## Version 2: conteneur, sort

---

```
#include <vector>    // vector
#include <algorithm> // sort
#include <iostream>  // cin, cout, ..
using namespace std;

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

    cout << "Utiliser Ctrl+D (Linux) ou "
         << "Ctrl+Z (Windows) pour finir"
         << endl;
    while (cin >> input)
        array.push_back(input);
}
```

```
// Ordonner les éléments
sort(array.begin(), array.end());

cout << "Entiers ordonnés:" << endl;
for (int i = 0; i < array.size(); i++)
    cout << array[i] << endl;

// Afficher juste ceux entre 10 et 30
cout << "Entiers avec une valeur entre 10 et 30:"
     << endl;
for (int i = 0; i < array.size(); i++) {
    if (array[i] >= 10 && array[i] <= 30)
        cout << array[i] << endl;
}

return 0;
```

# Améliorer *encore* notre code?

## De la version 2 à la version 3

---

- On a vu à quoi servent les itérateurs, est-ce qu'on peut les utiliser ailleurs?
  - On peut les utiliser pour l'affichage!
- On a juste à modifier nos boucles **for** pour les utiliser:

```
vector<int>::iterator end = array.end();  
for (vector<int>::iterator it = array.begin(); it != end; it++)  
    cout << *it << endl;
```

- Question: pourquoi pas “`it < end`” ?!

# Prenons un exemple...

## Version 3: boucles d'itérateurs

```
#include <vector>    // vector
#include <algorithm> // sort
#include <iostream>  // cin, cout, ..
using namespace std;

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

    cout << "Utiliser Ctrl+D (Linux) ou "
          << "Ctrl+Z (Windows) pour finir"
          << endl;
    while (cin >> input)
        array.push_back(input);
}
```

```
// Ordonner les éléments
sort(array.begin(), array.end());

cout << "Entiers ordonnés:" << endl;
for (int i = 0; i < array.size(); i++)
    cout << array[i] << endl;
vector<int>::iterator end = array.end();
for (vector<int>::iterator it = array.begin();
     it != end; it++)
    cout << *it << endl;

// Afficher juste ceux entre 10 et 30
cout << "Entiers avec une valeur entre 10 et 30:"
      << endl;
for (int i = 0; i < array.size(); i++) {
    if (array[i] >= 10 && array[i] <= 30)
        cout << array[i] << endl;
for (vector<int>::iterator it = array.begin();
     it != end; it++) {
    if (*it >= 10 && *it <= 30)
        cout << *it << endl;
}

return 0;
```



# Prenons un exemple...

## Version 3: boucles d'itérateurs

---

```
#include <vector>    // vector
#include <algorithm> // sort
#include <iostream>  // cin, cout, ..
using namespace std;

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

    cout << "Utiliser Ctrl+D (Linux) ou "
         << "Ctrl+Z (Windows) pour finir"
         << endl;
    while (cin >> input)
        array.push_back(input);
}
```

```
// Ordonner les éléments
sort(array.begin(), array.end());

cout << "Entiers ordonnés:" << endl;
vector<int>::iterator end = array.end();
for (vector<int>::iterator it = array.begin();
     it != end; it++)
    cout << *it << endl;

// Afficher juste ceux entre 10 et 30
cout << "Entiers avec une valeur entre 10 et 30:"
     << endl;
for (vector<int>::iterator it = array.begin();
     it != end; it++) {
    if (*it >= 10 && *it <= 30)
        cout << *it << endl;
}

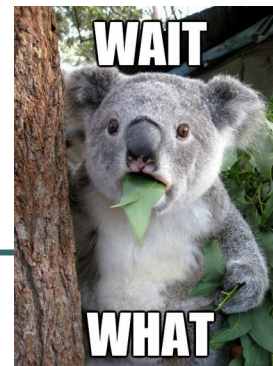
return 0;
```

# Améliorer *toujours* notre code?

## De la version 3 à la version 4

---

- On utilise *cin* et *cout* sur les éléments de notre vecteur, comment simplifier leur utilisation?
  - Un autre algorithme de la STL: **copy**
- **copy** permet de copier d'un conteneur à un autre... mais comment pourrait-on l'utiliser pour remplacer nos opérations sur les flux?
- La solution: les adaptateurs d'itérateurs
  - Mais... Qu'est-ce?!



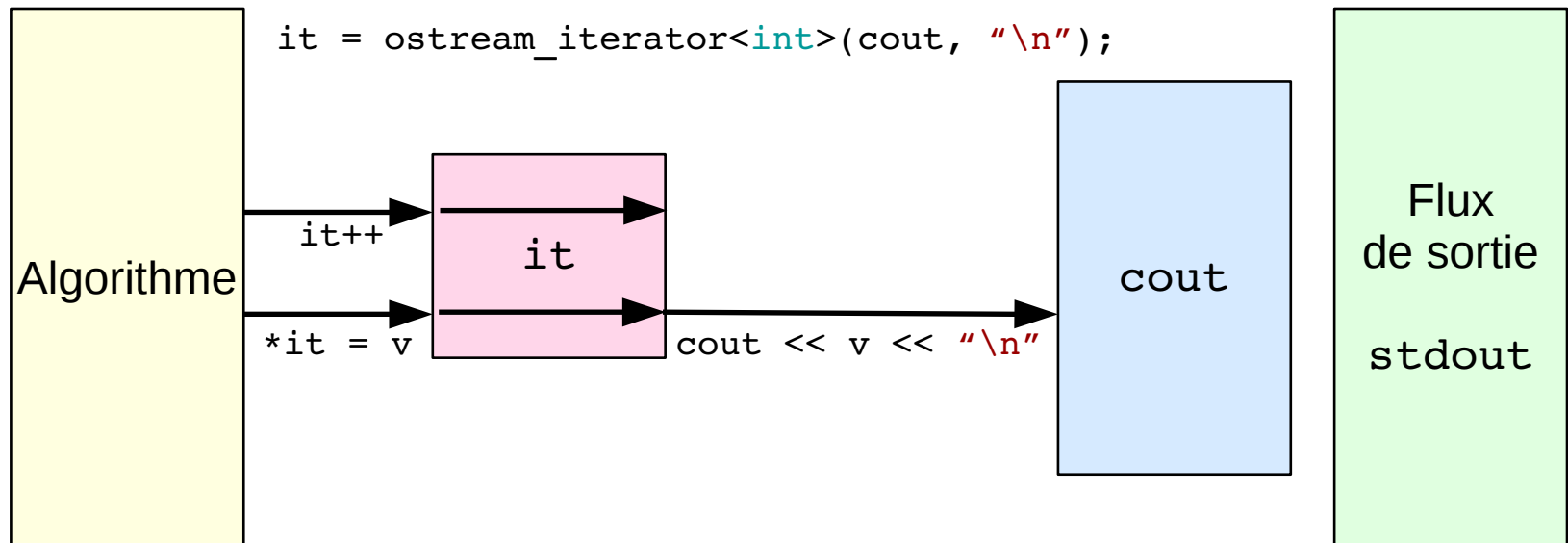
# Parenthèse: les adaptateurs d'itérateurs de flux

---

- Un flux comme *cout* propose les bonnes **fonctionnalités** pour un itérateur de sortie (itérateur qui sert à écrire), puisqu'on peut lui envoyer une séquence d'éléments
- *cout* n'a par contre pas la bonne **interface**: une opération qui utilise un itérateur s'attend à ce qu'il puisse être **incrémenté** et **déréférencé**.
- La STL propose ainsi des adaptateurs d'itérateurs, qui transforment l'interface d'autres types en interface d'itérateur

# Parenthèse: les adaptateurs d'itérateurs de flux

- Les adaptateurs d'itérateurs de flux fonctionnent un peu comme un adaptateur de prise électrique lorsqu'on voyage.
- Représentation avec `ostream_iterator` et `cout` pour nos entiers:



# Améliorer *toujours* notre code?

## De la version 3 à la version 4

---

- Donc, avec `ostream_iterator` et **copy**, on peut remplacer la boucle **for** servant à l'affichage de tous les éléments.
- **copy** prend trois paramètres: l'intervalle à copier (itérateur de début, itérateur de fin) et l'itérateur de début où copier.
- Pour notre boucle, on aura donc:

```
copy(array.begin(), array.end(),  
      ostream_iterator<int>(cout, "\n"));
```

# Prenons un exemple...

## Version 4: adaptateurs d'itérateurs

```
#include <vector>    // vector
#include <algorithm> // sort
#include <iostream>  // cin, cout, ..
#include <iterator>  // *_iterator
using namespace std;

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

    cout << "Utiliser Ctrl+D (Linux) ou "
         << "Ctrl+Z (Windows) pour finir"
         << endl;
    while (cin >> input)
        array.push_back(input);
}
```

```
// Ordonner les éléments
sort(array.begin(), array.end());

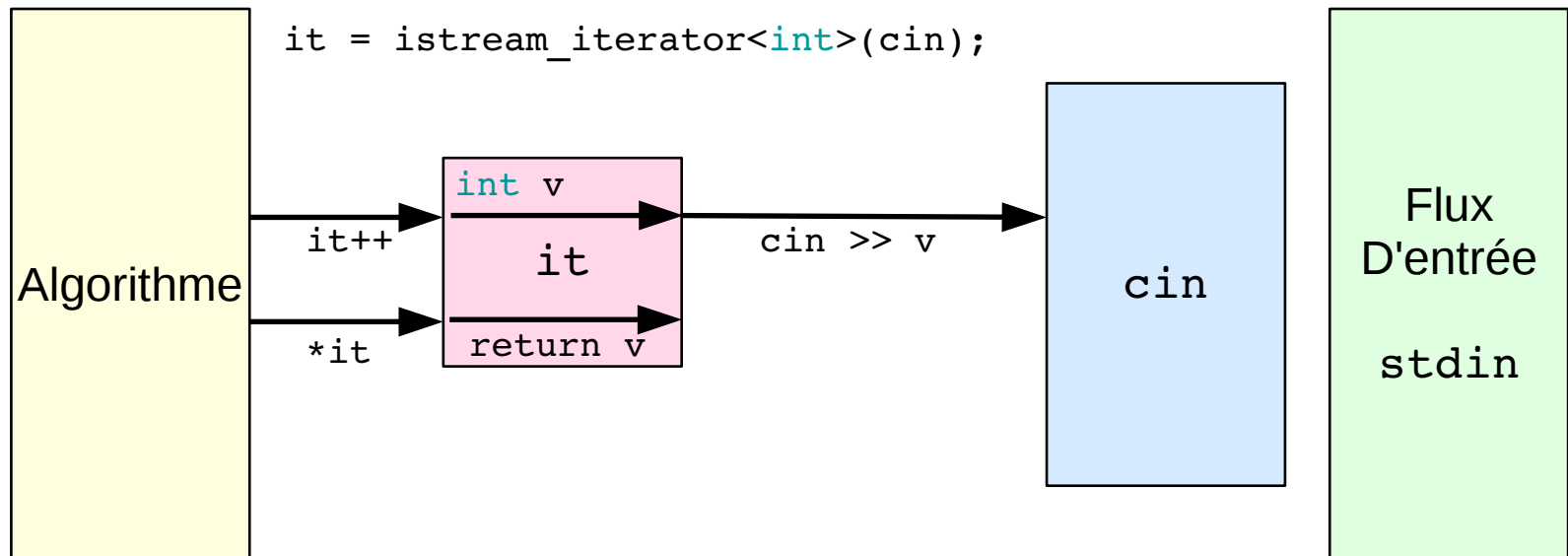
cout << "Entiers ordonnés:" << endl;
vector<int>::iterator end = array.end();
for (vector<int>::iterator it = array.begin();
    it != end; it++)
    cout << *it << endl;
    copy(array.begin(), end,
         ostream_iterator<int>(cout, "\n"));

// Afficher juste ceux entre 10 et 30
cout << "Entiers avec une valeur entre 10 et 30:"
     << endl;
for (vector<int>::iterator it = array.begin();
    it != end; it++) {
    if (*it >= 10 && *it <= 30)
        cout << *it << endl;
}

return 0;
```

# Parenthèse: les adaptateurs d'itérateurs de flux

- Pour `cin`, ça fonctionne de la même façon!
- Représentation avec `istream_iterator` et `cin` pour nos entiers :



# Améliorer *toujours* notre code?

## De la version 3 à la version 4

---

- Ok... mais on a dit que pour utiliser **copy** on a besoin de deux itérateurs pour l'intervalle à copier, et on ne connaît pas l'itérateur de fin du flux d'entrée?!
  - Oui! Et `istream_iterator` est bien fait, puisque avec son constructeur par défaut, on obtient un itérateur avec la valeur “past-the-end” !

```
istream_iterator<int> end; // itérateur pointant sur la valeur
                          // “past-the-end” que prendra un
                          // autre itérateur lorsqu'il
                          // atteindra la fin de son flux
                          // d'entrée
```



# Améliorer *toujours* notre code?

## De la version 3 à la version 4

---

- Donc... on a notre intervalle à copier... est-ce qu'on peut juste copier dans notre vecteur de la façon suivante ?

```
copy(istream_iterator<int>(cin),  
     istream_iterator<int>(),  
     array.begin());
```

# Améliorer *toujours* notre code?

## De la version 3 à la version 4

---

- En fait... oui et non!
  - **Oui**, si je connais la taille: on demande à nouveau la taille au préalable, mais il pourrait y avoir des erreurs si l'utilisateur ne s'arrête pas

```
int size;
cout << "Nombre d'entiers? " << endl;
cin >> size;

vector<int> array(size); // initialisation!!
copy(istream_iterator<int>(cin),
      istream_iterator<int>(),
      array.begin());
```

- **Non**, si je veux éviter les potentielles erreurs (du type *segmentation fault*). Comment faire alors?!

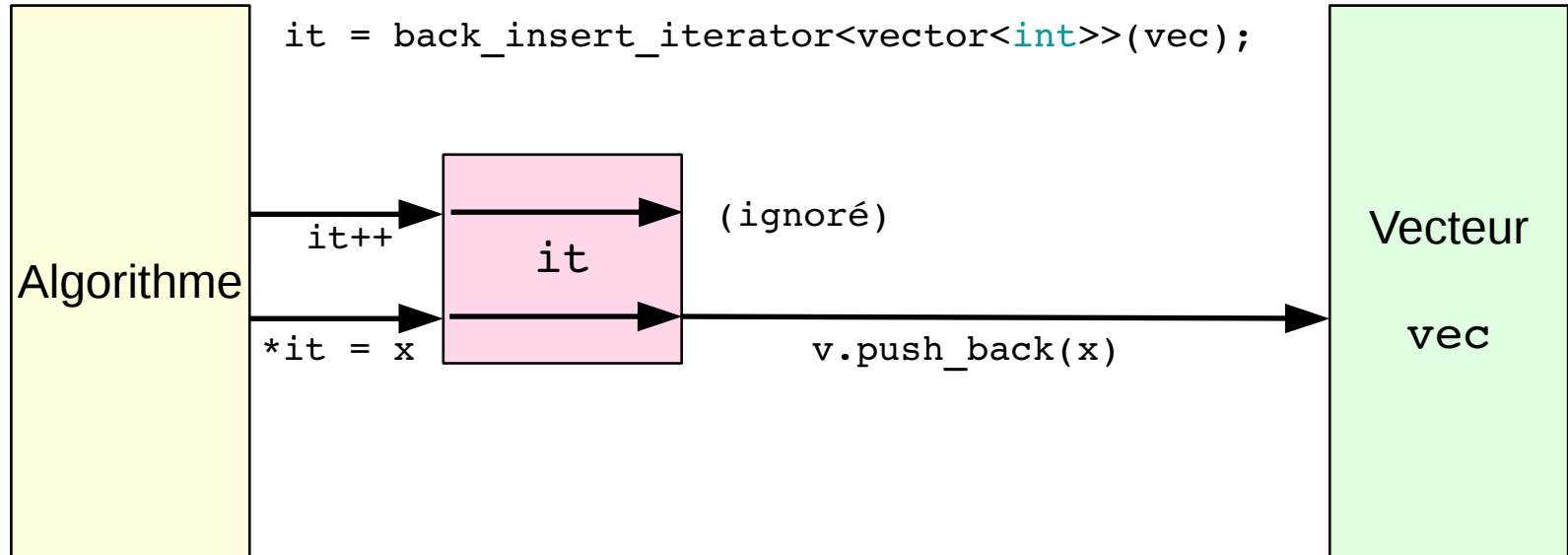
# Parenthèse: les adaptateurs d'itérateurs pour l'insertion

---

- On a vu que les adaptateurs d'itérateurs servent à “transformer l'interface”
- Ici, on veut pouvoir insérer des objets comme avec **push\_back**, mais via un itérateur! Solution? Les introducteurs!
  - `template <class Container>`  
`class back_insert_iterator;`
  - `vector<int> v;`  
`back_insert_iterator<vector<int>> dest(v);`

# Parenthèse: les adaptateurs d'itérateurs pour l'insertion

- Représentation de l'utilisation d'un `back_insert_iterator` pour un vecteur d'entiers `vec` :



# Parenthèse: les adaptateurs d'itérateurs pour l'insertion

---

- La STL fournit même une fonction “wrapper” `back_inserter` qui se charge de retourner le bon `back_insert_iterator`:
  - `vector<int> v;`  
`copy(..., ..., back_inserter(v));`
- Et ça marche aussi pour des **push\_front** (`front_insert_iterator` **et** `front_inserter`) ou encore des **insert** (`insert_iterator` **et** `inserter`)
- Pour **insert**, on ajoutera en deuxième paramètre un itérateur vers la position initiale où faire l'insertion:
  - `copy(..., ..., inserter(v, it));`

# Améliorer *toujours* notre code?

## De la version 3 à la version 4

---

- Et donc, ça donne quoi pour notre insertion en utilisant **copy**?

```
copy(istream_iterator<int>(cin),  
     istream_iterator<int>(),  
     back_inserter(array));
```

# Prenons un exemple...

## Version 4: adaptateurs d'itérateurs

```
#include <vector>    // vector
#include <algorithm> // sort
#include <iostream>  // cin, cout, ..
#include <iterator>  // *_iterator, *_inserter
using namespace std;

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

    cout << "Utiliser Ctrl+D (Linux) ou "
          << "Ctrl+Z (Windows) pour finir"
          << endl;
    while (cin >> input)
        array.push_back(input);

    copy(istream_iterator<int>(cin),
         istream_iterator<int>(),
         back_inserter(array));

}
```

```
// Ordonner les éléments
sort(array.begin(), array.end());

cout << "Entiers ordonnés:" << endl;
vector<int>::iterator end = array.end();
for (vector<int>::iterator it = array.begin();
     it != end; it++)
    cout << *it << endl;
copy(array.begin(), end,
     ostream_iterator<int>(cout, "\n"));

// Afficher juste ceux entre 10 et 30
cout << "Entiers avec une valeur entre 10 et 30:"
      << endl;
for (vector<int>::iterator it = array.begin();
     it != end; it++) {
    if (*it >= 10 && *it <= 30)
        cout << *it << endl;
}

return 0;
```

# Prenons un exemple...

## Version 4: adaptateurs d'itérateurs

---

```
#include <vector>    // vector
#include <algorithm> // sort
#include <iostream>  // cin, cout, ..
#include <iterator>  // *_iterator, *_inserter
using namespace std;
```

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

    cout << "Utiliser Ctrl+D (Linux) ou "
          << "Ctrl+Z (Windows) pour finir"
          << endl;
    copy(istream_iterator<int>(cin),
         istream_iterator<int>(),
         back_inserter(array));
```

```
    // Ordonner les éléments
    sort(array.begin(), array.end());
```

```
    cout << "Entiers ordonnés:" << endl;
    vector<int>::iterator end = array.end();
    copy(array.begin(), end,
          ostream_iterator<int>(cout, "\n"));
```

```
    // Afficher juste ceux entre 10 et 30
    cout << "Entiers avec une valeur entre 10 et 30:"
          << endl;
    for (vector<int>::iterator it = array.begin();
         it != end; it++) {
        if (*it >= 10 && *it <= 30)
            cout << *it << endl;
    }
```

```
    return 0;
```

```
}
```



# Améliorer *encore plus* notre code?

## De la version 4 à la version 5

---

- Il reste une partie du code que nous n'avons pas encore beaucoup améliorée...

```
for (vector<int>::iterator it = array.begin();  
     it != end; it++) {  
    if (*it >= 10 && *it <= 30)  
        cout << *it << endl;  
}
```

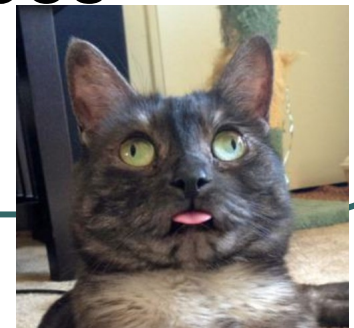
- On a vu comment utiliser l'algorithme **copy**... serait-il possible de spécifier des conditions à ce que l'on copie?!

# Améliorer *encore plus* notre code?

## De la version 4 à la version 5

---

- On peut utiliser **copy\_if** !
  - `template <class InputIterator, class OutputIterator, class UnaryPredicate>`  
`OutputIterator copy_if (InputIterator first,`  
`InputIterator last, OutputIterator result,`  
`UnaryPredicate pred);`
- Ça fonctionne comme **copy**, mais avec un paramètre de plus de type **UnaryPredicate**
- Un prédicat unaire est un des types récurrents de foncteurs... mais... c'est quoi un foncteur ?



## Parenthèse: les foncteurs

---

- Un foncteur est un objet fonction, c'est à dire que c'est un objet qui se comporte comme une fonction: il surcharge l'opérateur ()
- Étant des objets, ils permettent de faire des opérations que des fonctions de base ne peuvent pas faire:
  - Si j'ai une fonction qui permet de toujours incrémenter de  $X$ , il n'est pas possible de facilement changer cette valeur  $X$ .
  - Avec un foncteur,  $X$  peut être un attribut de l'objet, et être défini à la construction ou même changé par la suite

# Parenthèse: les foncteurs

---

- Par exemple:

```
class Additionner {  
public:  
    Additionner(int ajout)  
        : ajout_(ajout) {};  
    void operator()(int &val) {  
        val += ajout_;  
    };  
private:  
    int ajout_;  
};
```

```
vector<int> v;  
// push_back de valeurs...  
  
/* Algorithme pour appeler un  
 * foncteur sur tous les  
 * éléments d'un intervalle  
 */  
for_each(v.begin(),  
         v.end(),  
         Additionner(10));
```

- Avec une fonction, il aurait fallu utiliser une variable globale par exemple pour pouvoir spécifier la valeur.

# Parenthèse: les foncteurs

---

- La STL définit trois catégories de foncteurs:
  - Les générateurs, qui ne reçoivent aucun paramètre entre parenthèses et dont le but est de générer une valeur (utilisés en particulier avec des algorithmes servant à remplir des conteneurs par exemple)
  - Les foncteurs unaires, qui reçoivent un paramètre
  - Les foncteurs binaires, qui reçoivent deux paramètres

## Parenthèse: les foncteurs

---

- Un foncteur unaire ou binaire qui retourne une valeur de type booléen (vrai ou faux) sera appelé un prédicat.
- Que ce soit un générateur, unaire, ou binaire, un foncteur n'a aucune obligation de retourner la même valeur pour chaque appel identique.
- Le nombre de paramètres du constructeur d'un foncteur n'est pas lié à son type!
  - `MonFonctGen gen(i); cout << gen();`

# Améliorer *encore plus* notre code?

## De la version 4 à la version 5

---

- Ok... Donc avec ce qu'on vient de voir, on pourrait avoir le code suivant:
- Foncteur:

```
class IntInRange {  
public:  
    IntInRange(int min, int max)  
        : min_(min), max_(max) {};  
    bool operator()(int val) {  
        return (val >= min_  
                && val <= max_);  
    };  
private:  
    int min_, max_;  
};
```
- Algorithme: 

```
copy_if(array.begin(), array.end(),  
        ostream_iterator<int>(cout, "\n"),  
        IntInRange(10, 30));
```

# Prenons un exemple...

## Version 5: foncteurs

```
#include <vector>    // vector
#include <algorithm> // sort
#include <iostream>  // cin, cout, ..
#include <iterator>  // *_iterator, *_inserter
using namespace std;
```

```
class IntInRange {
public:
    IntInRange(int min, int max)
        : min_(min), max_(max) {};
    bool operator()(int val) {
        return (val >= min_
                && val <= max_);
    };
private:
    int min_, max_;
};
```

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

    cout << "Utiliser Ctrl+D (Linux) ou "
          << "Ctrl+Z (Windows) pour finir"
          << endl;
    copy(istream_iterator<int>(cin),
         istream_iterator<int>(),
         back_inserter(array));
```

```
    // Ordonner les éléments
    sort(array.begin(), array.end());
```

```
    cout << "Entiers ordonnés:" << endl;
    vector<int>::iterator end = array.end();
    copy(array.begin(), end,
          ostream_iterator<int>(cout, "\n"));
```

```
    // Afficher juste ceux entre 10 et 30
    cout << "Entiers avec une valeur entre 10 et 30:"
          << endl;
```

```
for (vector<int>::iterator it = array.begin();
    it != end; it++) {
    if (*it >= 10 && *it <= 30)
    cout << *it << endl;
}
```

```
copy_if(array.begin(), end(),
         ostream_iterator<int>(cout, "\n"),
         IntInRange(10, 30));
```

```
return 0;
```

```
}
```



# Prenons un exemple...

## Version 5: foncteurs

```
#include <vector>    // vector
#include <algorithm> // sort
#include <iostream>  // cin, cout, ..
#include <iterator>  // *_iterator, *_inserter
using namespace std;
```

```
class IntInRange {
public:
    IntInRange(int min, int max)
        : min_(min), max_(max) {};
    bool operator()(int val) {
        return (val >= min_
                && val <= max_);
    };
private:
    int min_, max_;
};
```

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

    cout << "Utiliser Ctrl+D (Linux) ou "
          << "Ctrl+Z (Windows) pour finir"
          << endl;
    copy(istream_iterator<int>(cin),
         istream_iterator<int>(),
         back_inserter(array));
```

```
    // Ordonner les éléments
    sort(array.begin(), array.end());
```

```
    cout << "Entiers ordonnés:" << endl;
    vector<int>::iterator end = array.end();
    copy(array.begin(), end,
          ostream_iterator<int>(cout, "\n"));
```

```
    // Afficher juste ceux entre 10 et 30
    cout << "Entiers avec une valeur entre 10 et 30:"
          << endl;
    copy_if(array.begin(), end(),
             ostream_iterator<int>(cout, "\n"),
             IntInRange(10, 30));
```

```
    return 0;
```

```
}
```

# Parenthèse: les foncteurs

---

- La STL fournit aussi un ensemble de foncteurs prédéfinis représentant les opérateurs classiques:
  - `plus<T>`, `minus<T>`, ...
  - `equal_to<T>`, `greater<T>`, ...
  - `logical_and<T>`, `logical_or<T>`, ...
- Un système pour combiner ces opérateurs à des valeurs et créer des foncteurs plus complexes est aussi disponible via **bind** depuis C++11:
  - `template< class F, class... Args >`  
`/*unspecified*/ bind( F&& f, Args&&... args );`

# Parenthèse: les foncteurs

---

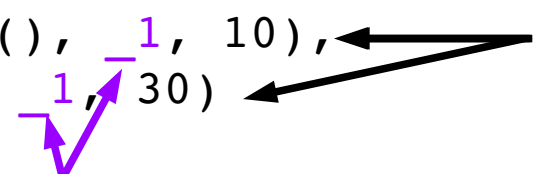
- L'utilisation de **bind** est permise par la disponibilité de paramètres fictifs, ou *placeholders*, qui se trouvent dans le namespace **std::placeholders**.
  - `_1, _2, _3, _4, ...`
- Un exemple d'appel à **bind**:
  - `f = bind(greater<int>(), _1, 2);`
    - `f(x)` retourne vrai si `x > 2`
  - `f = bind(greater<int>(), 2, _1);`
    - `f(x)` retourne vrai si `2 > x`
- `_1` signifie “paramètre 1 de l'appel à la fonction finale”, on peut utiliser ce paramètre comme second paramètre de la fonction sur laquelle on applique **bind**!

# Améliorer *encore plus* notre code?

## De la version 5 à la version 6

- On pourrait utiliser les commodités de la STL pour remplacer notre foncteur à l'aide de **bind**!
- On a donc besoin d'une fonction finale qui nous dit si l'unique paramètre qu'on lui passe est supérieur ou égal à 10 (**bind**, `greater_equal`) et (**bind**, `logical_and`) inférieur ou égal à 30 (**bind**, `less_equal`)

```
bind(  
    logical_and<bool>(),  
    bind(greater_equal<int>(), 1, 10),  
    bind(less_equal<int>(), 1, 30)  
)
```



Seul et unique paramètre qui sera  
reçu par la fonction générée par **bind**

# Prenons un exemple...

## Version 6: foncteurs prédéfinis

```
#include <vector>          // vector
#include <algorithm>        // sort
#include <iostream>         // cin, cout, ..
#include <iterator>         // *_iterator, *_inserter
#include <functional>        // bind, foncteurs STL
using namespace std;
using namespace std::placeholders;

class IntInRange {
public:
    IntInRange(int min, int max)
        : min_(min), max_(max) {}
    bool operator()(int val) {
        return (val >= min_
            && val <= max_);
    }
private:
    int min_, max_;
}

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

    cout << "Utiliser Ctrl+D (Linux) ou "
        << "Ctrl+Z (Windows) pour finir"
        << endl;
    copy(istream_iterator<int>(cin),
        istream_iterator<int>(),
        back_inserter(array));
```

```
// Ordonner les éléments
sort(array.begin(), array.end());

cout << "Entiers ordonnés:" << endl;
vector<int>::iterator end = array.end();
copy(array.begin(), end,
    ostream_iterator<int>(cout, "\n"));

// Afficher juste ceux entre 10 et 30
cout << "Entiers avec une valeur entre 10 et 30:"
    << endl;
copy_if(array.begin(), end(),
    ostream_iterator<int>(cout, "\n"),
    IntInRange(10, 30));
    bind(
        logical_and<bool>(),
        bind(greater_equal<int>(), _1, 10),
        bind(less_equal<int>(), _1, 30)
    ));

return 0;
```

```
}
```

# Prenons un exemple...

## Version 6: foncteurs prédéfinis

---

```
#include <vector>      // vector
#include <algorithm>    // sort
#include <iostream>     // cin, cout, ..
#include <iterator>     // *_iterator, *_inserter
#include <functional>   // bind, foncteurs STL
using namespace std;
using namespace std::placeholders;

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

    cout << "Utiliser Ctrl+D (Linux) ou "
          << "Ctrl+Z (Windows) pour finir"
          << endl;
    copy(istream_iterator<int>(cin),
         istream_iterator<int>(),
         back_inserter(array));

    // Ordonner les éléments
    sort(array.begin(), array.end());

    cout << "Entiers ordonnés:" << endl;
    vector<int>::iterator end = array.end();
    copy(array.begin(), end,
          ostream_iterator<int>(cout, "\n"));

    // Afficher juste ceux entre 10 et 30
    cout << "Entiers avec une valeur entre 10 et 30:"
          << endl;
    copy_if(array.begin(), end(),
             ostream_iterator<int>(cout, "\n"),
             bind(
                 logical_and<bool>(),
                 bind(greater_equal<int>(), _1, 10),
                 bind(less_equal<int>(), _1, 30)
             ));

    return 0;
}
```

```
// Ordonner les éléments
sort(array.begin(), array.end());

cout << "Entiers ordonnés:" << endl;
vector<int>::iterator end = array.end();
copy(array.begin(), end,
      ostream_iterator<int>(cout, "\n"));

// Afficher juste ceux entre 10 et 30
cout << "Entiers avec une valeur entre 10 et 30:"
      << endl;
copy_if(array.begin(), end(),
         ostream_iterator<int>(cout, "\n"),
         bind(
             logical_and<bool>(),
             bind(greater_equal<int>(), _1, 10),
             bind(less_equal<int>(), _1, 30)
         ));

return 0;
```

## Un dernier détail : le mot clé **auto**

---

- Lors de la déclaration d'une variable dont le type peut être facilement déduit, on peut utiliser le mot clé **auto**:
  - `auto a = 10;`
  - `auto b = monVecteur.size();`
  - `auto c = monVecteur;`
- Il est déconseillé d'utiliser **auto** partout, puisque le code serait plus difficile à suivre, mais il est bien pratique lorsque reconstituer le type est long, comme pour les itérateurs par exemple:
  - `vector<int>::iterator it1 = monVecteur.begin();`
  - `auto it2 = monVecteur.begin();`

# Prenons un exemple...

## Version 7: auto

```
#include <vector>      // vector
#include <algorithm>    // sort
#include <iostream>     // cin, cout, ..
#include <iterator>     // *_iterator, *_inserter
#include <functional>   // bind, foncteurs STL
using namespace std;
using namespace std::placeholders;

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

    cout << "Utiliser Ctrl+D (Linux) ou "
          << "Ctrl+Z (Windows) pour finir"
          << endl;
    copy(istream_iterator<int>(cin),
         istream_iterator<int>(),
         back_inserter(array));
}
```

```
// Ordonner les éléments
sort(array.begin(), array.end());

cout << "Entiers ordonnés:" << endl;
vector<int>::iterator end = array.end();
auto end = array.end();
copy(array.begin(), end,
      ostream_iterator<int>(cout, "\n"));

// Afficher juste ceux entre 10 et 30
cout << "Entiers avec une valeur entre 10 et 30:"
      << endl;
copy_if(array.begin(), end(),
         ostream_iterator<int>(cout, "\n"),
         bind(
             logical_and<bool>(),
             bind(greater_equal<int>(), _1, 10),
             bind(less_equal<int>(), _1, 30)
         ));

return 0;
```



# Prenons un exemple...

## Version 7: auto

```
#include <vector>      // vector
#include <algorithm>    // sort
#include <iostream>     // cin, cout, ..
#include <iterator>     // *_iterator, *_inserter
#include <functional>   // bind, foncteurs STL
using namespace std;
using namespace std::placeholders;

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

    cout << "Utiliser Ctrl+D (Linux) ou Ctrl+Z (Windows) pour finir" << endl;
    copy(istream_iterator<int>(cin), istream_iterator<int>(), back_inserter(array));

    // Ordonner les éléments
    sort(array.begin(), array.end());

    cout << "Entiers ordonnés:" << endl;
    auto end = array.end();
    copy(array.begin(), end, ostream_iterator<int>(cout, "\n"));

    // Afficher juste ceux entre 10 et 30
    cout << "Entiers avec une valeur entre 10 et 30:" << endl;
    copy_if(array.begin(), end(), ostream_iterator<int>(cout, "\n"),
        bind(logical_and<bool>(), bind(greater_equal<int>(), _1, 10),
            bind(less_equal<int>(), _1, 30)));

    return 0;
}
```

# Après la pratique... la théorie!

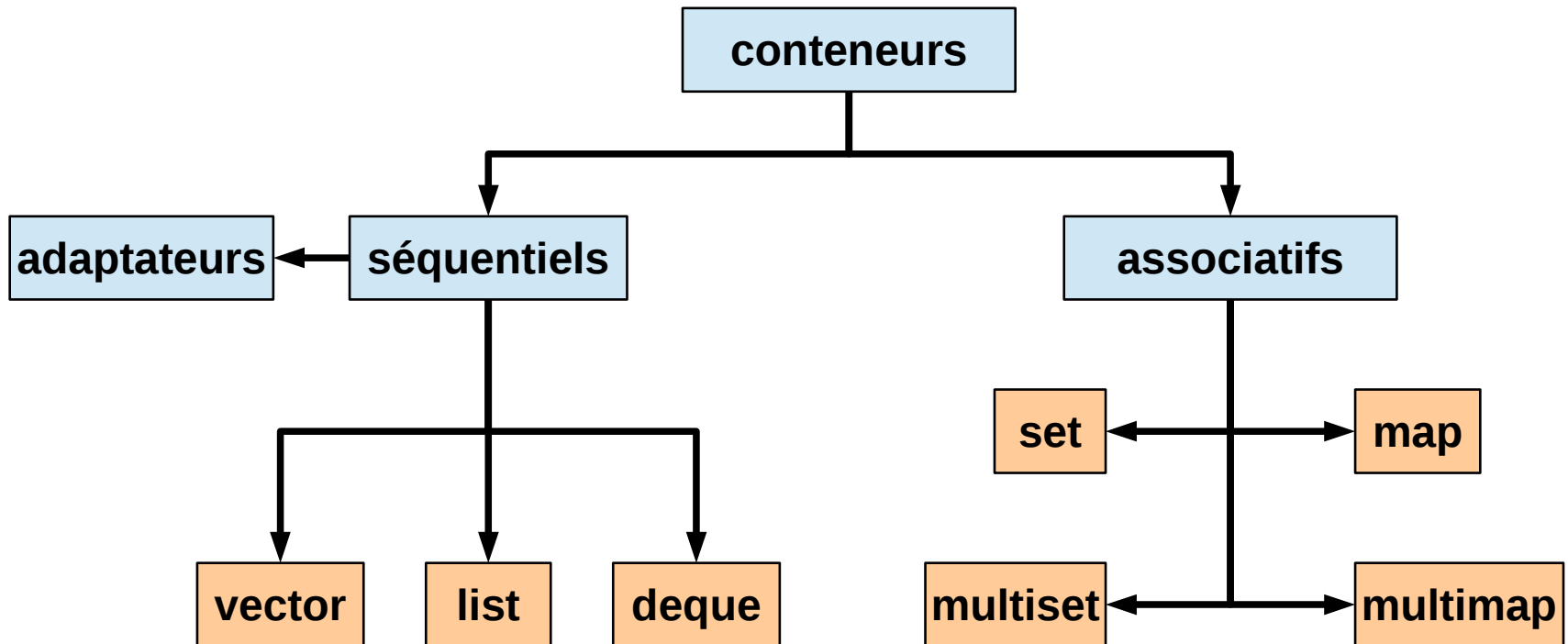
---

- On a vu comment on pouvait appliquer les différents composants de la STL à un cas pratique... et on a déjà abordé une partie de la théorie
- Allons voir ce que nous n'avons pas encore vu!



# Comment choisir le bon conteneur?

---



# Comment choisir le bon conteneur?

---

- La STL fournit trois catégories de conteneurs:
  - Les conteneurs séquentiels
  - Les adaptateurs de conteneurs, basés sur des conteneurs séquentiels
  - Les conteneurs associatifs
- Le choix dépend de...
  - ... comment l'on veut stocker nos objets
  - ... quelles opérations on veut faire sur nos objets
- Il faut donc connaître ces conteneurs!

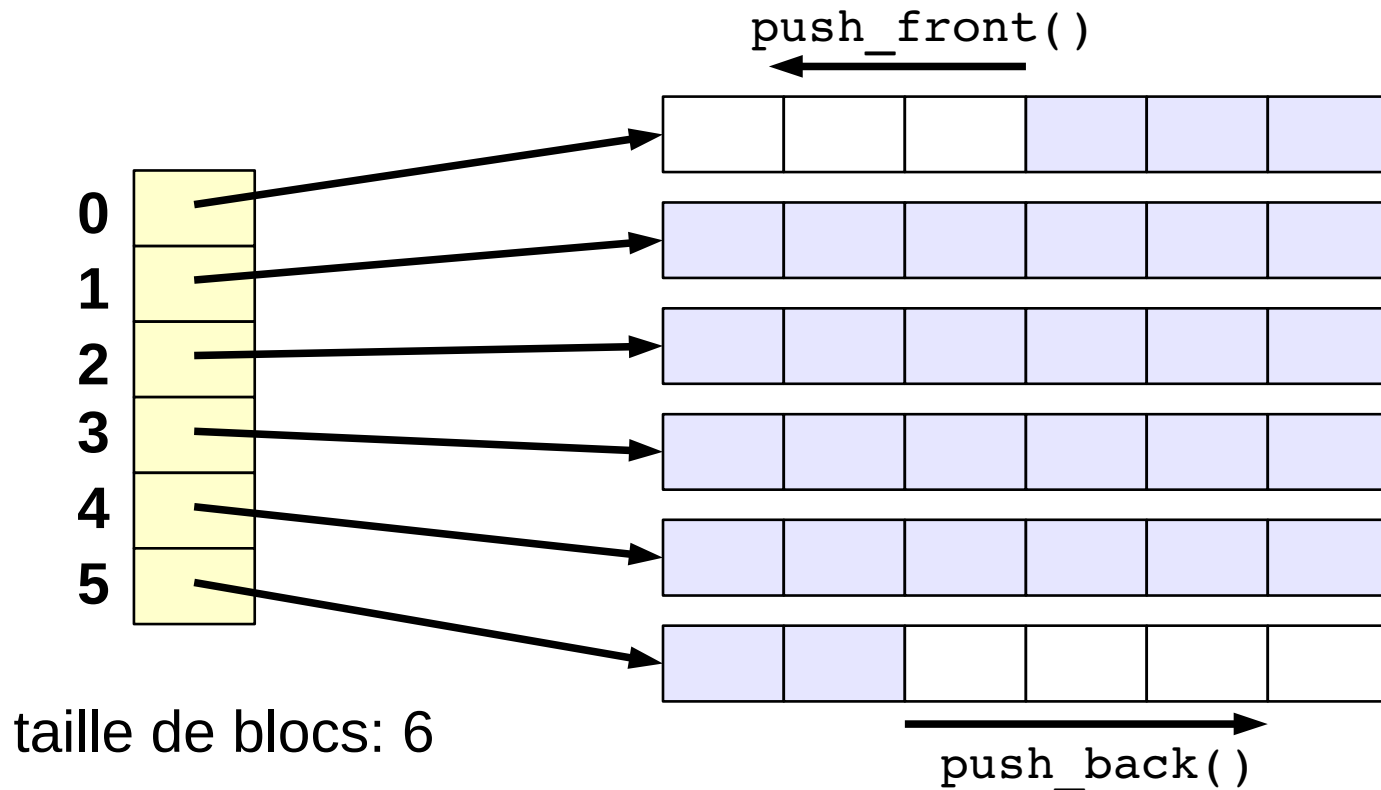
# Comment choisir le bon conteneur?

---

- Conteneurs séquentiels:
  - **vector**
    - Insertion et lecture rapide à la fin
    - Permet l'accès aléatoire (opérateur [])
  - **list** (et **forward\_list**)
    - Insertion rapide partout
    - Accès séquentiel seulement
  - **deque** (prononcez “deck”) : double-ended queue
    - Insertion rapide au début ou à la fin
    - Permet l'accès aléatoire (plus lent que **vector**)

# Comment choisir le bon conteneur?

- Conteneurs séquentiels: **deque**



# Comment choisir le bon conteneur?

- Conteneurs séquentiels: quelques méthodes

Taille	<ul style="list-style-type: none"><li>– <b>size()</b>: combien d'éléments contenus</li><li>– <b>empty()</b>: vrai si le conteneur est vide</li></ul>
Insertion	<ul style="list-style-type: none"><li>– <b>push_back(x)</b>: ajouter <b>x</b> à la fin (vector, list, deque)</li><li>– <b>push_front(x)</b>: ajouter <b>x</b> au début (list, deque)</li><li>– <b>insert(it, x)</b>: ajouter <b>x</b> avant l'élément pointé par <b>it</b></li></ul>
Retrait	<ul style="list-style-type: none"><li>– <b>pop_back()</b>: retirer à la fin (vector, list, deque)</li><li>– <b>pop_front()</b>: retirer au début (list, deque)</li><li>– <b>erase(it)</b>: retirer l'élément pointé par <b>it</b></li><li>– <b>clear()</b>: vider le conteneur</li></ul>
Lecture	<ul style="list-style-type: none"><li>– <b>back()</b>: valeur de l'élément à la fin (vector, list, deque)</li><li>– <b>front()</b>: valeur de l'élément au début (list, deque)</li></ul>

- Plus? [http://fr.cppreference.com/w/cpp/container/](http://fr.cppreference.com/w/cpp/container/vector)**vector**  
**list**  
**deque**

## Parenthèse: pourquoi `pop_*` ne retourne pas la valeur de l'élément?

---

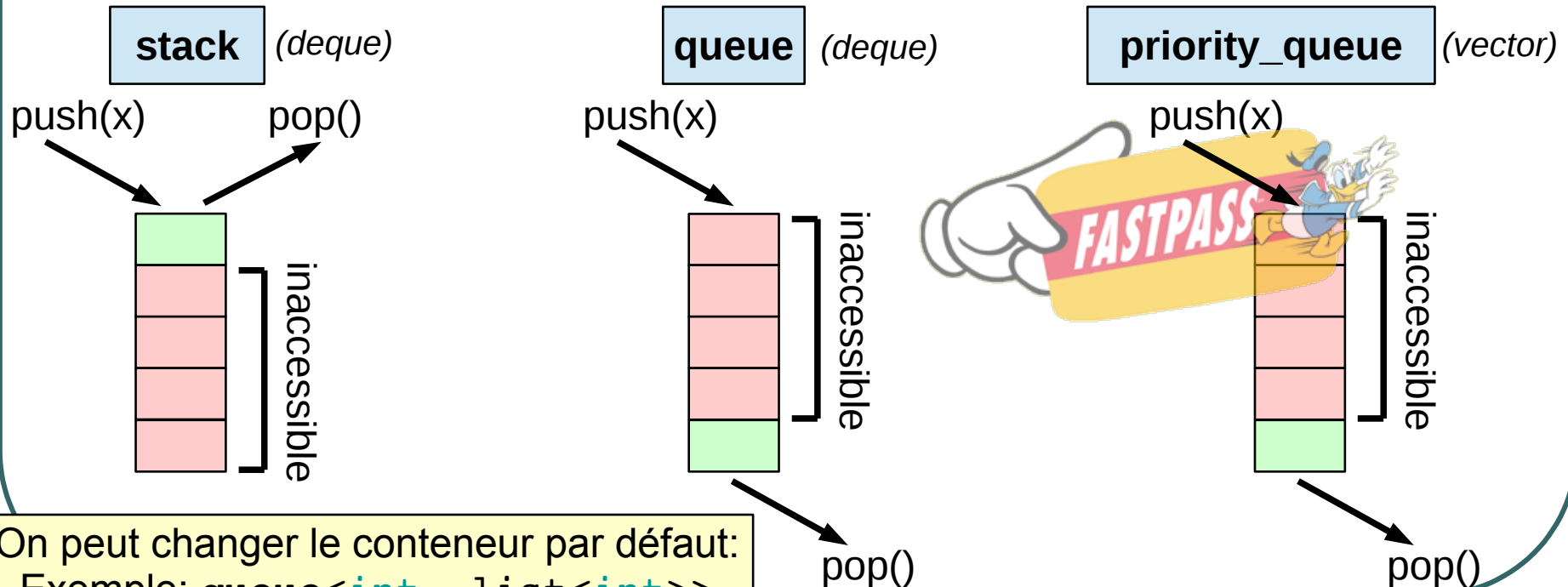
- C'est une question de performance
  - L'objet est détruit dans le conteneur: on ne peut donc pas le renvoyer par référence
  - Un renvoi par valeur prend du temps pour la copie
  - Est-ce qu'on utilise la valeur retournée? Pas toujours! On perd donc du temps pour rien...
- On préférera donc la boucle suivante pour afficher et sortir les éléments:

```
while (!conteneur.empty()) {  
    cout << conteneur.back() << endl; // ou .front()  
    conteneur.pop_back(); // ou .pop_front()  
}
```



# Comment choisir le bon conteneur?

- Adaptateurs de conteneurs (séquentiels)
  - Pour une utilisation spécifique: seules les fonctions utiles pour cette utilisation sont accessibles



On peut changer le conteneur par défaut:  
Exemple: `queue<int, list<int>>`

# Comment choisir le bon conteneur?

---

- Adaptateurs de conteneurs (séquentiels)
  - **push()** et **pop()** sont utilisés sur ces adaptateurs au lieu de **push\_back()**, **push\_front()**, **pop\_back()** et **pop\_front()** sur les conteneurs séquentiels.
  - **top()** permet de récupérer la valeur de l'élément accessible (au lieu de **back()** ou **front()**)
  - Tous les éléments ne sont pas accessibles pour les adaptateurs de conteneurs: on ne peut donc pas itérer dessus
  - Pas d'itérateurs = on ne peut pas utiliser les algorithmes de la STL sur ces conteneurs!

# Comment choisir le bon conteneur?

---

- Conteneurs associatifs
  - Généralisation des conteneurs séquentiels
  - Les conteneurs séquentiels sont indexés par un entier (0, 1, 2, etc..), tandis que les conteneurs associatifs peuvent l'être par n'importe quel type (type de l'élément choisi pour clé)
  - La clé est toujours **const**
  - La clé détermine la position d'insertion ou de retrait (vs. l'index pour les séquentiels)

# Comment choisir le bon conteneur?

---

- Conteneurs associatifs
  - **map** (et **multimap**)
    - “mapping” d'un type (la clé) vers un autre type (la valeur)
    - Permet d'associer une (**map**) ou plusieurs (**multimap**) valeurs à une clé
    - Permet, de façon efficace, de récupérer la valeur associée à une clé
    - Permet d'itérer au travers des clés de la **map** (ordonnée par clé)

# Comment choisir le bon conteneur?

---

- Conteneurs associatifs
  - **map** (et **multimap**)
    - Contient des objets de type **pair**, dont la clé est constante

```
// pair se trouve dans <utility>
#include <utility>
template <class T1, class T2>
class pair {
public:
    /* ... */
    T1 first;
    T2 second;
};
```

```
map<int,double>::iterator it;
it = uneMap.begin();
// Refusé à la compilation:
*it = make_pair(2, 5.9);
// Refusé à la compilation:
(*it).first = 2;
// Accepté:
(*it).second = 5.9;
```

- **make\_pair** construit une paire en déduisant les types via ses paramètres

# Comment choisir le bon conteneur?

---

- Conteneurs associatifs
  - **set** (et **multiset**)
    - Permet d'ajouter/supprimer des éléments
    - Éléments uniques dans **set**, potentiellement multiples dans **multiset**
    - Vérifier l'existence d'un élément
    - Itérer au travers du **set** (ordonné par élément)
    - Très proche de **map**, mais ici la clé et la valeur sont un même et identique élément
    - On ne peut donc pas changer un élément, il faut le retirer et insérer la nouvelle valeur: Pourquoi?

# Comment choisir le bon conteneur?

---

- Conteneurs associatifs
  - ... car les conteneurs associatifs sont ordonnés, c'est à dire que les valeurs sont rangées par rapport à la valeur de leurs clés.
  - Les clés doivent donc **surcharger** l'opérateur utilisé pour le tri!
  - Par défaut, cet opérateur est <, ou `less<clé>`
  - On peut indiquer un autre opérateur à la construction: `map<string, double, greater<string>>`
  - Les éléments sont ordonnés en tout temps: ils sont triés lors de l'insertion pour une lecture rapide

# Comment choisir le bon conteneur?

---

- Conteneurs associatifs
  - Il y a cependant une exception: les conteneurs associatifs non ordonnés, ou hashés
  - Ces conteneurs sont préfixés par `unordered_`:

<code>unordered_map</code>	<code>unordered_set</code>
<code>unordered_multimap</code>	<code>unordered_multiset</code>
  - Ils ne sont pas ordonnés par une clé, mais par ce qu'on appelle une table de hachage
  - L'insertion et la recherche d'éléments dans ces conteneurs est considérée comme étant du temps constant en moyenne



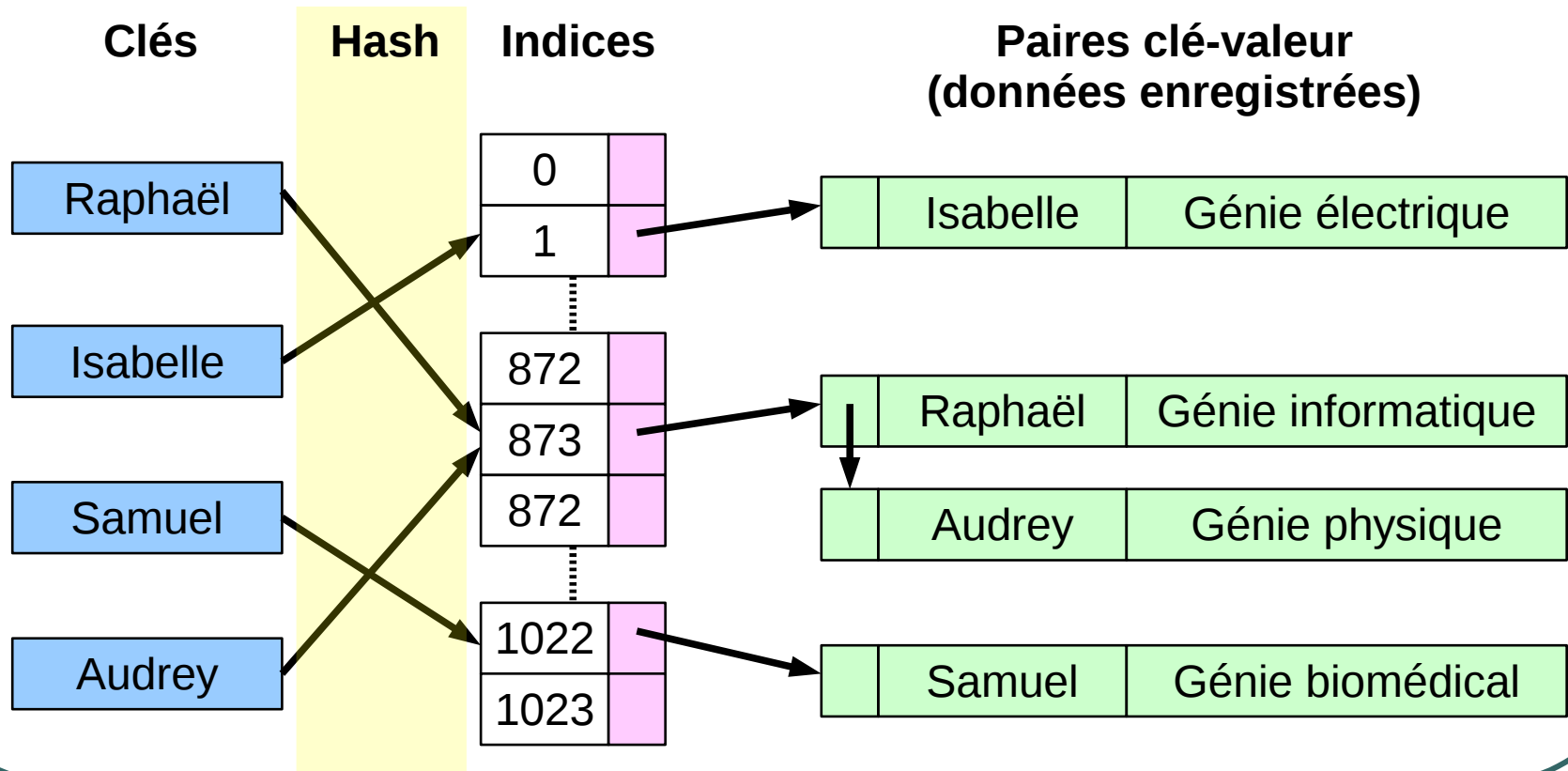
# Parenthèse: les tables de hachage

---

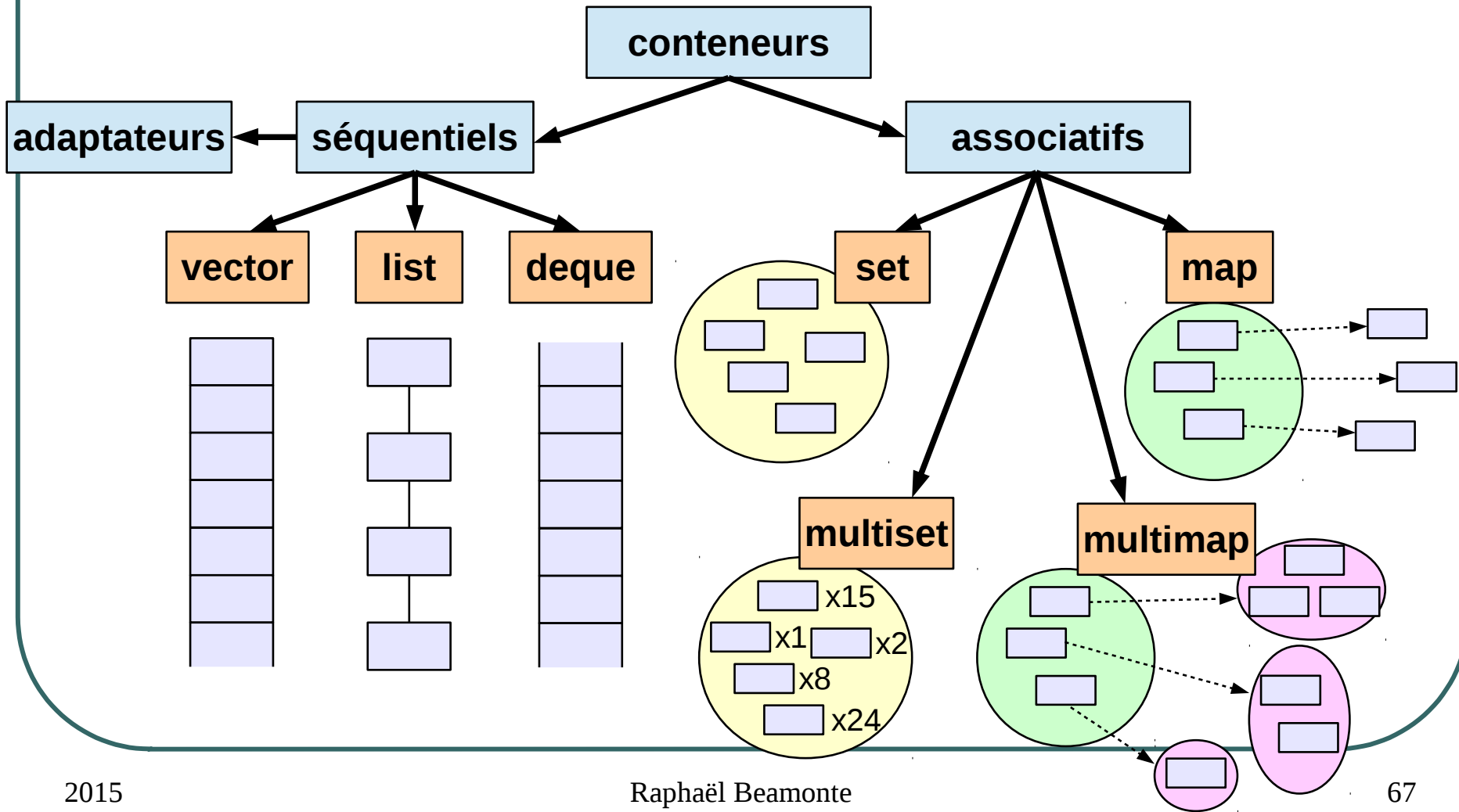
- Hasher un élément consiste à le passer dans une fonction dont le retour est une valeur plus courte et/ou de taille fixe qui permet de le ranger dans une case
- Plutôt que d'analyser caractère par caractère notre clé, on analyse une valeur beaucoup plus rapide à analyser (un entier par exemple)
- Plusieurs valeurs peuvent se retrouver dans la même case, mais beaucoup moins que le nombre de valeurs totales
- Il est quand même pertinent d'augmenter le nombre de cases lorsque cela arrive trop souvent

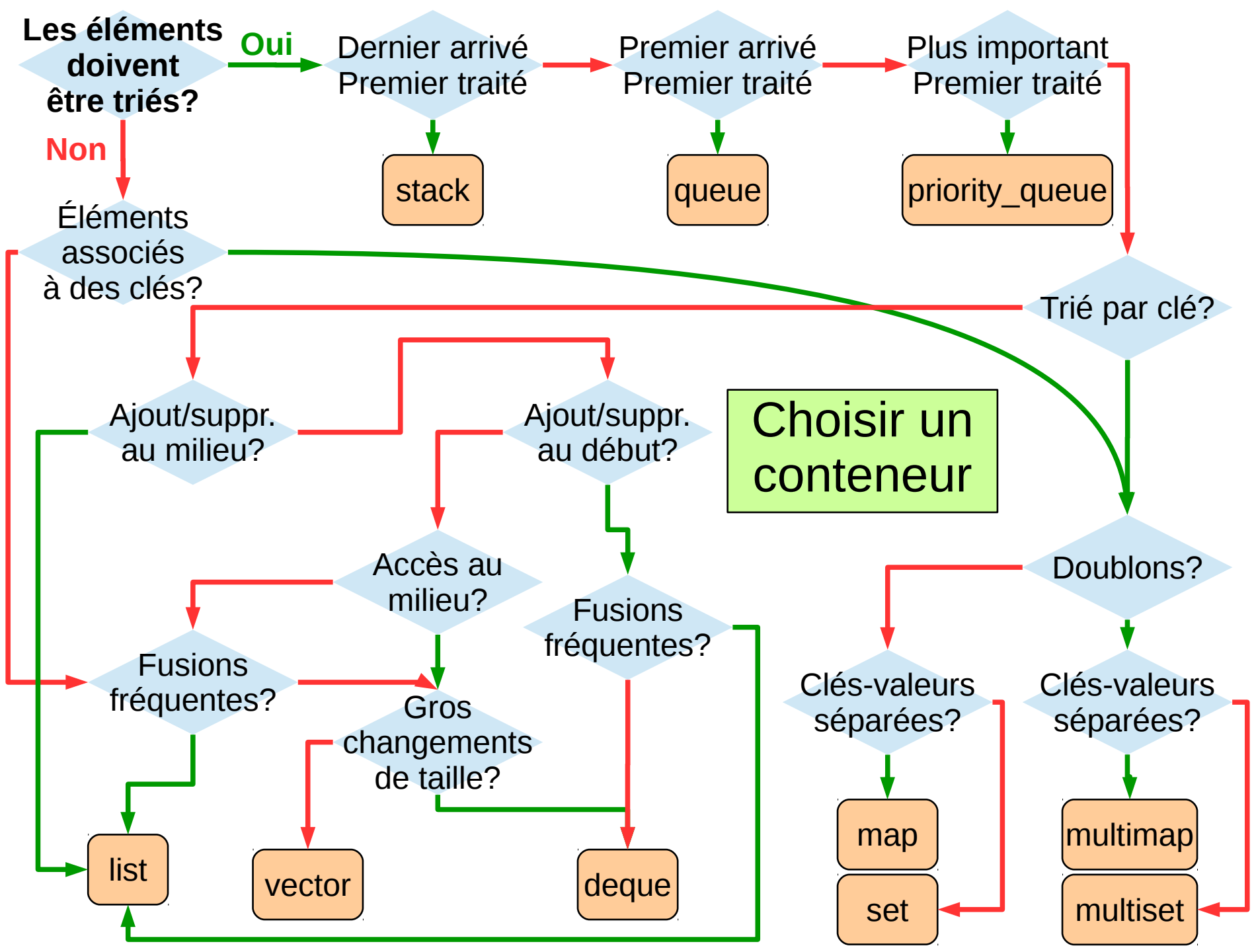
# Parenthèse: les tables de hachage

- Représentation du hachage d'une **map**:



# Récapitulatif: représentation graphique des conteneurs

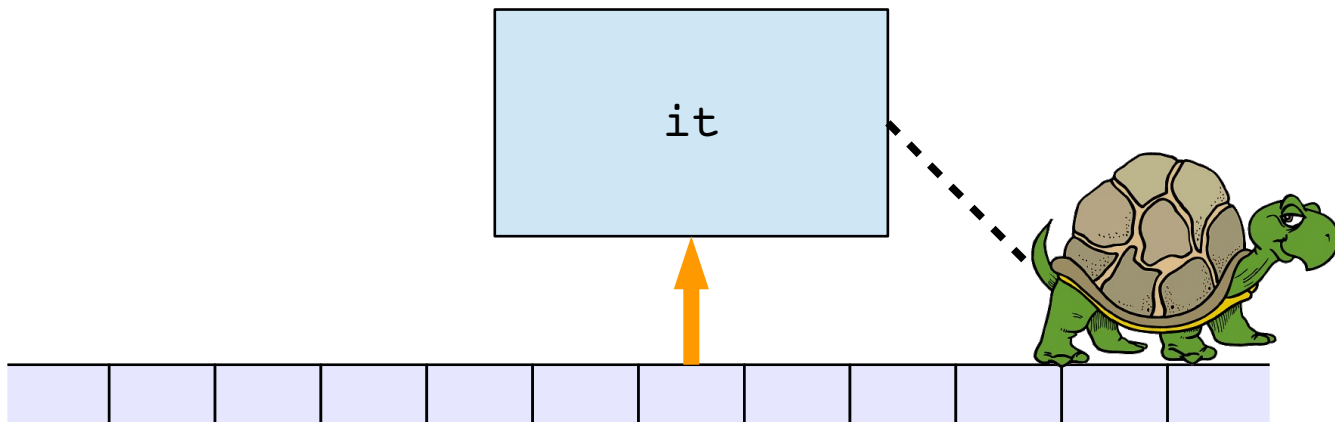




# Quels sont les différents types d'itérateurs?

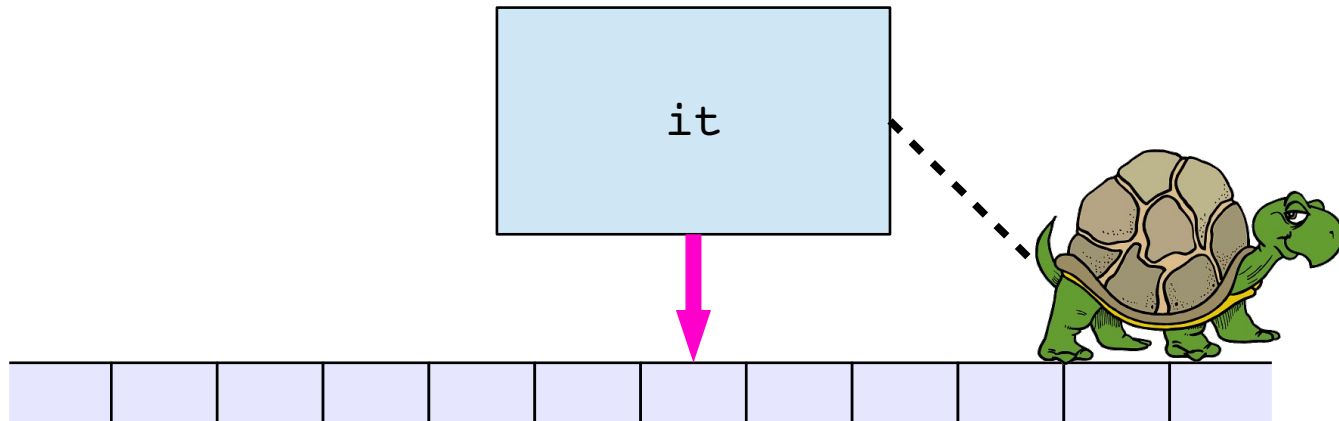
---

- Les itérateurs d'entrée (Input Iterators)
  - Donne accès à une source de données (lecture!)
  - Cette source peut être un conteneur, un stream, etc.
  - On en a vu un exemple avec `istream_iterator`!



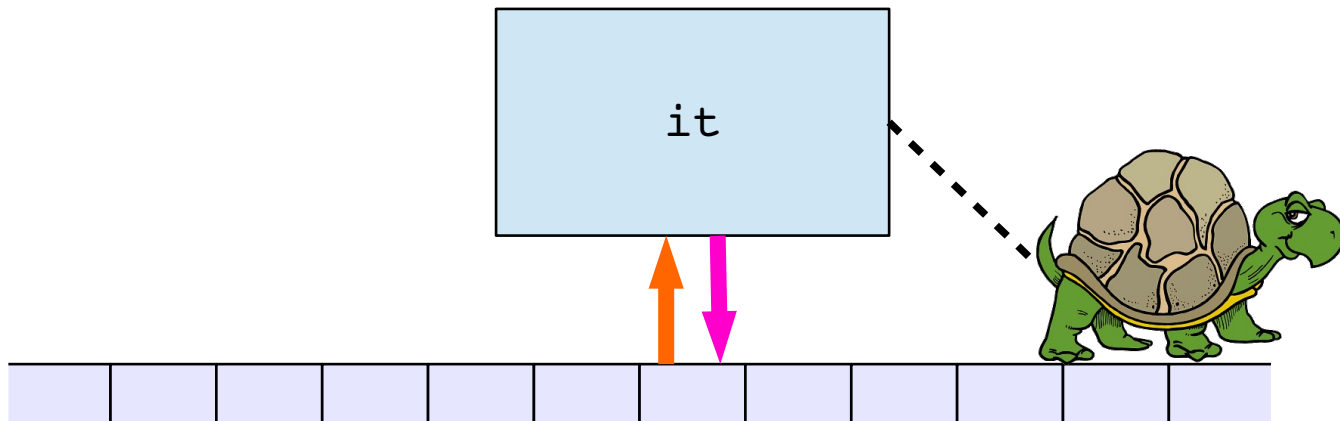
# Quels sont les différents types d'itérateurs?

- Les itérateurs de sortie (Output Iterators)
  - Donne accès à des collecteurs de données (emplacement pour stocker des informations) (écriture!)
  - Le collecteur peut être un conteneur, un stream, etc.
  - On en a vu un exemple avec `ostream_iterator`, ainsi qu'avec les introducteurs!



# Quels sont les différents types d'itérateurs?

- Les itérateurs avançant (Forward Iterators)
  - Permettent tout ce que permettent les itérateurs d'entrée et de sortie (déréférencement, lecture/écriture, `it++`, `it1 == it2`, `*it1 == *it2`)
  - Peuvent traverser un intervalle (une ou plusieurs fois, si on repart du même itérateur d'origine) via incrémentation



# Quels sont les différents types d'itérateurs?

---

- Les itérateurs avançant (Forward Iterators)
  - De nombreux algorithmes utilisent ce genre d'itérateurs:

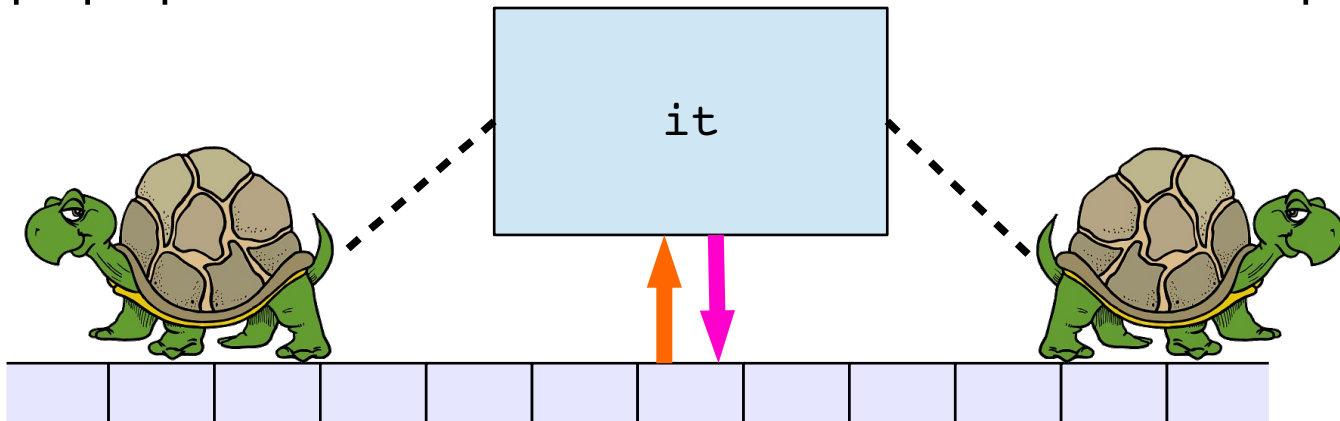
```
int *p = new int [1000];  
fill (p, p+100, 0);  
fill (p+101, p+1000, 30);
```

Les arguments 1 et 2 de **fill** sont des itérateurs avançants pour spécifier l'intervalle de travail.  
L'argument 3 est la valeur à mettre dans cet intervalle.



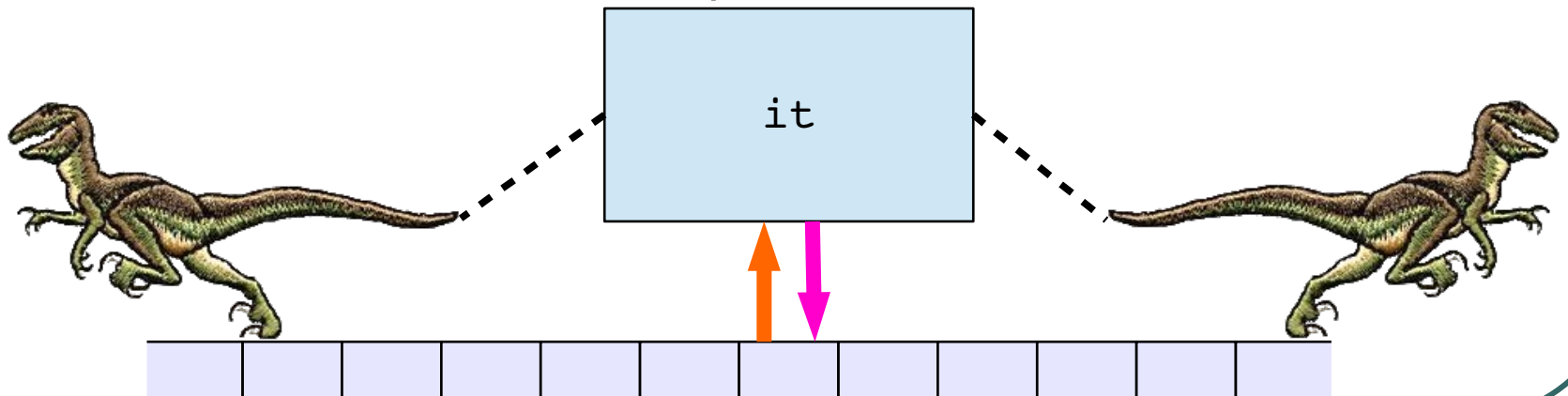
# Quels sont les différents types d'itérateurs?

- Les itérateurs bidirectionnels (Bidirectional Iterators)
  - Peuvent avancer (incrémentation) ou reculer (décrémentation)
  - À part ça, ils donnent les mêmes commodités que les itérateurs avançants
  - Comme pour les itérateurs avançants, déplacer un itérateur bidirectionnel d'un élément de la séquence à un autre prend un temps proportionnel au nombre d'éléments entre les deux positions.

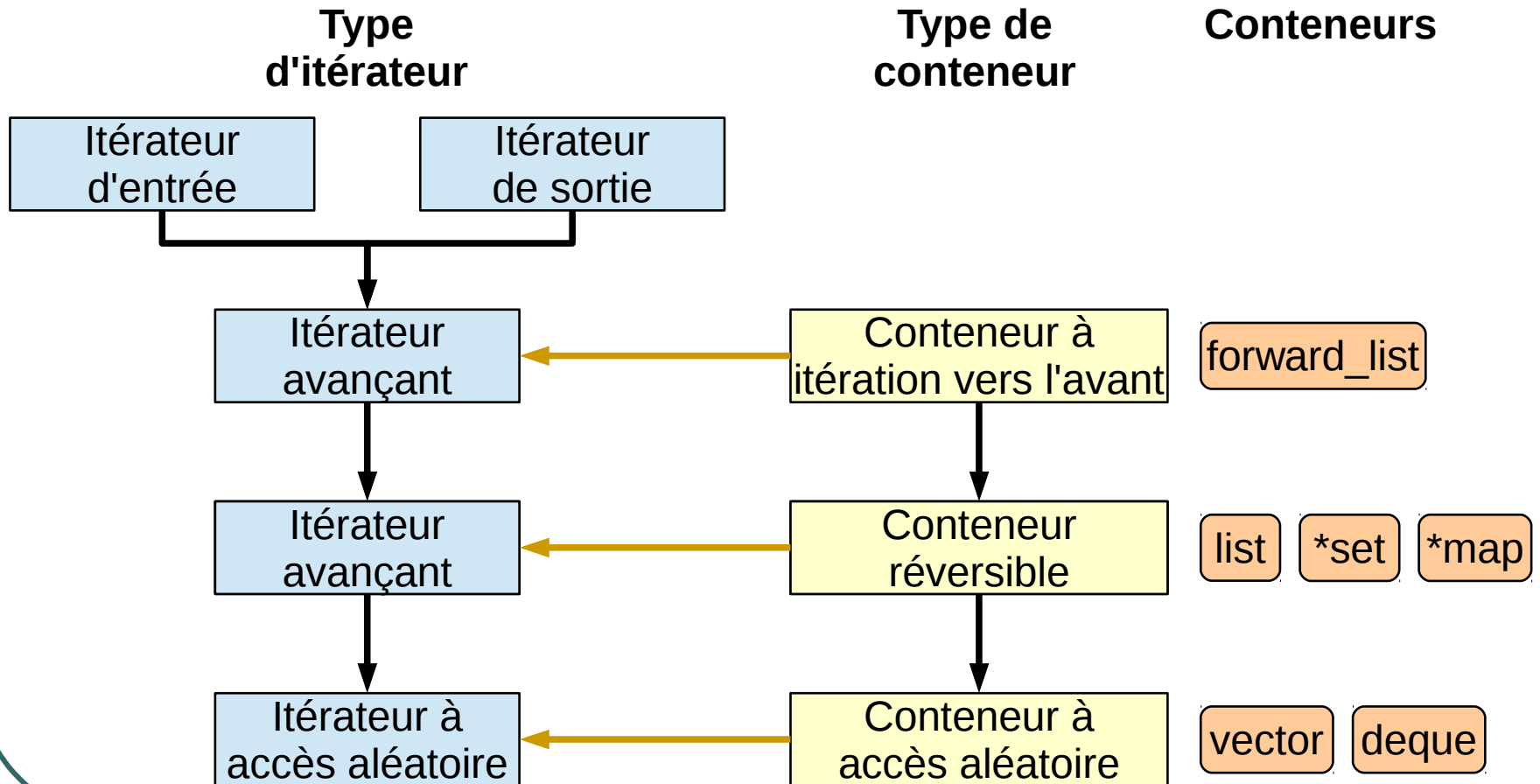


# Quels sont les différents types d'itérateurs?

- Les itérateurs à accès aléatoire (Random Access Iterators)
  - Mêmes commodités que les itérateurs à accès bidirectionnels
  - Peuvent aussi sauter de n'importe quelle position à n'importe quelle autre en temps constant
  - Accès à l'opérateur []
  - Tous les pointeurs C standards sont considérés comme des itérateurs à accès aléatoires pour les tableaux C



# Et quel conteneur fournit quel type d'itérateur?



# Et qu'en est-il des algorithmes?

- Les algorithmes de la STL sont des procédures qui sont appliquées sur les conteneurs pour traiter leurs données
- Les algorithmes utilisent des itérateurs pour travailler sur les conteneurs:

Algorithme	Début de l'intervalle	Fin de l'intervalle	Foncteur
<code>for_each</code>	<code>v.begin()</code>	<code>v.end()</code>	<code>negate&lt;int&gt;()</code>

- On en a déjà vu quelques uns dans les exemples précédents (`sort`, `fill`, ...)
- On identifie quatre catégories d'algorithmes dans la STL

# Et qu'en est-il des algorithmes?

---

- Il y a les algorithmes de tri:
  - Ils permettent de trier ou effectuer des opérations sur un conteneur trié
  - Ils trient par défaut avec l'opérateur  $<$ , mais on peut spécifier un autre foncteur
  - Ils permettent d'effectuer sur un conteneur trié:
    - Des recherches binaires
    - Trouver le minimum ou maximum
    - Faire des opérations sur les tas
    - Fusionner des conteneurs

# Et qu'en est-il des algorithmes?

- Il y a les algorithmes de tri:
  - Quelques algorithmes pour:
    - `template<class RandomIt, class Compare>`  
`void sort(RandomIt first,`  
`RandomIt last,`  
`Compare comp);`
      - Ne marche pas pour **list**  
Pourquoi?!
    - `template<class ForwardIt, class Compare>`  
`bool is_sorted(ForwardIt first,`  
`ForwardIt last,`  
`Compare comp);`

# Et qu'en est-il des algorithmes?

---

- Il y a les algorithmes de tri:
  - Quelques algorithmes (conteneur trié avant!):
    - `lower_bound` / `upper_bound` / `equal_range`
    - `binary_search`
    - `min` / `max` (deux valeurs)
    - `min_element` / `max_element` (valeurs d'un conteneur)

# Et qu'en est-il des algorithmes?

- Il y a les algorithmes de tri:
  - Quelques algorithmes (conteneur trié avant!):

```
#include <algorithm>
int main() {
    int* tab[] = {1, 5, 3, 8, 4, 12, 8, 15, 8};
    int nbElems = sizeof(tab) / sizeof(int);

    sort(tab, tab + nbElems); // {1, 3, 4, 5, 8, 8, 8, 12, 15}

    int* debut = lower_bound(tab, tab + nbElems, 8);
    int* fin = upper_bound(tab, tab + nbElems, 8);
    pair<int*, int*> inter = equal_range(tab, tab + nbElems, 8);

    assert(debut == inter.first && fin == inter.second);
    return 0;
}
```



# Et qu'en est-il des algorithmes?

---

- Il y a les algorithmes de tri:
  - Certains de ces algorithmes permettent de travailler sur les tas
  - Mais... C'est quoi les tas?



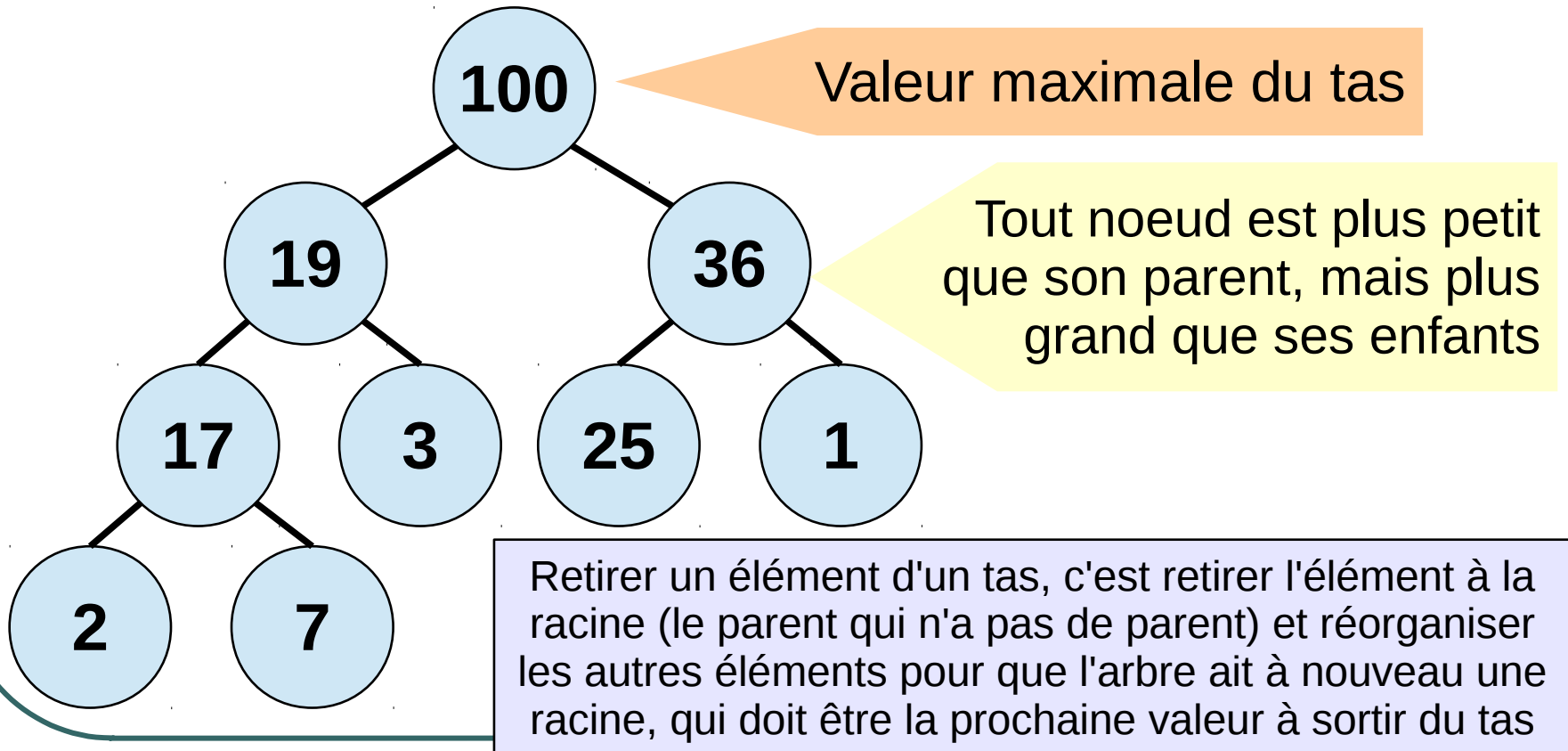
# Parenthèse: introduction aux tas

---

- Un tas, c'est une structure de données en arbre.
- Un tas garantit l'insertion et le retrait en  $O(\log(n))$
- En retirant tous les éléments d'un tas un par un, on les retirera en ordre croissant (*min heap*) ou décroissant (*max heap*)
- Dans la STL, on peut bien sûr spécifier l'opérateur de comparaison à utiliser

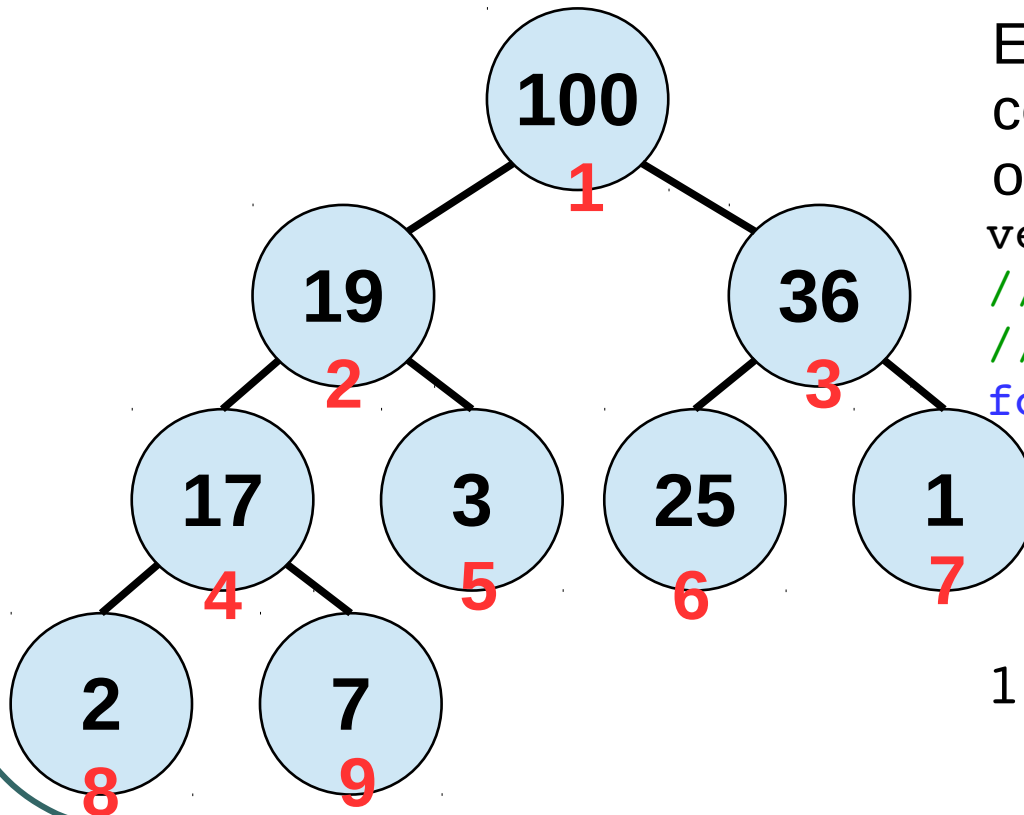
# Parenthèse: introduction aux tas

- Exemple de tas maximum (*max heap*):



# Parenthèse: introduction aux tas

- Exemple de tas maximum (*max heap*):



En affichant le contenu d'un conteneur transformé en tas, on aurait ainsi:

```
vector<int> v;  
// push_back ordre arbitraire  
// transformation en tas  
for (int i = 0; i < v.size();  
      i++)  
    cout << v[i] << " ";
```

100 19 36 17 3 25 1 2 7  
1 2 3 4 5 6 7 8 9

# Et qu'en est-il des algorithmes?

---

- Il y a les algorithmes de tri:
  - Quelques algorithmes pour les tas:
    - `make_heap`: transformer un intervalle en tas (pas besoin de `sort` avant)
    - `is_heap`: retourne vrai si un intervalle est organisé en tas
    - `sort_heap`: transforme un intervalle organisé en tas en intervalle trié (on suppose `is_heap == true` avant, et on attend `is_heap == false` après)

# Et qu'en est-il des algorithmes?

- Il y a les algorithmes de tri:
  - Quelques algorithmes pour les tas:
    - `push_heap` et `pop_heap` pour réorganiser le tas avec respectivement l'ajout et le retrait d'un élément:
      - Dans les deux cas, on suppose `is_heap == true` avant

```
vector<int> v;  
// push_back ordre arbitraire  
make_heap(v.begin(), v.end());  
v.push_back(8);  
push_heap(v.begin(), v.end());  
pop_heap(v.begin(), v.end());  
cout << v.back() << endl;  
v.pop_back();
```

**Ajout d'un élément**

**Retrait du prochain élément**

# Et qu'en est-il des algorithmes?

- Il y a les algorithmes de tri:
  - Quelques algorithmes pour les tas:
    - `push_heap` et `pop_heap` pour réorganiser le tas avec respectivement l'ajout et le retrait d'un élément:
      - Dans les deux cas, on suppose `is_heap == true` avant

```
vector<int> v;  
// push_back ordre arbitraire  
make_heap(v.begin(), v.end());  
v.push_back(8);  
push_heap(v.begin(), v.end());  
pop_heap(v.begin(), v.end());  
cout << v.back() << endl;  
v.pop_back();
```

Contenu de v

100	19	36	17	3	25	1	2	7	
100	19	36	17	3	25	1	2	7	8
100	19	36	17	8	25	1	2	7	3
36	19	25	17	8	3	1	2	7	100
36	19	25	17	8	3	1	2	7	

# Et qu'en est-il des algorithmes?

---

- Il y a les algorithmes de tri:
  - Quelques algorithmes de fusion:
    - On suppose les deux conteneurs à fusionner déjà triés selon l'opérateur voulu (à préciser si ce n'est pas  $<$ )
    - `merge`: fusionne deux intervalles, envoie le résultat dans un 5ème itérateur
    - `set_union/set_intersection`: envoie dans un 5ème itérateur l'union/intersection de deux intervalles
    - `set_difference`: envoie dans un 5ème itérateur les éléments du 1er intervalle qui ne sont pas dans le 2nd



# Et qu'en est-il des algorithmes?

---

- Il y a les algorithmes de recherche:
  - Ils font des tests sur les éléments pour trouver ce que l'on recherche (valeur, décompte, etc.)
  - Contrairement aux algorithmes de tri, lorsqu'on travaille sur deux conteneurs on n'aura ici généralement besoin que de l'itérateur du début du 2nd intervalle, car la taille du 1er intervalle déterminera le nombre d'éléments à traiter
  - Ils ne **modifient pas** les données, on peut donc utiliser aussi des ***itérateurs constants***.. C'est quoi?!

# Parenthèse: les types d'itérateurs spécifiques

---

- On a vu les différentes catégories d'itérateurs, mais en plus du type `iterator`, les conteneurs donnent accès à d'autres types d'itérateurs:
  - `reverse_iterator`
    - Ils permettent de parcourir un conteneur à l'envers
    - On utilise `.rbegin()` et `.rend()` pour récupérer l'intervalle d'un conteneur (bidirectionnel au moins)
  - `const_iterator`
    - Permet de respecter l'engagement `const`
  - `const_reverse_iterator`
    - Cumule `reverse_iterator` et `const_iterator`

# Et qu'en est-il des algorithmes?

---

- Il y a les algorithmes de recherche:
  - `count(it1, it2, val)`  
`count_if(it1, it2, prédicat unaire)`
    - Compte les éléments de l'intervalle répondant au critère (`== val` ou `prédicat == true`)
  - `find(it1, it2, val)`  
`find_if(it1, it2, prédicat unaire)`
    - Cherche le premier élément de l'intervalle répondant au critère (`== val` ou `prédicat == true`)
  - `equal(it1, it2, it3)`
    - Vérifie l'égalité de deux intervalles

# Et qu'en est-il des algorithmes?

---

- Il y a les algorithmes de modification de données:
  - Ils permettent de copier vers un itérateur de sortie
    - `copy(it1, it2, itS)`  
`copy_n(it1, nb, itS)`  
`copy_backward(it1, it2, itS)`
    - Retournent un itérateur pointant vers la fin de l'intervalle copié dans la sortie
    - `copy_n` permet de signifier un nombre d'éléments à copier au lieu d'un itérateur de fin. Tous les algorithmes finissant par `_n` proposent cette alternative.

# Et qu'en est-il des algorithmes?

---

- Il y a les algorithmes de modification de données:
  - Ils permettent d'échanger des éléments
    - `swap(obj1, obj2)`  
`iter_swap(it1, it2)`  
`swap_ranges(it1, it2, it3)`
    - Ces algorithmes sont particulièrement utiles comme opérations de base pour les algorithmes de tri

# Et qu'en est-il des algorithmes?

- Il y a les algorithmes de modification de données:
  - Ils permettent de remplacer des éléments répondant à un critère donné
    - `replace(it1, it2, val1, val2)`  
`replace_if(it1, it2, prédicat unaire, val2)`  
`replace_copy(it1, it2, itS, val1, val2)`  
`replace_copy_if(it1, it2, itS, prédicat unaire, val2)`
    - `*_copy*` ne modifient pas le conteneur mais envoient le résultat dans un itérateur de sortie, le retour est le même que pour `copy*`

# Et qu'en est-il des algorithmes?

---

- Il y a les algorithmes de modification de données:
  - Ils permettent de remplacer des éléments répondant à un critère donné

```
vector<unsigned char> samples;  
// push_back de valeurs dans le vecteur  
vector<unsigned char> cleanSamples;  
  
replace_copy_if(  
    samples.begin(),  
    samples.end(),  
    back_inserter(cleanSamples),  
    bind(greater<unsigned char>(), _1, 127),  
    127);
```

# Et qu'en est-il des algorithmes?

- Il y a les algorithmes de modification de données:
  - Ils permettent de remplir avec une valeur ou à l'aide d'un foncteur générateur
    - `fill(it1, it2, val) / fill_n`  
`generate(it1, it2, générateur) / generate_n`
    - Insérer 8 valeurs à l'aide d'un générateur:

```
class Fibonacci {  
public:  
    Fibonacci():  
        n1(1), n2(0) {};  
    int operator()() {  
        n1 = n1+n2;  
        n2 = n1-n2;  
        return n2;  
    };  
private:  
    int n1, n2;  
};
```

```
vector<int> v;  
generate_n(back_inserter(v),  
           8, Fibonacci());
```

1	1	2	3	5	8	13	21
---	---	---	---	---	---	----	----



# Et qu'en est-il des algorithmes?

---

- Il y a les algorithmes de modification de données:
  - Ils permettent de retirer des éléments répondant à un critère donné
    - `remove_copy(it1, it2, itS, val) / remove_copy_if`  
`unique_copy (it1, it2, itS)`
    - `remove(it1, it2, val) / remove_if`  
`unique(it1, it2)`
      - Retournent un itérateur qui indique la nouvelle fin du conteneur: les éléments qui suivent sont indéterminés. L'algorithme `erase` permet de retirer ces éléments du conteneur

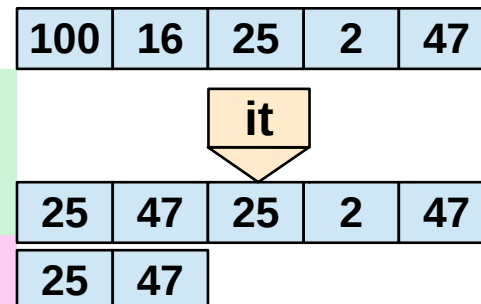
# Et qu'en est-il des algorithmes?

- Il y a les algorithmes de modification de données:
  - Ils permettent de retirer des éléments répondant à un critère donné

```
bool estPair(int i) {  
    return (i%2 == 0);  
}
```

```
vector<int> v;  
// push_back ordre arbitraire  
// de 100, 16, 25, 2, 47  
auto it = remove_if(v.begin(),  
                    v.end(),  
                    estPair);  
v.erase(it, v.end());
```

Contenu de v



# Et qu'en est-il des algorithmes?

---

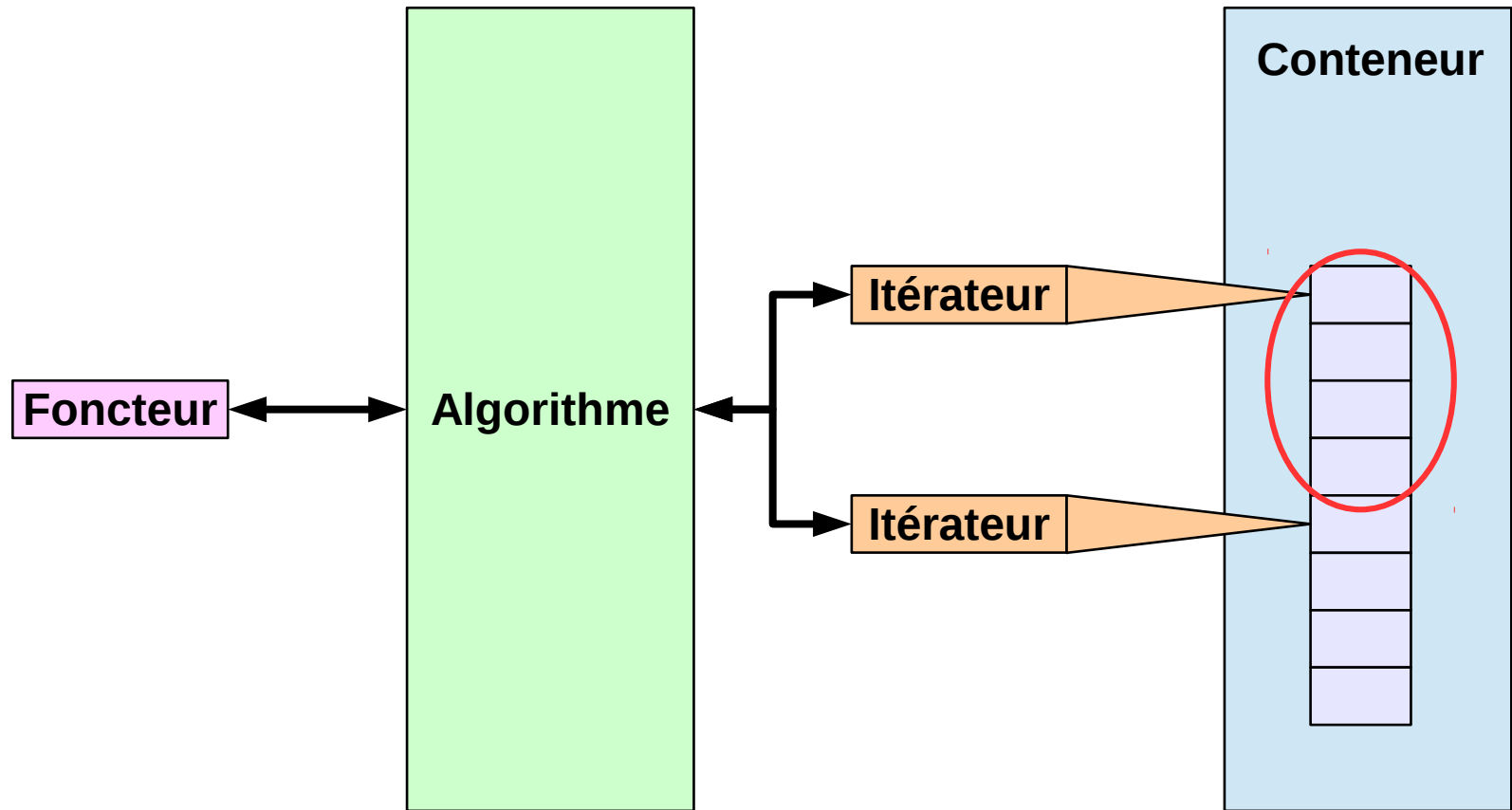
- Il y a les algorithmes de modification de données:
  - Ils permettent de transformer les données
    - `transform(it1, it2, itS, foncteur unaire)`
      - Passe les valeurs de l'intervalle dans le foncteur unaire et place le résultat dans itS
    - `transform(it1, it2, it3, itS, foncteur binaire)`
      - Passe au foncteur binaire les valeurs des deux intervalles d'entrée (`[it1, it2[`, et `[it3, ...[`) et place le résultat dans itS

# Et qu'en est-il des algorithmes?

---

- Et enfin, les opérations arithmétiques généralisées:
  - **Ce sont des algorithmes optimisés pour les calculs mathématiques**
  - `iota(it1, it2, n)`: construit une suite  $n, n+1, n+2, \dots$  sur l'intervalle donné
  - `inner_product(it1, it2, it3, n)`: calcule le produit scalaire entre les deux intervalles
  - `accumulate(it1, it2, n)`: somme des éléments de l'intervalle +  $n$
  - `partial_sum(it1, it2, itS)`: envoie les sommes partielles de l'intervalle dans `itS`
  - `adjacent_difference(it1, it2, itS)`: retourne la différence entre un élément et son précédent

# Récapitulatif: interaction des éléments de la STL



# Et... comment la STL peut-elle nous aider avec notre application?

- Modifions `getEmployee(string name)`:

**Rappel:** `employees_` est un `vector<Employee*>`

```
Employee* Company::getEmployee(string name) const {  
    // Basic search with for each loop  
    for (Employee* employee : employees_) {  
        if (employee->Employee::getName() == name) {  
            return employee;  
        }  
    }  
    return 0;  
}
```

- 1) La refaire en utilisant des itérateurs
- 2) La refaire en utilisant `find_if`

# Et... comment la STL peut-elle nous aider avec notre application?

- Modifions `delEmployee(Employee* employee)`:

```
void Company::delEmployee(Employee* employee) {  
    for (int i = 0; i < employees_.size(); i++) {  
        if (employees_[i] == employee) {  
            for (j = i; j < employees_.size() - 1; j++)  
                employees[j] = employees[j + 1];  
  
            employees_.pop_back();  
            break;  
        }  
    }  
}
```

**Indice:** vous pouvez utiliser la méthode `erase` des conteneurs qui prend en paramètre un itérateur!

- 1) La refaire en utilisant des itérateurs
- 2) La refaire en utilisant `find`

## **Et... comment la STL peut-elle nous aider avec notre application?**

---

- Créer un conteneur permettant de trouver rapidement un employé grâce à son nom qui sera retourné par `getEmployeesPerName()`
- 1) De façon procédurale (boucle for each)
- 2) Avec l'algorithme `for_each`