Chapitre 5
Tableaux

Buts du chapitre 5

- Apprendre le formalisme des **tableaux classiques**[1] **à une dimension**[2] (1D) pour leur déclaration, leur initialisation, l'accès à leurs éléments et leur passage en paramètre

  [1] Les tableaux classiques sont ceux «hérités» du langage C

  [2] Les tableaux classiques à plusieurs dimensions seront étudiés dans le cours PRG2

- Étudier divers **algorithmes simples** sur les tableaux classiques 1D
- Différencier les notions de taille et de capacité sur des tableaux classiques 1D partiellement remplis
- Introduire la classe `vector` permettant de disposer de tableaux redimensionnables
- Introduire la classe `array` permettant de disposer de tableaux non redimensionnables
- Étudier quelques **algorithmes de tri** simples
- Introduire `<algorithm>` et `<numeric>` de la librairie standard, y compris les bases de la notion d'itérateur

Plan du chapitre 5

1. Tableaux classiques 1D : concepts de base

Exemple de problème

- Énoncé
  - À partir des notes de 10 élèves, calculer le nombre d'élèves qui ont obtenu une note plus élevée que la moyenne
- Algorithme de résolution
  - Lire les 10 notes
  - Calculer la moyenne
  - Comparer chacune des 10 notes à la moyenne
- Difficulté
  - On ne dispose de la moyenne qu'après avoir lu la 10^{e} note. On ne peut donc pas effectuer les comparaisons au fur et à mesure. Il faut stocker les 10 valeurs. Comment les stocker ?
  - *Réponse* : dans un **tableau**

Solution du problème

```
const int N = 10; double moyenne, somme = 0.0; int cnt = 0;
double notes[N];

for (int i = 0; i < N; ++i) {
   cout << "Donnez la note # " << i + 1 << " : ";
   cin >> notes[i];
}
for (int i = 0; i < N; ++i) {
   somme += notes[i];
}
moyenne = somme / N;
cout << "Moyenne de la classe : " << moyenne << endl;
for (int i = 0; i < N; ++i) {
   if (notes[i] > moyenne)
       cnt++;
}
cout << cnt << " eleves ont plus que la moyenne" << endl;
```

Déclaration du tableau notes

Écriture dans le i^{ième} élément

Lecture du i^{ième} élément

Déclaration - syntaxe

```
classe type ident[expr], …;
```

`classe`	La classe d'allocation (`extern, static`). Si on l'omet, la variable est par défaut automatique.
`type`	Le type des données stockées (`int, double, string,…`)
`ident`	Le nom du tableau
`expr`	Le nombre d'éléments. L'expression doit être évaluable à la compilation.

Déclaration - exemples

- Avec `N` de type `const int`, les déclarations suivantes sont valides

```
double tab1[N];
static char tab2[N*2];
string tab3[20];
```

- Pour un tableau extern, il est inutile de déclarer sa taille (parce que le compilateur n'en tiendra pas compte)

```
extern int tab[];
```

- On peut également déclarer un tableau constant avec le mot clé const, mais cela requiert de l'initialiser lors de la déclaration (*voir slide 12*)
- Certains compilateurs (dont `g++` si on ne spécifie pas l'option `-Wvla`) ne suivent pas la norme et acceptent des tailles variables pour spécifier le nombre d'éléments. À éviter si on veut produire du code portable !

Accès aux éléments

- On accède aux éléments d'un tableau **`tab`** avec l'opérateur crochet, **`tab[i]`**, qui prend l'indice **`i`** de l'élément en paramètre.
- Un indice peut être n'importe quelle expression arithmétique d'un type entier (`char`, `short`, `int`, `long`, `long long`) signé ou pas.
- **Les indices d'un tableau de `N` éléments vont de `0` à `N-1`**
  - Un élément de tableau est une *lvalue*.
  - On peut donc lui affecter une valeur, l'incrémenter, le décrémenter, …

```
int i = 0, j = 1;
int tab[2];
tab[0] = 5;
tab[j] = 1;
tab[i + j]++;
--tab[(i + 5) % 2];
```

Attention !

- Par contre, le tableau lui-même n'est pas une *lvalue*.
  C++ n'offre pas d'affectation globale de tableaux.

```
int tab1[2], tab2[2];
tab1 = tab2;
```

→ `Error: Array type 'int [2]' is not assignable`

- Notons également que l'accès aux éléments se fait **sans contrôle de débordement d'indice**.

- Accéder à un élément d'indice plus petit que 0 ou plus grand ou égal au nombre d'éléments à un effet indéterminé, mais en général néfaste.

Initialisation

- Un tableau peut être initialisé lors de sa déclaration par un agrégat, i.e. entre accolades

```
int tab[5] = { 10, 42, 7, -2, 1 };
```

10	42	7	-2	1

- On peut n'indiquer que les premières valeurs. Les valeurs manquantes sont initialisées à zéro

```
int tab[5] = { 10, 42, 7 };
```

10	42	7	0	0

- On peut même n'indiquer aucune valeur dans l'agrégat, ce qui initialise tout le tableau à zéro, sans qu'il soit nécessaire d'écrire une boucle

```
int tab[5] = { };
```

0	0	0	0	0

- Par contre, si l'agrégat a plus de valeurs que la taille déclarée du tableau, le compilateur signale une erreur.

Initialisation

- Si l'on initialise toutes les valeurs du tableau, on peut omettre la taille de celui-ci, qui sera déduite par le compilateur

```
int tab[] = { 10, 42, 7, -2, 1 };
```

10	42	7	-2	1

- L'initialisation nous permet également de déclarer des tableaux constants

```
const char VOYELLES[] = { 'a', 'e', 'i', 'o', 'u', 'y' };
```

- Notons que, comme pour les variables simples, un tableau qui n'est pas explicitement initialisé aura un contenu
  - indéterminé s'il est alloué automatiquement ou dynamiquement
  - initialisé à zéro s'il est alloué statiquement

Tableaux en paramètre

- On peut passer un tableau en paramètre d'une fonction, en indiquant `[]` dans le prototype après le nom du paramètre

```
void f(int tab[]);
```

- On appelle cette fonction ainsi

```
int tab[5] = { 10, 42, 7, -2, 1 };
f(tab);
```

- Notons que l'on ne spécifie pas la taille du tableau passé en paramètre.
  On pourrait la spécifier, mais le compilateur n'en tient pas compte.
  Il reste possible de lui passer des tableaux d'une autre taille :

```
void f(int tab[8]);
```

Tableaux en paramètre

- La taille du tableau n'étant pas connue, il est en général nécessaire de la transmettre (*elle ne peut être déterminée à l'exécution à partir de tab*)

```
void f(int tab[], size_t taille);
```

- On peut modifier le contenu d'un tableau passé en paramètre, comme pour une variable passée par référence

```
void f(int tab[], size_t taille) {
   for (size_t i = 0; i < taille; ++i)
      tab[i] = i;  // par exemple
}
```

```
int tab[5];
f(tab, 5);
for (size_t i = 0; i < 5; ++i)
   cout << tab[i] << " ";
```

```
0 1 2 3 4
```

Tableaux en paramètre

- Si l'on ne veut pas en modifier le contenu, il convient de le déclarer dans le prototype avec le mot-clé `const`.

```
void afficher(const int tab[], size_t taille) {
   for (size_t i = 0; i < taille; ++i)
      cout << tab[i] << " ";
   cout << endl;
}
```

```
const int TAB[5] = { 10, 42, 7, -2, 1 };
afficher(TAB, 5);
```

- Cela permet également d'appeler la fonction en lui transmettant un tableau constant.

Passage en retour de fonction

- Par contre, il n'est pas possible de retourner un tableau

```
int[3] f();
```

```
Error: Function cannot return array type 'int [3]'
```

- Une fonction peut donc modifier un tableau qu'elle a reçu en paramètre, mais pas en créer un qu'elle fournirait en résultat.

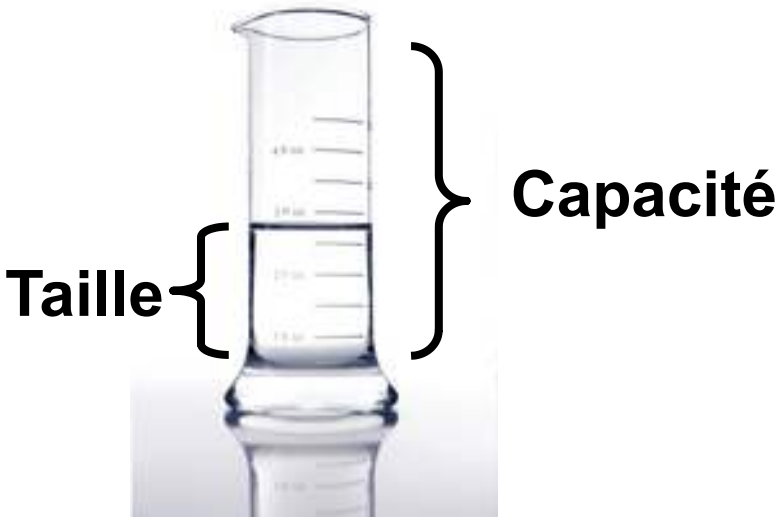
Note sur la mise en œuvre pratique

- Quand le compilateur alloue un tableau de N éléments, il réserve en mémoire N emplacements consécutifs pour y stocker les N éléments.
- Le nom du tableau lui permet de connaître l'emplacement du premier élément en mémoire.
- L'indice d'un élément lui permet de calculer un décalage par rapport à ce premier élément pour trouver l'élément indicé.
- *Ces éléments seront approfondis dans le cours PRG2*

2. Tableaux classiques 1D : quelques algorithmes simples

Tableaux partiellement remplis

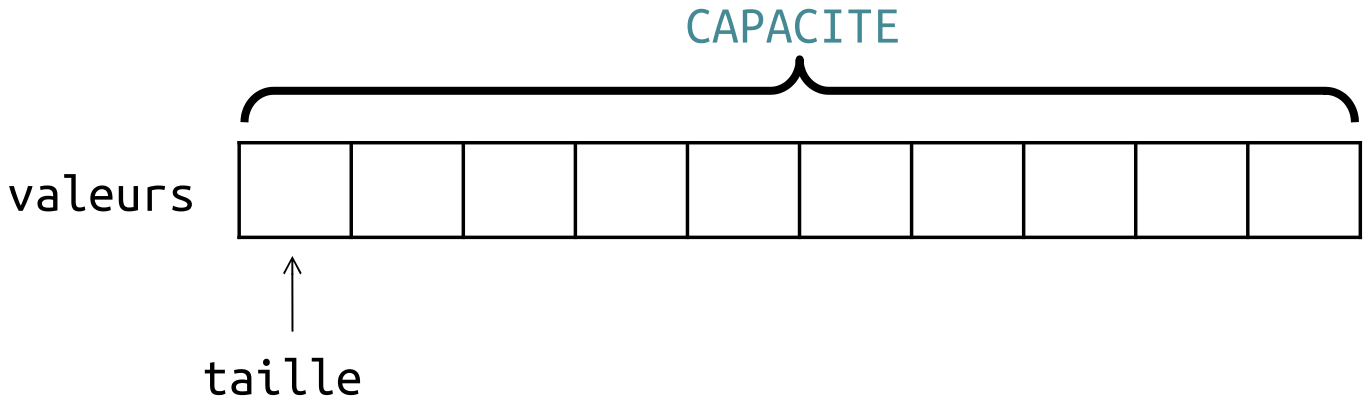
- Un tableau qui peut contenir 10 éléments n'en contient pas nécessairement autant
- On distingue :
  - **la capacité** – le nombre maximum d'éléments qu'un tableau peut contenir. La quantité de mémoire réservée.
  - **la taille** – le nombre d'éléments réellement présents



Taille et capacité

- La mise en œuvre en C++ se fait en utilisant une variable compagne pour stocker le nombre d'éléments effectivement présents dans un tableau classique 1D

```
const size_t CAPACITE = 10;
double valeurs[CAPACITE];
size_t taille = 0; // tableau vide
```

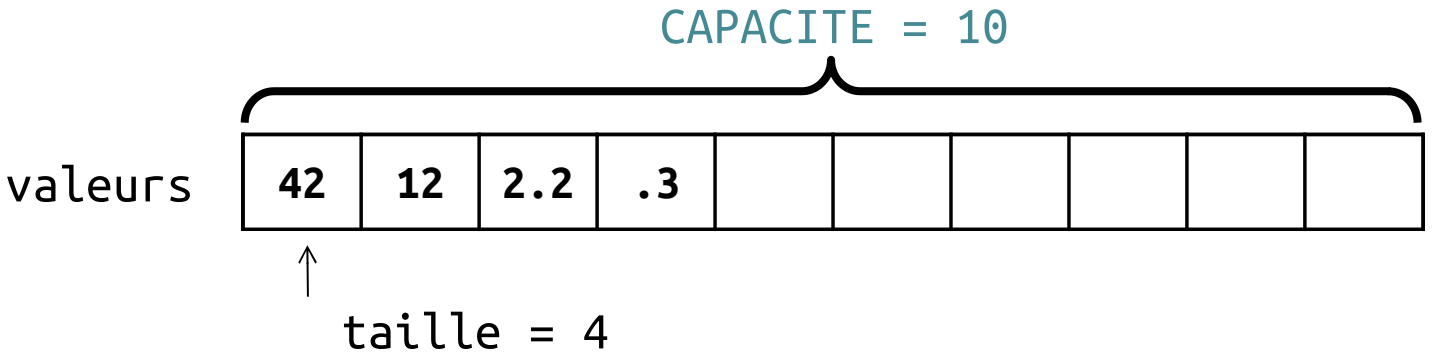


Tableaux partiellement remplis

- Entrer des données dans un tel tableau se fait en modifiant simultanément le contenu de `valeurs` et la valeur de `taille`

```
double entree;
while (cin >> entree) {
   if (taille < CAPACITE) {
      valeurs[taille] = entree;
      ++taille;
   }
}
```

```
42
12
2.2
.3
q
```



Remplir

- Pour remplir un tableau, il suffit de le parcourir avec une boucle `for` pour les indices allant de `0` *compris* à `taille` *non compris*

```
/**
 *  @param tab    tableau à remplir
 *  @param taille nombre d'éléments à remplir
 *  @param val    valeur à utiliser pour remplir
 */
void remplir(double tab[], size_t taille, double val) {
   for (size_t i = 0; i < taille; ++i)
      tab[i] = val;
}
```

La taille est positive ou nulle

Un tableau passé en paramètre est modifiable

L'indice du dernier élément est `taille-1`

Copier

- Pour copier un tableau dans un autre, on peut parcourir les deux tableaux avec le même indice
- Il faut évidemment que la capacité de `tabOut` soit suffisante

```
/**
 *  @param tabIn  tableau contenant les éléments à copier
 *  @param tabOut emplacement où les éléments sont copiés
 *  @param taille nombre d'éléments à copier
 */
void copier(const double tabIn[], double tabOut[], size_t taille) {
   for (size_t i = 0; i < taille; ++i)
      tabOut[i] = tabIn[i];
}
```

`tabIn` n'est pas modifié par la fonction

Somme

- On calcule la somme des éléments en parcourant le tableau et en accumulant les valeurs dans une variable

```
/**
 *  @param tab    tableau contenant les éléments à sommer
 *  @param taille nombre d'éléments à sommer
 *  @return la somme des éléments
 */
double somme(const double tab[], size_t taille) {
   double somme = 0;
   for (size_t i = 0; i < taille; ++i)
      somme += tab[i];
   return somme;
}
```

On retourne la valeur de la somme

`tab` n'est pas modifié par la fonction

Moyenne

- La somme permet immédiatement de calculer la moyenne

```
/**
 *  @param tab    tableau contenant les éléments à moyenner
 *  @param taille nombre d'éléments à moyenner
 *  @return moyenne des éléments. 0 si pas d'éléments
 */
double moyenne(const double tab[], size_t taille) {
   return taille ? (somme(tab, taille) / taille) : 0.0;
}
```

Attention à toujours se méfier du cas d'un tableau vide

Minimum (1)

- Pour rechercher le minimum, il suffit de parcourir le tableau et de comparer le minimum trouvé jusque-là à l'élément courant

```
/**
 *  @param tab    tableau dans lequel on cherche
 *  @param taille nombre d'éléments du tableau
 *  @return la valeur minimale du tableau
 */
double minimum(const double tab[], size_t taille) {
   double minimum = tab[0]; // on suppose taille > 0
   for (size_t i = 1; i < taille; ++i) {
      if (tab[i] < minimum)
         minimum = tab[i];
   }
   return minimum;
}
```

On initialise avec la valeur de l'élément en position 0

Et donc on parcourt le reste du tableau à partir de l'élément 1

Position du minimum

- De manière plus générale, on peut chercher la position de l'élément minimum (*la première, si cette valeur apparaît plusieurs fois*)

```
/**
 *  @param tab    tableau dans lequel on cherche
 *  @param taille nombre d'éléments du tableau
 *  @return la position de la valeur minimale du tableau
 */
size_t element_min(const double tab[], size_t taille) {
   size_t pos = 0; // on suppose taille > 0
   for (size_t i = 1; i < taille; ++i) {
      if (tab[i] < tab[pos])
         pos = i;
   }
   return pos;
}
```

tab[pos] est la valeur courante du minimum

On retourne une position, donc de type size_t

Minimum (2)

- Il est alors pertinent de simplifier le code de notre fonction `minimum`

```
/**
 *  @param tab    tableau dans lequel on cherche
 *  @param taille nombre d'éléments du tableau
 *  @return la valeur minimale du tableau
 */
double minimum(const double tab[], size_t taille) {
   // on suppose taille > 0
   return tab[element_min(tab, taille)];
}
```

Recherche linéaire

```c
/**
 *  @param tab    tableau dans lequel on cherche
 *  @param taille nombre d'éléments du tableau
 *  @param val    valeur recherchée
 *  @return position du premier élément trouvé.
 *          retourne taille si la recherche est infructeuse
 */
size_t chercher(const int tab[], size_t taille, int val) {
   size_t pos = 0;
   while (pos < taille) {
      if (tab[pos] == val)
         break;
      ++pos;
   }
   return pos;
}
```

On arrête le parcours dès que l'on a trouvé `val`

On retourne une position, donc de type size_t

Sans break

- Le même sans utiliser de break

```
size_t chercher(const int tab[], size_t taille, int val) {
   size_t pos = 0;
   bool trouve = false;
   while (pos < taille and not trouve) {
      if (tab[pos] == val)
         trouve = true;
      else
         ++pos;
   }
   return pos;
}
```

Sans break

- Ou encore…

```
size_t chercher(const int tab[], size_t taille, int val) {
   size_t pos = 0;
   for ( ; pos < taille; ++pos) {
      if (tab[pos] == val)
         return pos;
   }
   return taille;
}
```

Encore mieux …

- Mettre la position de départ en paramètre permet de rechercher les éléments au-delà du premier

```
size_t chercher(const int tab[], size_t taille, int val, size_t pos = 0) {
   for ( ; pos < taille; ++pos) {
      if (tab[pos] == val)
         return pos;
   }
   return taille;
}
```

Recherche multiple

- Par exemple, pour trouver toutes les positions des éléments de valeur 2

```
size_t chercher(const int tab[], size_t taille, int val, size_t pos = 0);

int main() {

   int tab[] = { 2, 1, 2, 3, 2, 5, 4, 2, 1, 7, 2 };
   const size_t CAPACITE = sizeof(tab) / sizeof(int);
   size_t taille = CAPACITE; // le tableau est plein

   size_t pos = 0;
   while ( (pos = chercher(tab, taille, 2, pos)) != taille ) {
      cout << pos << ' ';
      ++pos; // chercher à partir de la position suivant celle trouvée
   }
}
```

```
0 2 4 7 10
```

Effacer un élément, ordre modifié

- Effacer un élément modifie la taille du tableau
- Si l'ordre importe peu, il suffit de copier le dernier élément à la position à supprimer, puis diminuer la taille

```
/**
 *  @param[in,out] tab     tableau à modifier
 *  @param[in,out] taille  nombre d'éléments.
 *  @param[in]     pos     position de l'élément à supprimer
 */
void supprimer(double tab[], size_t& taille, size_t pos) {
   if (taille and pos < taille) {
      tab[pos] = tab[taille - 1];
      --taille;
   }
}
```

La taille est modifiée

vérification des bornes et de taille non nulle

Dernier élément du tableau avant suppression. Sera l'emplacement supprimé

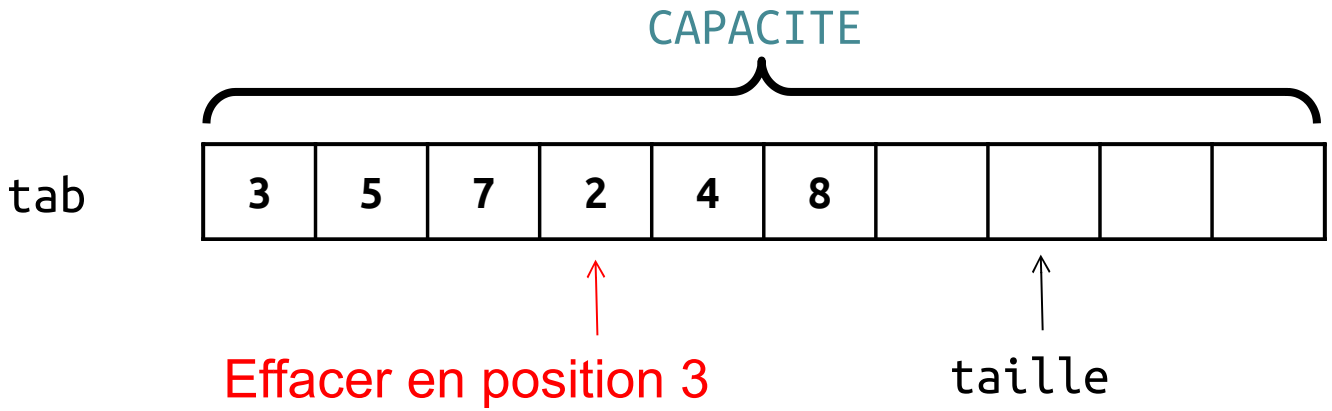
Effacer un élément, ordre modifié

- Notons que le test (`taille != 0`) est inutile puisqu'inclus dans le test (`pos < taille`)
- En utilisant la décrémentation préfixe, le code est très compact

```
void supprimer(double tab[], size_t& taille, size_t pos) {
   if (pos < taille)
      tab[pos] = tab[--taille];
}
```

Effacer un élément, ordre conservé

- Si l'ordre est important, il faut décaler d'une position vers la gauche tous les éléments dont la position est supérieure à celle de l'élément à supprimer



- Décaler position `4 -> 3`
- Décaler position `5 -> 4`
- Décaler position `6 -> 5`
- Décrémenter `taille 7 -> 6`

Effacer un élément, ordre conservé

```
/**
 *  @param[in,out] tab     tableau à modifier
 *  @param[in,out] taille  nombre d'éléments.
 *  @param[in]     pos     position de l'élément à supprimer
 */
void supprimerOrdonne(double tab[], size_t& taille, size_t pos) {
   if (pos < taille) {
      for (size_t i = pos + 1; i < taille; ++i) {
         tab[i - 1] = tab[i];
      }
      --taille;
   }
}
```

Insérer un élément en fin de tableau

- Insérer un élément modifie la taille du tableau.
- Si l'ordre importe peu, il suffit d'insérer en dernière position, puis d'augmenter la taille.

```
/**
 *  @param[in,out] tab      tableau à modifier
 *  @param[in,out] taille   nombre d'éléments.
 *  @param[in]     capacite capacité de tab
 *  @param[in]     val      valeur à insérer en queue
 */
void inserer(double tab[], size_t& taille, size_t capacite, double val) {
   if (taille < capacite) {
      tab[taille] = val;
      ++taille;
   }
}
```

Attention, il faut vérifier si la capacité est suffisante

Insérer un élément en position quelconque

- Si la position d'insertion importe, il faut décaler d'une position vers la droite tous les éléments dont la position est supérieure à celle de l'élément à insérer, afin de pouvoir libérer une place.

```
/** @param[in,out] tab      tableau à modifier
 *  @param[in,out] taille   nombre d'éléments.
 *  @param[in]     capacite capacité de tab
 *  @param[in]     pos      position d'insertion
 *  @param[in]     val      valeur à insérer
 */
void insererOrdonne(double tab[], size_t& taille, size_t capacite,
                    size_t pos, double val) {
   if (taille < capacite and pos <= taille) {
      for (size_t i = taille; i > pos; --i)
         tab[i] = tab[i - 1];
      tab[pos] = val;
      ++taille;
   }
}
```

Insérer un élément en position quelconque

- `(taille < capacite)` vérifie qu'il reste de la place pour insérer l'élément
- `(pos <= taille)` vérifie que `pos` est une des `taille + 1` positions d'insertion valides
- Attention, la boucle apparemment équivalente

```
for (size_t i = taille - 1; i >= pos; --i)
   tab[i + 1] = tab[i];
```

est une boucle infinie pour (`pos == 0`).
En effet, `(i >= 0)` est toujours vrai quand `i` est de type size_t

Limitations des tableaux classiques

- La capacité est constante et doit être décidée avant la compilation
- Il faut utiliser une variable (constante) annexe pour stocker la taille (la capacité) d'un tableau
- On ne peut pas retourner de tableau depuis une fonction
- Il n'y a pas de contrôle de bornes lors de l'accès à un élément du tableau

Toutes ces limitations disparaissent en utilisant la classe `vector`

3. Introduction aux classes et à la généricité

Buts de cette partie

- Introduire la notion de **généricité** avant de présenter les classes génériques `vector` et `array` et la librairie de fonctions génériques `<algorithm>`
- Introduire les notions et formalismes communs aux **classes** avant de présenter les classes `vector` et `array` en détail (puis `string` au chapitre 6)
- Comprendre ces notions du point de vue de l'utilisateur de ces classes, et pas encore du point de vue de leur concepteur (ce qui sera fait au chapitre 7)

Utiliser des fonctions génériques

- Il est possible d'écrire des fonctions valables pour plusieurs types de paramètres
- Le type de certains paramètres n'est pas spécifié. Il est déterminé selon le type spécifié par le code appelant.
- Par exemple, la fonction cos de `<cmath>` est surchargée pour les types `float`, `double`, `long double`, mais également de manière générique.

function

cos

C90 | C99 | C++98 | C++11

```
     double cos (double x);
      float cos (float x);
long double cos (long double x);
     double cos (T x);           // additional overloads for integral types
```

Utiliser du code générique

- On peut spécifier quelle version de la fonction générique est appelée en précisant le `<type>` après le nom de la fonction
  - par exemple pour un paramètre entier :

```
double x = cos<int>(1);
```

- Le compilateur est aussi capable de déduire quelle fonction générique est appelée en fonction des types des paramètres fournis

```
double x = cos(1U); // appelle cos<unsigned int>
```

Une classe est un type composé

- Les types simples (booléens, entiers, réels, ...) sont caractérisés par
  - Les données qu'ils stockent, y compris leur représentation
  - Les opérations que l'on peut effectuer sur ces données
- Les types composés (classes) étendent ce concept en créant des types définis par le développeur (ou prédéfinis dans une librairie).
  - Quelles données sont stockées ?
  - Comment les initialiser ?
  - Quels opérateurs sont définis ?
  - En plus des opérateurs, comment utiliser les fonctions membres (méthodes) ?

Un objet est une instance de ce type

- L'emporte-pièce est la classe
- Les biscuits sont les objets

```
string s1("Hello");
string s2 = "World!";
string s3 = s1 + ", " + s2;
string s4 = s3;
```

- `string` est le nom de la classe
- `s1`, `s2`, `s3` et `s4` sont des objets de type `string`

Données

- En général, les données stockées dans les objets d'une classe sont plus complexes que pour les types simples. Par exemple :
  - `std::complex` stocke deux réels, pour les parties réelles et imaginaires
  - `std::string` stocke un tableau de `char` de taille variable (alloué dynamiquement), sa taille et sa capacité
  - `std::vector` stocke un tableau d'éléments de taille variable (alloué dynamiquement), sa taille et sa capacité
- Dans une classe respectant strictement le principe d'encapsulation, l'utilisateur n'a pas directement accès à ces données.

Classes génériques

- Comme pour les fonctions, les classes peuvent être écrites de manière générique. C'est le cas de `complex` et de `vector`

```
complex<float> cf;   // complex utilisant float pour stocker les composantes R/I
complex<double> cd;  // complex utilisant double pour stocker les composantes R/I

vector<int> vi;      // vector d'éléments int
vector<char> vc;     // vector d'éléments char
```

- Pour les classes, il faut toujours spécifier le type effectif entre <>, contrairement aux fonctions. Le compilateur n'essaie pas de le deviner.
- Nous l'avons déjà rencontré avec `numeric_limits`

Initialisation

- Les données étant plus complexes, l'initialisation l'est aussi.
  Par exemple, pour créer un `complex`, on peut écrire :

```
complex<double> c1;        // par défaut (0, 0)
complex<double> c2 = c1;   // par copie
complex<double> c3(c1);    // par copie
complex<double> c4(1, 1); // en spécifiant réel / imag.
complex<double> c5(1);     // en spécifiant réel (imag = 0)
```

- Par rapport aux types simples, notons que
  - `c1` est initialisé à (0, 0)
  - `c4` et `c5` utilisent un constructeur d'initialisation

public member function

std::complex::complex

initialization (1) `complex (const T& re = T(), const T& im = T());`

Constructeurs

- Une classe comme `string` offre de très nombreuses méthodes d'initialisation via ses constructeurs. Nous y reviendrons au chapitre 6.

public member function

std::string::string

C++98 | C++11

```
default (1)          string();
copy (2)             string (const string& str);
substring (3)        string (const string& str, size_t pos, size_t len = npos);
from c-string (4)    string (const char* s);
from buffer (5)      string (const char* s, size_t n);
fill (6)             string (size_t n, char c);
range (7)            template <class InputIterator>
                       string  (InputIterator first, InputIterator last);
initializer list (8) string (initializer_list<char> il);
move (9)             string (string&& str) noexcept;
```

Construct string object

Constructs a string object, initializing its value depending on the constructor version used:

Opérateurs

- Certaines classes définissent les mêmes opérateurs que pour les réels.
  C'est le cas de `complex`. On peut écrire :

```
complex<double> c1(1, 1);
complex<double> c2 = c1 + c1;
c2 *= c1;
cout << c1 << ' ' << c2 << endl;
```

```
(1,1) (0,4)
```

- D'autres classes ne définissent que peu ou pas d'opérateurs.

```
string::operator+=
string::operator=
string::operator[]
operator+ (string)
operator<< (string)
operator>> (string)
```

public member function

std::string::operator+=

C++98 | C++11

```
string (1)      string& operator+= (const string& str);
c-string (2)    string& operator+= (const char* s);
character (3)   string& operator+= (char c);
```

Méthodes

- Certaines manipulations des données n'ont pas d'équivalent logique dans les symboles d'opérateurs existants, ou nécessitent plus de paramètres
- Ces manipulations sont alors disponibles sous forme de **fonctions membres ou « méthodes »**. Par exemple, string permet d'extraire une sous chaîne avec `substr`

public member function

std::string::substr

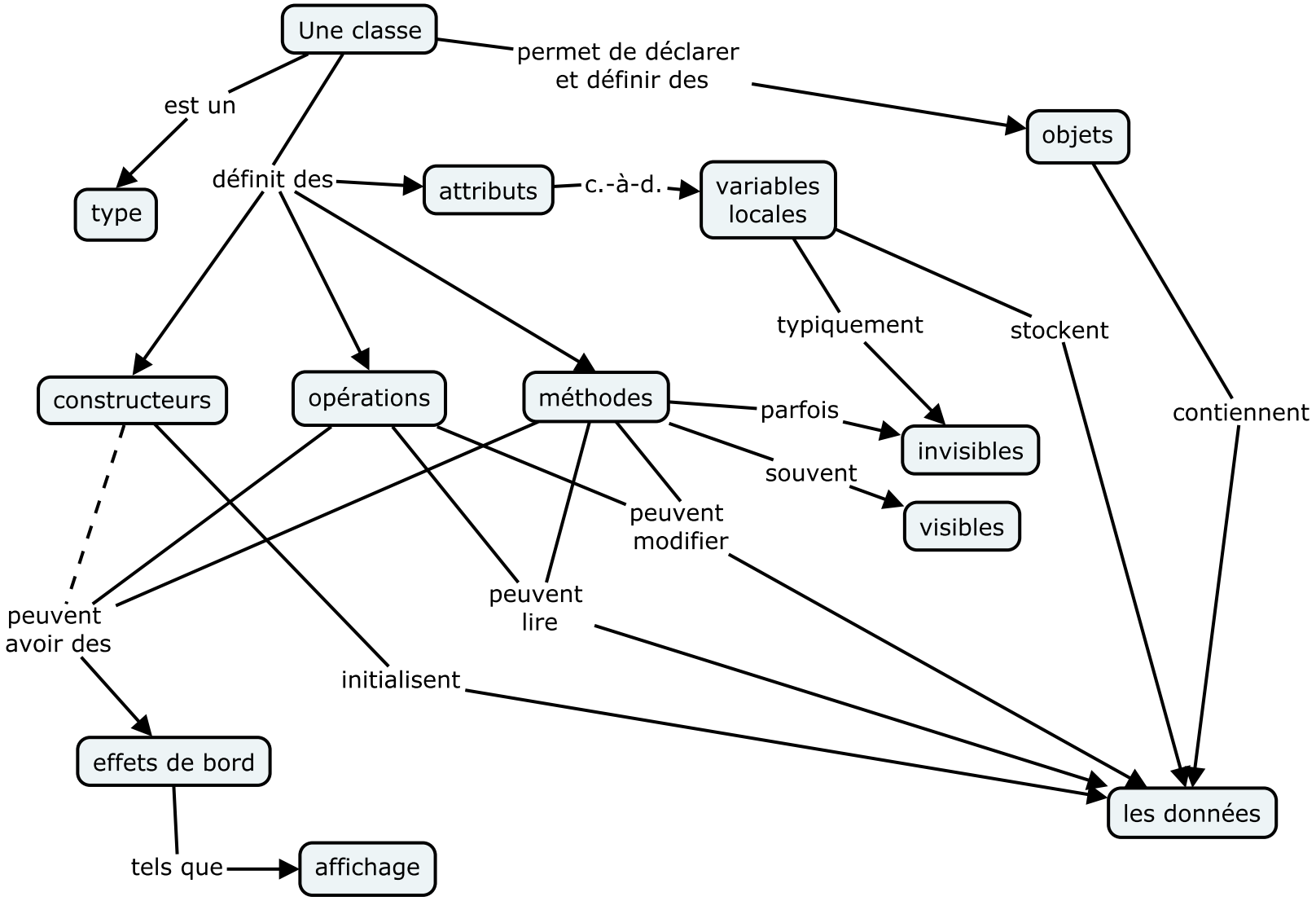
```
string substr (size_t pos = 0, size_t len = npos) const;
```

- Pour l'appeler, on écrit le nom de l'objet, suivi d'un point, du nom de la méthode, et des paramètres entre parenthèses.

```
string s1("informatique");
string s2 = s1.substr(2, 6);
```

format

En résumé


Une classe
permet de déclarer et définir des
est un
objets
définit des
attributs
c.-à-d.
variables locales
type
typiquement
stockent
constructeurs
opérations
méthodes
parfois
contiennent
invisibles
souvent
visibles
peuvent modifier
peuvent lire
peuvent avoir des
initialisent
effets de bord
les données
tels que
affichage

4. La classe vector

Conteneurs

- La bibliothèque standard propose des alternatives aux tableaux classiques pour stocker des données multiples : les **conteneurs**
- Leur étude détaillée se fera dans le cours ASD
- Ici, nous étudions les conteneurs `vector` et `array`

Container class templates

Sequence containers:

array C++11	Array class (class template)
vector	Vector (class template)
deque	Double ended queue (class template)
forward_list C++11	Forward list (class template)
list	List (class template)

Container adaptors:

stack	LIFO stack (class template)
queue	FIFO queue (class template)
priority_queue	Priority queue (class template)

Associative containers:

set	Set (class template)
multiset	Multiple-key set (class template)
map	Map (class template)
multimap	Multiple-key map (class template)

Unordered associative containers:

unordered_set C++11	Unordered Set (class template)
unordered_multiset C++11	Unordered Multiset (class template)
unordered_map C++11	Unordered Map (class template)
unordered_multimap C++11	Unordered Multimap (class template)

std::vector

- `std::vector` est un conteneur séquentiel qui encapsule les **tableaux de taille dynamique**.
- Les éléments sont stockés en mémoire de façon contigüe, comme pour les tableaux classiques. L'accès aux éléments est donc similaire.
- Le stockage du `vector` est pris en charge automatiquement : sa *capacité* peut être augmentée ou diminuée au besoin.
- Pour des raisons d'efficacité, taille et capacité peuvent différer.
  Ils prennent donc souvent plus de place en mémoire que les tableaux classiques.

Déclaration

- Pour utiliser un `vector`, il faut inclure l'en-tête de la classe

```
#include <vector>
```

- La déclaration d'un objet `vector` est typiquement de la forme :

```
vector<type> ident(taille, valeur);
```

`type`	le type des données stockées (`int`, `double`, `string`, voire même `vector`,…)
`ident`	Le nom du tableau
`taille`	Le nombre d'éléments, par défaut 0
`valeur`	La valeur initiale commune de ces éléments, par défaut 0

Déclaration

- Les déclarations suivantes sont donc valides :

```
vector<int>    v1;              // 0 élément int

vector<int>    v2(3);           // 3 éléments int valant 0

vector<int>    v3(5, 7);        // 5 éléments int valant 7

vector<double> v4(9);           // 9 éléments double valant 0

vector<string> v5(4, "Hello"); // 4 éléments string valant "Hello"

vector<string> v6(6);           // 6 éléments string valant ""
```

Constructeurs

- Les déclarations précédentes correspondent au **constructeur par défaut** (sans paramètre) et au **constructeur de remplissage**
- Il existe encore d'autres constructeurs permettant d'initialiser de façon plus élaborée les valeurs d'un `vector`.
  - Le **constructeur de copie** qui permet de copier la taille et les valeurs d'un autre `vector`
  - Le **constructeur de *range***, qui permet de recopier une séquence d'éléments venant d'un autre conteneur ou d'un tableau classique.
- Et depuis C++11 :
  - Le **constructeur avec liste d'initialisation**
  - Le **constructeur de déplacement** (pas étudié ici)

Constructeur de copie

- Un `vector` peut être construit, initialisé ou affecté en copiant la taille (mais pas la capacité) et la valeur des éléments d'un autre `vector`

```
vector<int> v1(3, 7); // 3 éléments int valant 7
vector<int> v2(4);    // 4 éléments int valant 0

vector<int> v3(v1);   // copie v1, 3 éléments valant 7

vector<int> v4 = v2;  // copie v2, 4 éléments valant 0

v4 = v1;              // maintenant, v4 est une copie de v1,
                      // donc 3 éléments valant 7
```

- Attention !
  - `vector<int> v4 = v2;` ☾ appel du constructeur de copie
  - `v4 = v1;` ☾ appel de l'opérateur d'affectation

Constructeur de *range*

- Un `vector` peut être construit à partir d'une séquence d'éléments provenant d'un autre conteneur ou d'un tableau classique.
- Il reçoit deux paramètres : un itérateur vers le premier élément et un autre vers l'élément suivant le dernier.

```
// à partir d'un tableau classique
int tab[4] = {2, 3, 5, 7};
vector<int> v1(tab, tab + 4);

// à partir d'un autre vector
vector<int> v2(v1.begin(), v1.end());

// à partir d'un autre conteneur
list<int> liste{1, 2, 3};
vector<int> v3(liste.begin(), liste.end());
```

désigne l'adresse de début de `tab` en mémoire

voir la suite de ce chapitre

voir le cours ASD

Constructeur avec liste d'initialisation

- Enfin, depuis C++11, un `vector` peut être construit et initialisé grâce à une liste d'initialisation. Les trois syntaxes suivantes sont possibles :

```
vector<int> v1{1, 2, 3};
vector<int> v2({1, 2, 3});
vector<int> v3 = {1, 2, 3};
```

- Notons que contrairement aux tableaux classiques, on ne peut pas mélanger les syntaxes. Le code suivant ne compile pas :

```
vector<int> v(3) = {1, 2, 3};
```

```
error: expected ',' or ';' before '=' token
```

Passage en paramètre et en retour

- Enfin, on peut avoir accès à un `vector` comme paramètre d'une fonction.
  - Le passage en paramètre se comporte exactement comme pour toute autre variable simple (`int`, `double`,…) ou composée (`string`,...)
  - Comme pour `string`, on préfère généralement le passage par référence ou par référence constante plutôt que par valeur, pour éviter des copies inutiles.
  - Le mécanisme est beaucoup plus simple que pour les tableaux classiques, car on ne doit pas se soucier de la taille ou de la capacité de l'objet `vector`
- Comme pour tout autre type, et contrairement aux tableaux classiques, on peut retourner un `vector` en valeur de retour d'une fonction.

operator[…], size()

- Comme pour les tableaux classiques, on peut accéder aux éléments d'un `vector` avec l'opérateur `[]`.
- `vector` propose une méthode `size()` qui retourne le nombre d'éléments, et simplifie le code. Plutôt que d'écrire :

```
int tab[] = {2, 3, 5, 7, 11};
for (size_t i = 0; i < 5; ++i)
   cout << tab[i] << " ";
```

on peut écrire :

```
vector<int> v {2, 3, 5, 7, 11};
for (size_t i = 0; i < v.size(); ++i)
   cout << v[i] << " ";
```

ce qui économise un nombre magique / une constante / un paramètre de fonction et rend le code portable

at(…)

- Comme pour les tableaux classiques, l'opérateur `[]` ne vérifie pas que l'indice passé en paramètre est licite.
- Accéder avec cet opérateur à un élément d'indice `< 0` ou `>= .size()` a un comportement indéfini.
- Il est plus sûr et plus simple de corriger un programme en accédant aux éléments avec la méthode `at()`. La boucle précédente se réécrit ainsi :

```
for (size_t i = 0; i < v.size(); ++i)
   cout << v.at(i) << " ";
```

  - `at()` vérifie que l'indice est compris dans les bornes
  - `at()` lance l'exception `std::out_of_range` si ce n'est pas le cas (*voir chap. 9*)

front(), back()

- On peut accéder directement au premier et au dernier élément avec les méthodes `front()` et `back()`.
- Pour un `vector` `v`,
  - `v.front()` est équivalent à `v[0]`
  - `v.back()` est équivalent à `v[v.size() - 1]`
- Comme pour l'opérateur `[]`, le comportement est indéfini en cas de débordement de bornes.
  - Mais il n'y a débordement que pour un `vector` vide.

for(:)

- Comme vu au chapitre 3, pour accéder à tous les éléments d'un conteneur v en C++11, on peut aussi utiliser une boucle `for` spéciale.
  La boucle précédente se réécrit :

```
for (int val : v)
   cout << val << " ";
```

où la variable `val` est une copie de `v[i]` à la `(i + 1)`ième itération de la boucle.

- Si l'on veut pouvoir modifier le contenu de v — par exemple mettre tous ses éléments au carré — il faut spécifier explicitement que `val` est une référence vers `v[i]` :

```
for (int& val : v)
   val *= val;
```

for(:) et les tableaux classiques

- Notons qu'il est aussi possible d'utiliser cette boucle `for` avec des tableaux classiques

```
int tab[] = {2, 3, 5, 7, 11};
for (int val : tab)
   cout << val << " ";
```

- Mais c'est uniquement possible à l'intérieur du bloc où le tableau a été déclaré. Si le tableau est passé en paramètre, cela ne compile pas.

```
void f(int tab[]) {
   for(int val : tab)
      cout << val << " ";
}
```

```
Error: Cannot build range expression
with array function parameter 'tab'
since parameter with array type 'int
[]' is treated as pointer type 'int
*'
```

iterator, begin(), end()

- Enfin, on peut accéder aux éléments de v via un **itérateur**.
  La boucle précédente s'écrit alors :

```
for (vector<int>::iterator i = v.begin(); i != v.end(); ++i)
        cout << *i << " ";
```

- La variable de boucle `i` est un itérateur sur un vecteur d'entiers, de type `vector<int>::iterator`
- `v.begin()` retourne un itérateur sur le premier élément de v
- `v.end()` retourne un itérateur sur un élément immédiatement après le dernier élément de v
- Incrémenter `i` le fait passer à l'élément suivant
- `*i` accède (en lvalue) à l'élément sur lequel itère `i`

itérateurs

- Pourquoi utiliser une syntaxe aussi complexe ?
- Si l'on utilisait seulement des `vector`, cela n'aurait en effet aucun intérêt.

  Mais…
- **Le concept d'itérateur permet de parcourir avec la même syntaxe tous les types de conteneurs de la STL.**
- **Cela permet de séparer les concepts de conteneur et d'algorithme.**
- Par exemple, la librairie `<algorithm>` — que l'on verra plus tard — définit des fonctions qui peuvent traiter les éléments de conteneurs divers, à condition qu'on leur fournisse les itérateurs idoines.

auto

- Depuis C++11, il est courant de simplifier cette syntaxe ainsi :

```
for (auto i = v.begin(); i != v.end(); ++i)
   cout << *i << " ";
```

- Le mot clé `auto` indique au compilateur d'inférer le type de la variable à partir du type de son initialisation.
  - `auto` peut être utilisé pour toute variable que vous initialisez, mais en abuser nuit à la lisibilité du code.
  - Ce mot-clé existait avant C++11. Il s'utilisait comme `static`, mais pour indiquer qu'une variable est de classe d'allocation automatique. Vu que c'est le cas par défaut pour les variables locales, personne ne l'utilisait en pratique. Il est déprécié dans cet usage.

`rbegin, cbegin, crbegin,…`

- Il y a en fait quatre manières d'itérer sur les éléments d'un vector avec des itérateurs :
  - de `begin()` à `end()`, comme nous venons de le voir
  - de `rbegin()` à `rend()`, qui parcourt le `vector` du dernier au premier élément
  - de `cbegin()` à `cend()`, qui parcourt le `vector` comme `begin()`, mais avec un itérateur constant, i.e. où l'opérateur `*` fournit une rvalue et pas une lvalue.
  - de `crbegin()` à `crend()`, comme `cbegin()` mais du dernier au premier élément
- On peut aussi utiliser `begin()` et `rbegin()` sur un `const vector`. L'itérateur fourni est alors lui aussi constant.

Exemple

```
vector<int> v{2, 3, 5, 7, 11};

for (auto i = v.begin(); i != v.end(); ++i)
   cout << *i << " ";
cout << endl;
```

```
2 3 5 7 11
```

```
for (auto i = v.rbegin(); i != v.rend(); ++i)
   cout << *i << " ";
cout << endl;
```

```
11 7 5 3 2
```

```
for (auto i = v.cbegin(); i != v.cend() - 1; ++i)
   cout << *i << " ";
cout << endl;
```

```
2 3 5 7
```

```
for (auto i = v.crbegin() + 2; i != v.crend(); ++i)
   cout << *i << " ";
cout << endl;
```

```
5 3 2
```

resize(…)

- La méthode `resize` modifie explicitement la taille

```
void resize(size_type taille, value_type valeur = value_type());
```

`taille`	Le nouveau nombre d'éléments
`valeur`	La valeur des éventuels éléments à ajouter si taille est plus grand que la taille actuelle. La valeur par défaut (en général 0) du type des éléments est utilisée par défaut.

- Exemples

```
vector<int> v{2, 3, 5, 7, 11};
v.resize(3);    // {2, 3, 5}
v.resize(6, 1); // {2, 3, 5, 1, 1, 1}
v.resize(8);    // {2, 3, 5, 1, 1, 1, 0, 0}
```

clear()

- Pour le cas particulier où l'on redimensionne à une taille `0`, on peut aussi utiliser la méthode `clear`

```
void clear();
```

- Comme pour toutes les opérations qui diminuent le nombre d'éléments, la méthode a deux actions
  - Elle modifie la taille
  - Elle détruit tous les éléments précédemment stockés au-delà de la nouvelle taille.
    - Cela n'a aucun effet pour les types simples
    - Mais c'est important pour les types composés.
      Typiquement cela libère alors la mémoire qu'ils ont allouée dynamiquement.

empty()

- La méthode `empty` permet de vérifier si un `vector` est vide

```
bool empty();
```

- Les deux expressions suivantes sont équivalentes pour un vector v

```
v.empty()
```

```
(v.size() == 0)
```

push_back(…), pop_back()

- La taille peut aussi être modifiée implicitement via des méthodes qui ajoutent ou retirent des éléments.
- Le plus simple consiste à ajouter/retirer un élément à l'arrière (back) du vector avec les méthodes

```
void push_back(const value_type& val);
void pop_back();
```

  - `push_back` incrémente la taille de 1 et copie la valeur `val` en dernière position (`size() - 1`)
  - `pop_back` décrémente la taille de 1 et détruit l'élément en dernière position

push_back(…), pop_back() (exemple)

```
vector<int> v{2, 3, 5, 7, 11};

v.push_back(13); // {2, 3, 5, 7, 11, 13}

v.resize(3);     // {2, 3, 5}

v.pop_back();    // {2, 3}

v.resize(6,1);   // {2, 3, 1, 1, 1, 1}

v.clear();       // {}

v.push_back(42); // {42}

v.resize(5);     // {42, 0, 0, 0, 0}
```

erase(…)

- La méthode `erase` permet de supprimer des éléments à des positions quelconques.

```
iterator erase(iterator position);
iterator erase(iterator first, iterator last);
```

- Elle supprime soit un seul élément, soit une séquence d'éléments allant de `first` compris à `last` non compris.
- Les paramètres sont des itérateurs. Typiquement `begin()` ou `end()` auxquels on ajoute ou retire éventuellement un décalage entier.

```
vector<int> v{2, 3, 5, 7, 11, 13, 17};
v.erase(v.begin());                       // {3, 5, 7, 11, 13, 17}
v.erase(v.begin() + 1, v.begin() + 3);    // {3, 11, 13, 17}
v.erase(v.begin() + 1, v.end() - 1);      // {3, 17}
```

insert(…)

- La méthode insert permet d'insérer des éléments avant une position spécifiée par un itérateur.

```
iterator insert(iterator position,…);
```

- Les éléments à insérer peuvent être un élément unique ou répété n fois, une séquence, une liste d'initialisation (C++11), …

```
vector<int> v{2, 3};
v.insert(v.begin(), 1);          // {1, 2, 3}
v.insert(v.begin(), 3, 0);       // {0, 0, 0, 1, 2, 3}
v.insert(v.begin() + 5, -1);     // {0, 0, 0, 1, 2, -1, 3}
v.insert(v.begin(), {1, 2, 3}); // {1, 2, 3, 0, 0, 0, 1, 2, -1, 3}

vector<int> w{5, 7};
v.insert(v.begin(), w.begin(), w.end()); // {5, 7, 1, 2, 3, 0, 0, 0, 1, 2, -1, 3}
```

Complexité de ces opérations

- Les algorithmes utilisés par la classe `vector` pour insérer ou supprimer des éléments à l'arrière ou à une position quelconque sont essentiellement ceux vus pour les tableaux classiques.
- Insérer/supprimer à l'arrière est une opération peu coûteuse.
- Insérer/supprimer à une autre position utilise une boucle qui déplace tous les éléments qui suivent cette position.

Gestion de la capacité

- `vector` distingue les notions de
  - taille – le nombre d'éléments stockés
  - capacité – le nombre d'élément stockables dans la mémoire actuellement réservée
- Quand une méthode doit augmenter la taille au-delà de la capacité, une capacité supérieure est réallouée et les éléments sont copiés dans ce nouvel emplacement.
- Augmenter la capacité est une opération coûteuse mais rare, ce qui la rend efficace en moyenne.
- Quand une méthode doit diminuer la taille, elle ne modifie pas la capacité. Cela peut être coûteux en mémoire utilisée.

reserve(), capacity(), shrink_to_fit()

- Si nécessaire, il est possible d'optimiser explicitement la gestion de cette capacité grâce aux méthodes

```
void reserve(size_type n);
size_type capacity();
void shrink_to_fit();
```

- `reserve` spécifie la capacité à réserver en mémoire. N'a aucun effet si la capacité est déjà supérieure ou égale à la quantité demandée.
- `capacity` renvoie la capacité actuelle.
- `shrink_to_fit` (C++11) diminue la capacité pour qu'elle soit égale à la taille (ou pas, cela dépend du compilateur utilisé)

vector à 2 dimensions

- Un `vector` peut lui-même se composer d'éléments de type `vector`

```
// un vecteur de 10 entiers
vector<int> v1(10);
```

```
// un vecteur de vecteurs d'entiers
vector<vector<int>> v2;
```

- Ainsi déclaré, `v2` ne contient aucun élément.
  Si l'on veut le redimensionner, il faut le faire en deux temps :

```
v2.resize(3);        // 1 vecteur de 3 vecteurs vides
for (size_t i = 0; i < v2.size(); ++i)
   v2[i].resize(4); // redimensionner chaque vecteur
```

vector à 2 dimensions

- Notons que rien n'oblige à utiliser la même taille pour chaque vecteur imbriqué.
- L'accès aux données se fait comme suit :

```
cout << v2[2][3];
cout << v2.at(2).at(3);
```

- On peut aussi définir les dimensions dans la déclaration en utilisant bien les constructeurs

```
// un vector de 3 vectors de 4 int initialisés à 0
vector<vector<int>> v(3, vector<int>(4));
```

typedef

- Un type tel que `vector<vector<int>>` devient difficilement lisible, surtout si l'on passe à 3, 4, … dimensions.
- On peut retrouver de la lisibilité en utilisant des types synonymes avec l'instruction `typedef`

```
typedef typeExistant nouveauType;
```

- Pour définir une matrice de 3 lignes de 4 entiers, on peut écrire :

```
typedef vector<int>   Ligne;
typedef vector<Ligne> Matrice;
Matrice m(3, Ligne(4));
```

Types synonymes, `using`

- Les types synonymes peuvent être utilisés dans d'autres contextes pour améliorer la lisibilité du code
- Attention, ce sont de purs synonymes. Il n'y a pas de vérification de type. Le code suivant est correct :

```
typedef int Pomme;
typedef int Orange;
Pomme  p = 1;
Orange q = p;
```

En C++, on utilisera using de préférence à typedef

- C++11 a introduit une syntaxe alternative : `using nouveauType = typeExistant;`
- La définition d'une matrice devient :

```
using Ligne   = vector<int>;
using Matrice = vector<Ligne>;
```

5. La classe `array`

array

- C++11 a introduit une classe `array` pour des tableaux non redimensionnables, de taille connue à la compilation qui
  - ne demande pas plus de mémoire que pour les tableaux « classiques » en C
  - fournit une interface similaire à la classe `vector`
- Seule la déclaration est différente de `vector`

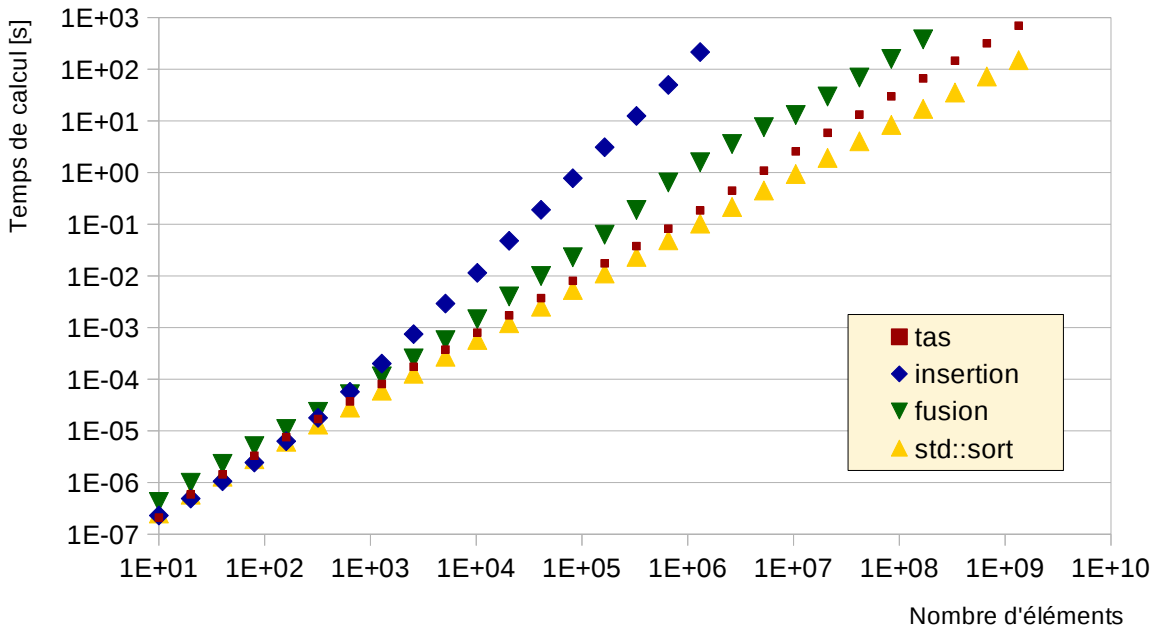
```
#include <array>
…
array<int, 10>   tabInt;   // tab de 10 int
array<char, 5>   tabChar;  // tab de  5 char
array<float, 20> tabFloat; // tab de 20 float

// tableau de 10 lignes de 80 char
array<array<char, 80>, 10> lignes;
```

array

- Attention, avec `array`, **la taille doit toujours être spécifiée dans le nom du type**, y compris pour l'initialisation avec agrégat et pour le passage en paramètre, même pour un `array` à une seule dimension.
  Cela rend ce type mal pratique sans la notion de généricité (*voir chapitre 8*).
- L'accès aux éléments est identique aux objets de type `vector` :
  on dispose de `operator[]`, `at()`, `back()`, `front()`, `begin()`, `end()`, `cbegin()`, `cend()`, `rbegin()`, `rend()`, `crbegin()`, `crend()`
- On dispose d'une méthode supplémentaire `fill()` qui remplit un objet `array` d'éléments identiques.
- Il n'y a par contre (évidemment) pas de méthodes liées au redimensionnement.
  Les seules méthodes liées à la taille sont `size()` et `empty()`.

6. Algorithmes de tri simples



Introduction

Pourquoi étudier les tris ?

- Pour les humains, les données triées sont plus faciles à appréhender
  - dictionnaire ☾ trié alphabétiquement
  - agenda ☾ trié par date
  - annuaire téléphonique ☾ trié par localité, puis alphabétiquement
- Pour les ordinateurs, le tri permet un accès plus efficace aux données
  - par exemple via une recherche « dichotomique »

Introduction

- Le choix et la mise en œuvre du bon algorithme de tri, dans un contexte donné, est essentiel.
- **Exemple**
  Tri de N éléments (i7-6700K @ 4GHz, Linux)
  (temps comprenant la génération de valeurs aléatoires)

N	Tri par insertion	Tri par fusion de listes	Tri std::sort	Tri par tas
10	24ns	41ns	25ns	22ns
10^3	133µs	82µs	47µs	64µs
10^6	123 secondes	1,09 seconde	0,078 seconde	0,139 seconde
10^9	env. 4 ans	env. 17 minutes	112 secondes	503 secondes

Introduction

- Il existe de nombreux algorithmes de tri :
  tri à bulles, tri par insertion, tri par sélection, tri de Shell, tri par fusion,
  tri rapide…
  Pour plus d'informations, voir par exemple https://fr.wikipedia.org/wiki/Algorithme_de_tri
- L'étude détaillée des tris se fera dans le cours d'algorithmes et structures de données (ASD)
- … mais vu leur utilisation fréquente, nous allons, ici déjà, présenter les plus simples d'entres eux, à savoir :
  - le **tri à bulles** (*bubble sort*)
  - le **tri par insertion** *(insertion sort)*
  - le **tri par sélection** *(selection sort)*

Introduction : complexité des tris

- Les algorithmes de tri peuvent être classés en trois catégories selon leur complexité moyenne en temps
  - La complexité en temps est une mesure du temps utilisé par un algorithme, exprimé comme fonction de la taille de l'entrée. Le temps compte le nombre d'étapes de calcul avant d'arriver à un résultat. (Wikipédia)

1. Les tris en n^2 (**complexité quadratique** : notée **O(n^2)**)
   - Cas des trois algorithmes étudiés ici

2. Les tris en $n \cdot \log(n)$ (**complexité linéarithmique** : notée **O($n \cdot \log(n)$)**)
   - Sont en général plus rapides que les tris en n^2, disponible de manière standard dans tous les langages évolués, très simple à utiliser en C++, **à privilégier**

3. Les tris en n (**complexité linéaire** : notée **O(n)**)
   - Sont les plus rapides, mais ne s'appliquent qu'à des données très particulières

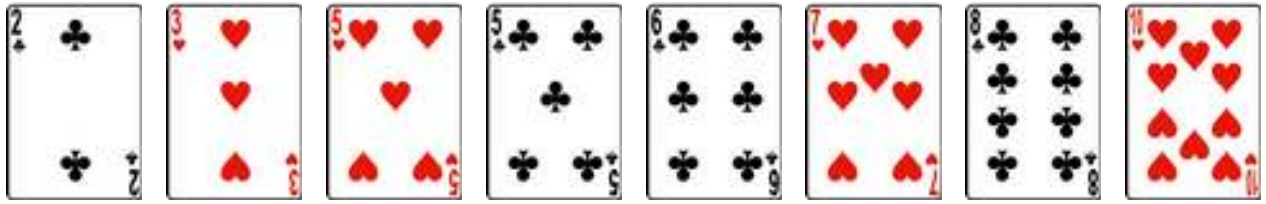
- Ci-dessus, n désigne le nombre d'éléments à trier. Un tri en n^2, par ex., signifie que le temps nécessaire au tri est, en moyenne, proportionnel au carré de n. O(xxx) est la notation de Landau.

Autres propriétés des algorithmes de tri

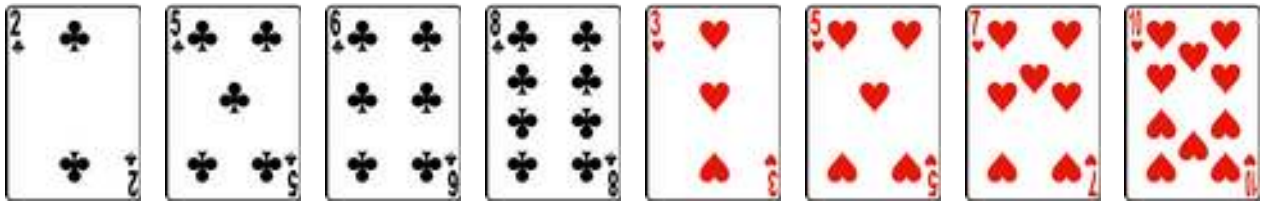
- La complexité n'est pas le seul critère d'appréciation d'un algorithme. D'autres facteurs importants sont à prendre en compte :
  - La quantité et la distribution des données à trier
    - Suivant les cas (données partiellement / déjà triées, ou totalement aléatoires), on doit considérer parfois la complexité dite « dans le meilleur des cas » ou, au contraire, la complexité dite « dans le pire des cas »
  - La stabilité ou non de l'algorithme *(voir slide suivant)*
  - Le recours à un tableau auxiliaire ou a de la récursion dans l'algorithme de tri (place mémoire)
  - Etc.

Stabilité d'un algorithme de tri : exemple

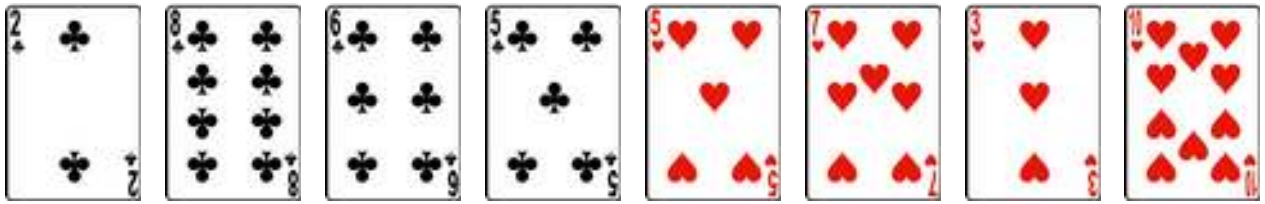
- Entrée : cartes triées par valeur croissante



- Sortie : cartes triées par couleur, avec la convention suivante : ♣ < ♦ < ♥ < ♠
  - **Tri stable** = l'ordre des valeurs est conservé au sein de chaque couleur



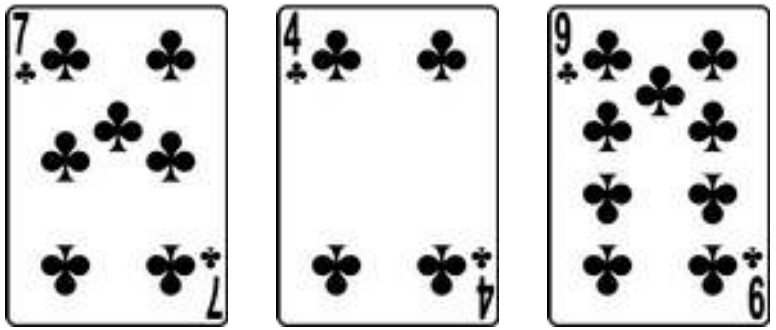
  - **Tri non stable** = l'ordre des valeurs *n'est pas* conservé au sein des couleurs



Présentation des algorithmes de tri

Pour chaque algorithme, nous présenterons :

- son principe de fonctionnement et une animation



- le pseudo-code de celui-ci
- un exemple simple d'implémentation en C++

Conventions utilisées pour le pseudo-code

- Les variables sont colorées **en vert**.
- Les tableaux sont représentés par des lettres majuscules.
- Les nombres sont représentés par des lettres minuscules
- Les tableaux de **n** éléments sont indexés de 1 à **n**. On note **A(i)** le i^{ième} élément du tableau **A**.

```
fonction SelectionSort(A, n)

   pour i de 1 à n-1 boucler
      imin ← i
      pour j de i+1 à n boucler
         si A(j) < A(imin), alors
            imin ← j
         fin si
      fin pour j
      permuter A(i) et A(imin)
   fin pour i
```

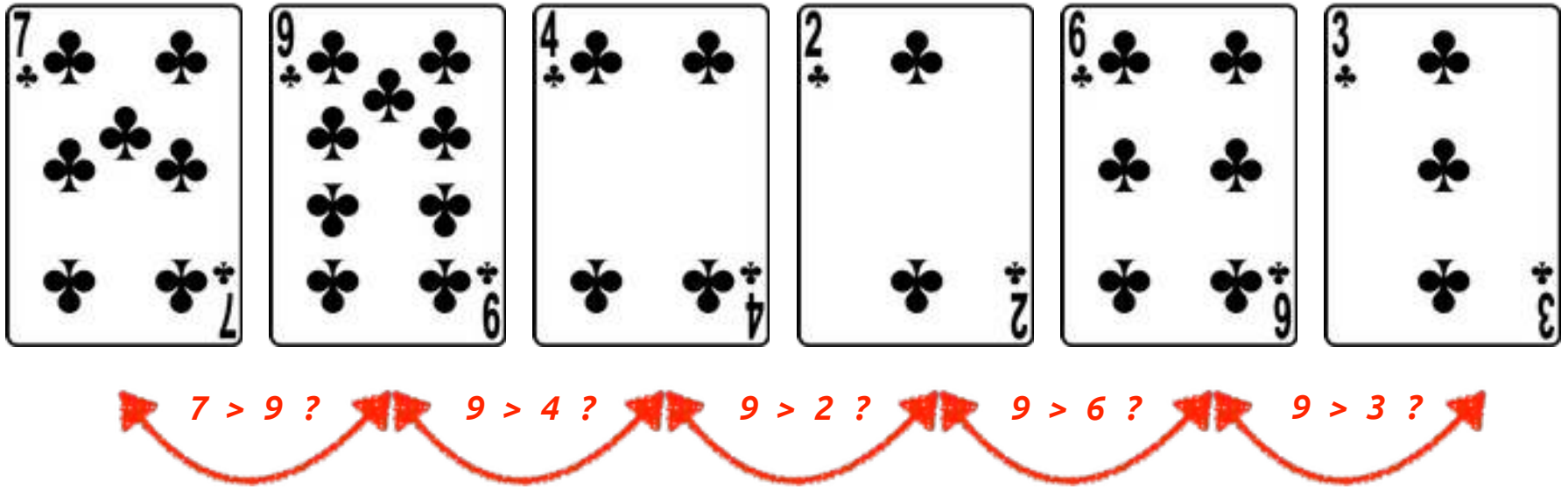
Tri à bulles : principe

Source : *Algorithmes et structures de données avec Ada, C++ et Java*, A. Guerid, P. Breguet, H. Röthlisberger, PPUR

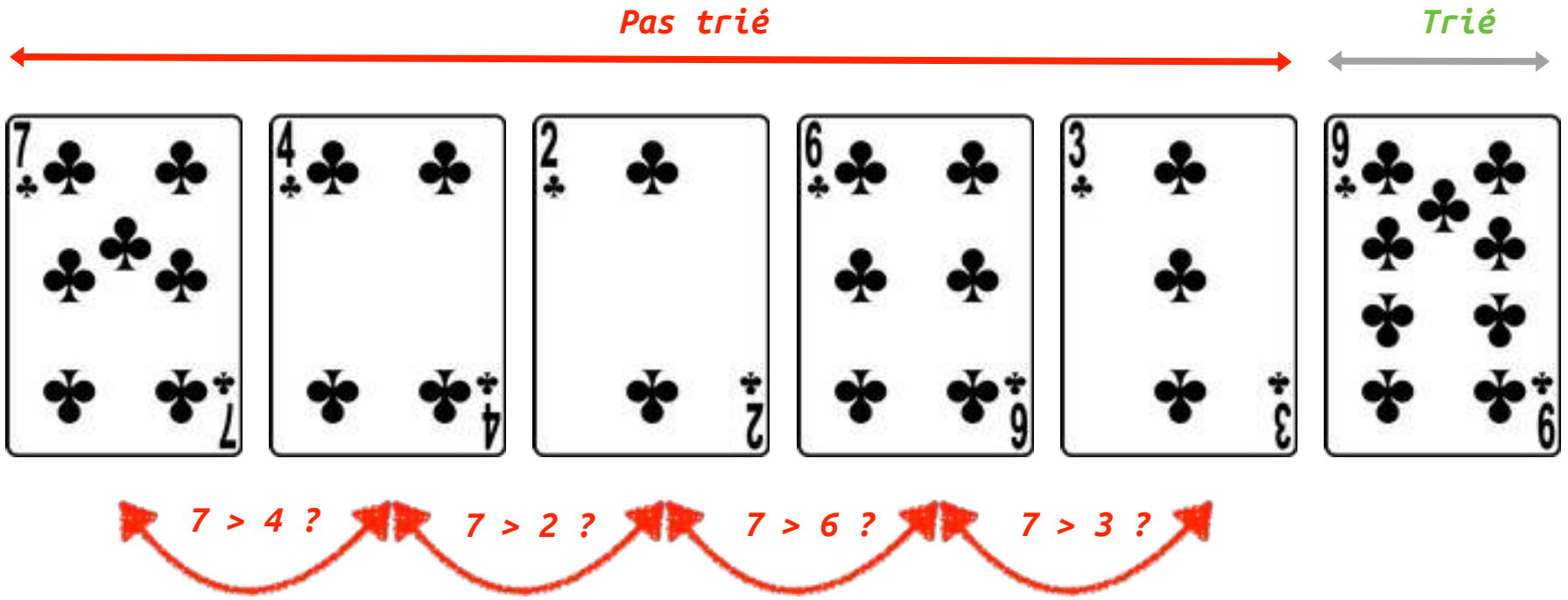
- **Principe de fonctionnement**
  - Comparer deux éléments consécutifs et les permuter s'ils ne sont pas bien ordonnés.
  - Après avoir traité les éléments en 1ère et en 2e position, on recommence avec ceux en 2e et 3e position, etc., jusqu'à l'échange (éventuel) de l'avant-dernier et du dernier. Ce parcours a pour effet de placer l'élément le plus grand en dernier (fin du tableau).
  - Un second parcours permet de placer correctement l'avant-dernier élément.
  - Après *i* parcours, les *i* derniers (plus grands) éléments sont bien triés.
  - Il suffit donc d'effectuer *n* – 1 parcours, où *n* est la taille du tableau.

❖ Le nom *tri à bulles* vient du fait que le décalage de l'élément le plus grand de gauche à droite lors d'un parcours s'apparente à la remontée d'une bulle d'air dans un liquide.

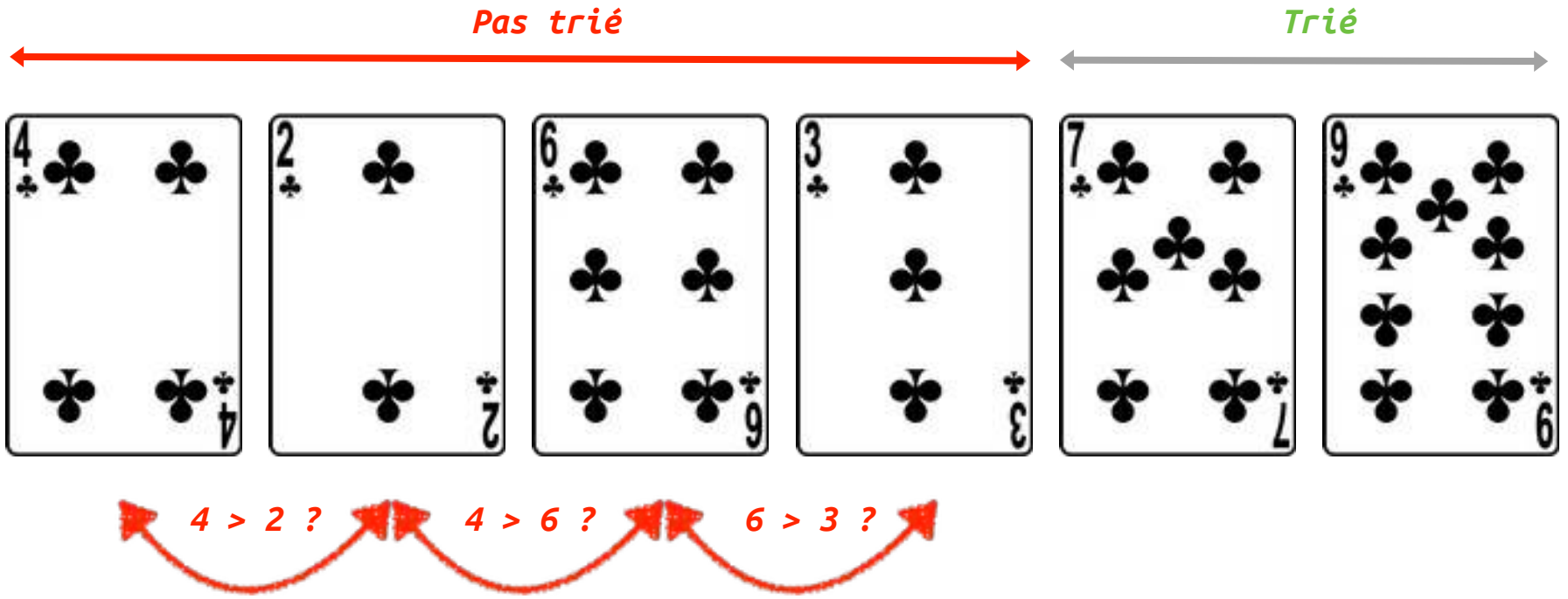
Tri à bulles : animation



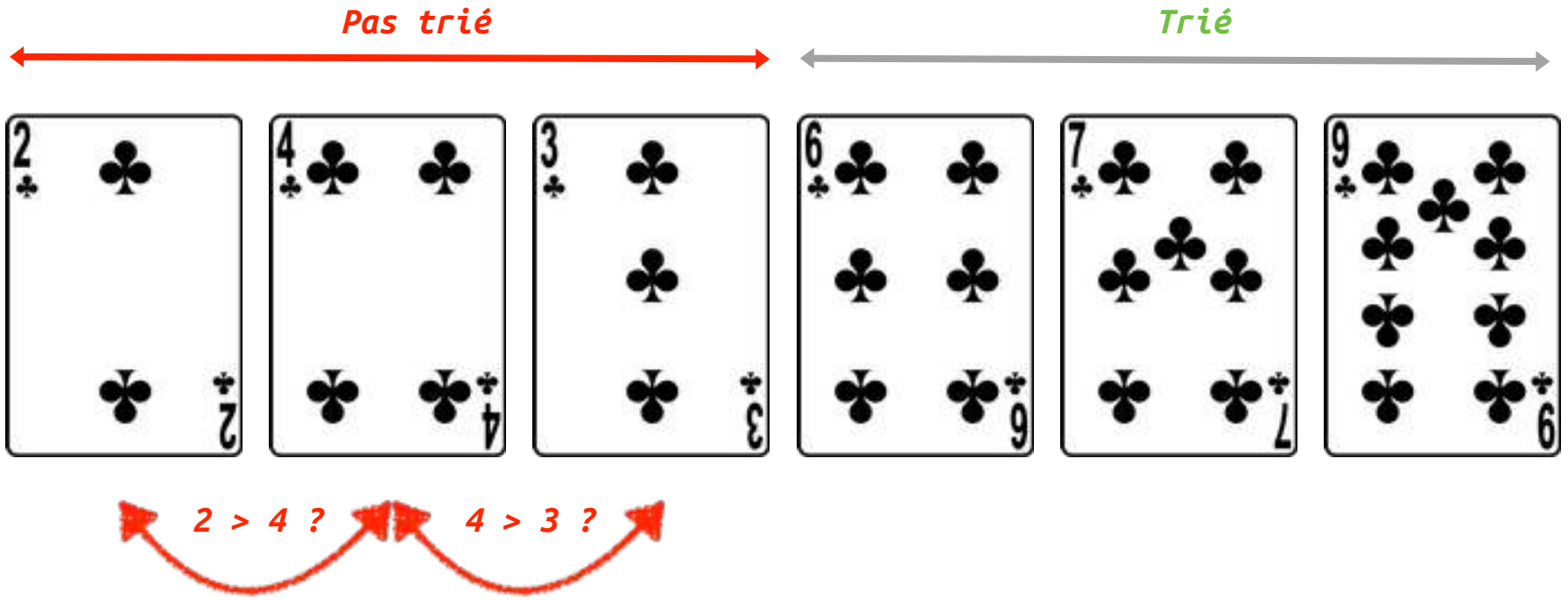
Tri à bulles : animation



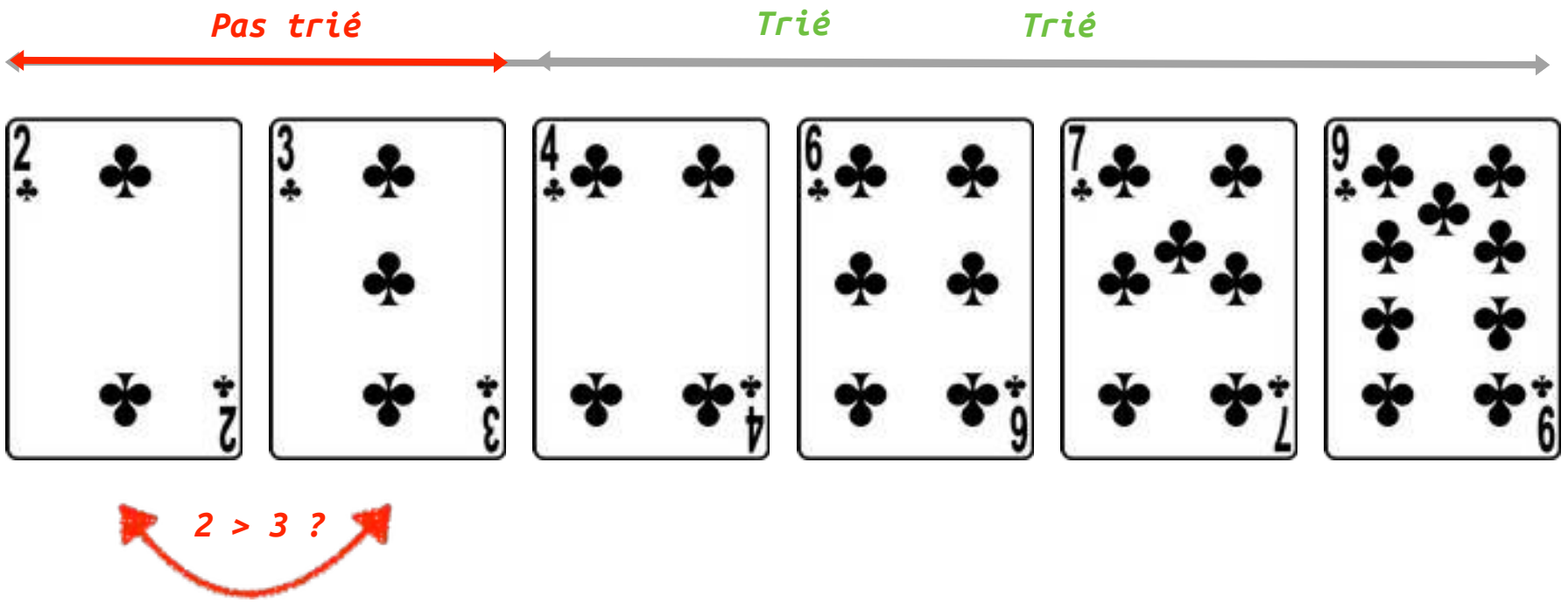
Tri à bulles : animation



Tri à bulles : animation



Tri à bulles : animation



Tri à bulles : pseudo-code et exemple C++

```
fonction bubbleSort(A, n)

   pour i de 1 à n-1 boucler
      pour j de 1 à n-i boucler
         si A(j) > A(j+1), alors
            permuter A(j) et A(j+1)
         fin si
      fin pour j
   fin pour i
```

```
void bubbleSort(vector<int>& v) {
   if (v.size() > 0) {
      for (size_t i = 0; i < v.size() - 1; ++i) {
         for (size_t j = 1; j < v.size() - i; ++j) {
            if (v[j - 1] > v[j])
               swap(v[j - 1], v[j]); // permutation
         }
      }
   }
}

int main() {
   vector<int> v{9, 5, 2, 6, 7, 3, 4, 1, 8};
   bubbleSort(v);
   for (int n : v)
      cout << n << ' '; // 1 2 3 4 5 6 7 8 9
}
```

Tri à bulles : algorithme revisité

- L'implémentation de l'algorithme du tri à bulles précédente est correcte… mais pas optimale.
- Il est, en effet, inutile de poursuivre l'algorithme si, pour un *i* donné, aucune des itérations sur *j* ne donne lieu à une permutation (cela signifie que le tableau est complètement trié)
- Le code ci-contre propose une version revisitée de l'algorithme tenant compte de cette observation.

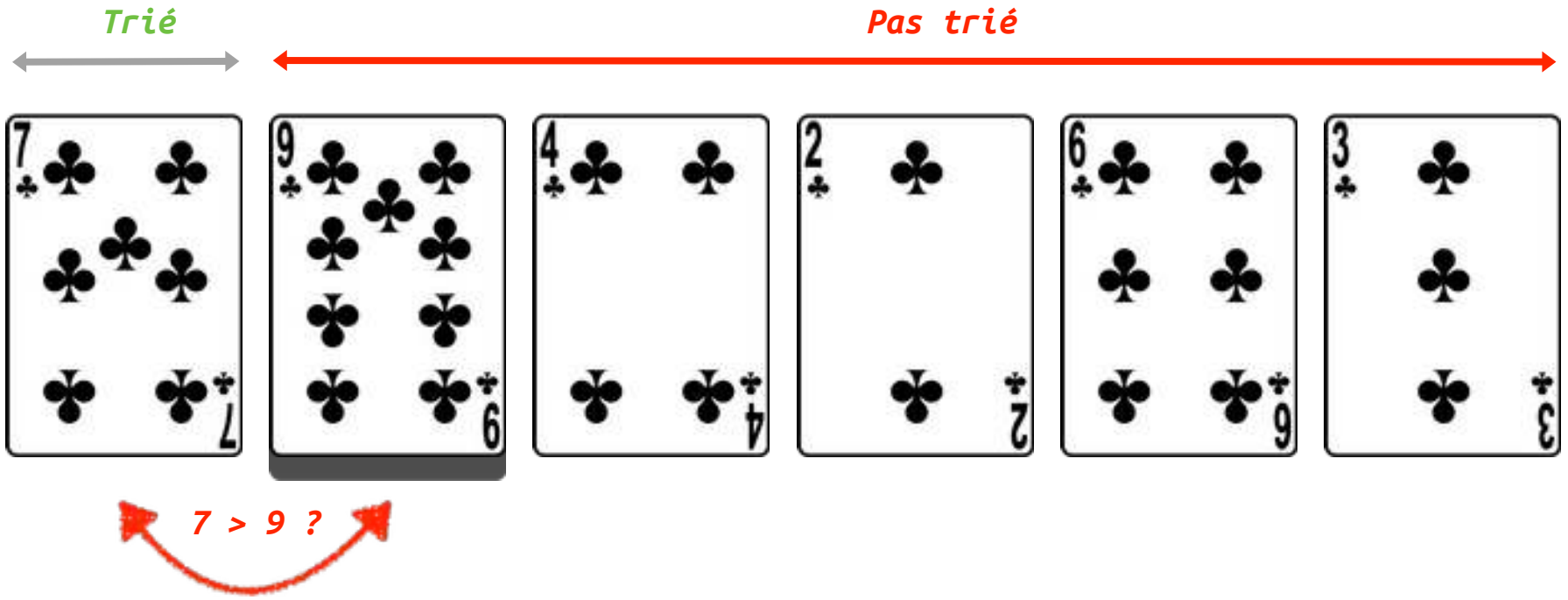
```
void bubbleSort(vector<int>& v) {
   if (v.size() > 0) {
      bool fini = false;
      size_t taille = v.size();
      while (!fini) {
         fini = true;
         for (size_t i = 0; i < taille - 1; ++i) {
            if (v[i] > v[i + 1]) {
               swap(v[i], v[i + 1]);
               fini = false;
            }
         }
         --taille;
      }
   }
}
```

Tri par insertion : principe

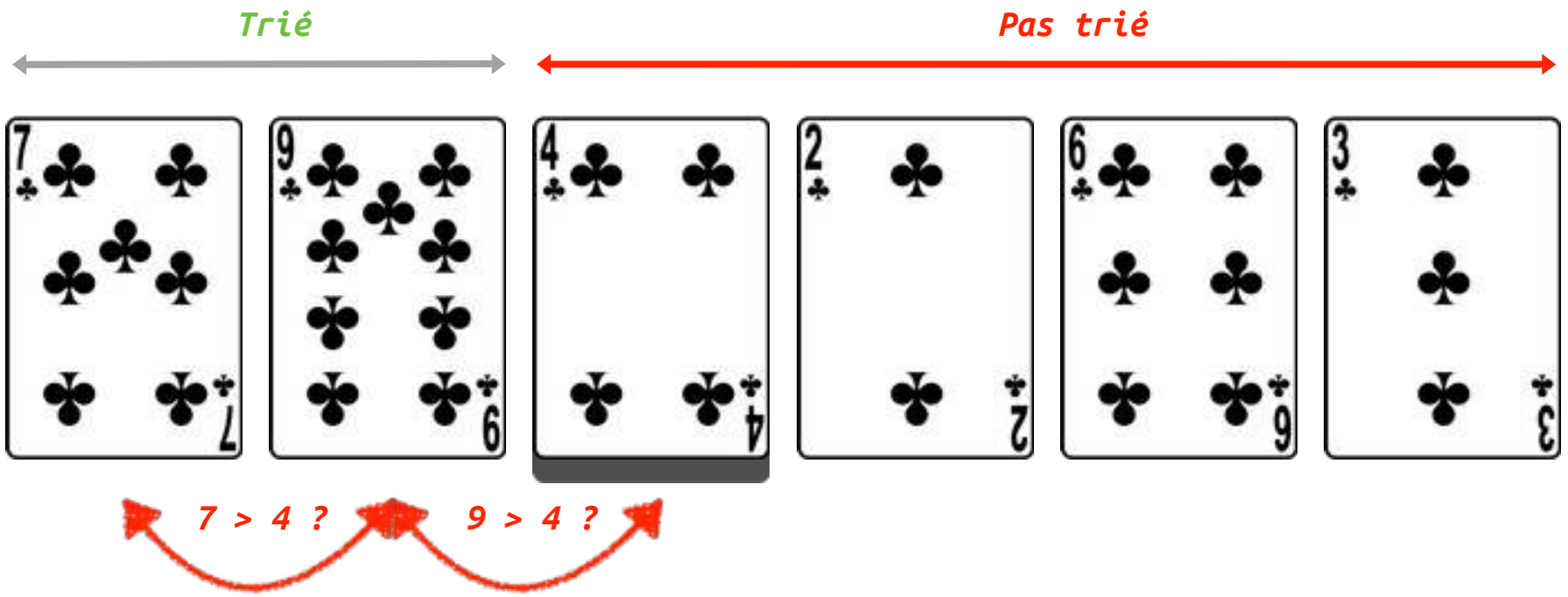
Source : *Algorithmes et structures de données avec Ada, C++ et Java*, A. Guerid, P. Breguet, H. Röthlisberger, PPUR

- Le tri par insertion est une amélioration du tri à bulles, l'objectif étant de remplacer les permutations par des affectations, ce qui est plus efficace.
- **Principe de fonctionnement**
  - Au $i^{ième}$ parcours, on considère que les i premiers éléments sont déjà triés; il suffit alors d'insérer le $i+1^{ième}$ élément au bon endroit parmi les i premiers.
  - Pour cela, on met d'abord la valeur de l'élément $i+1$ dans une variable tampon.
  - Ensuite, par décalages successifs, un emplacement libre est créé parmi les éléments triés, emplacement dans lequel cette valeur sera insérée.

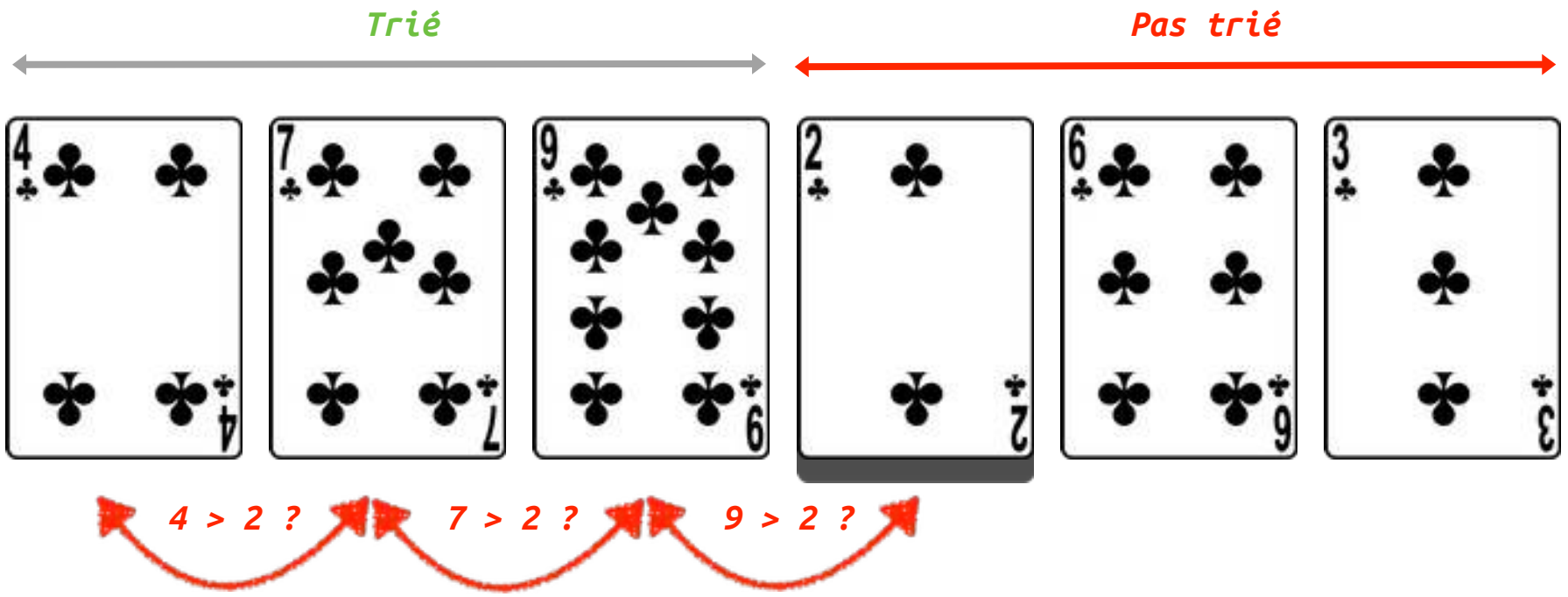
Tri par insertion : animation



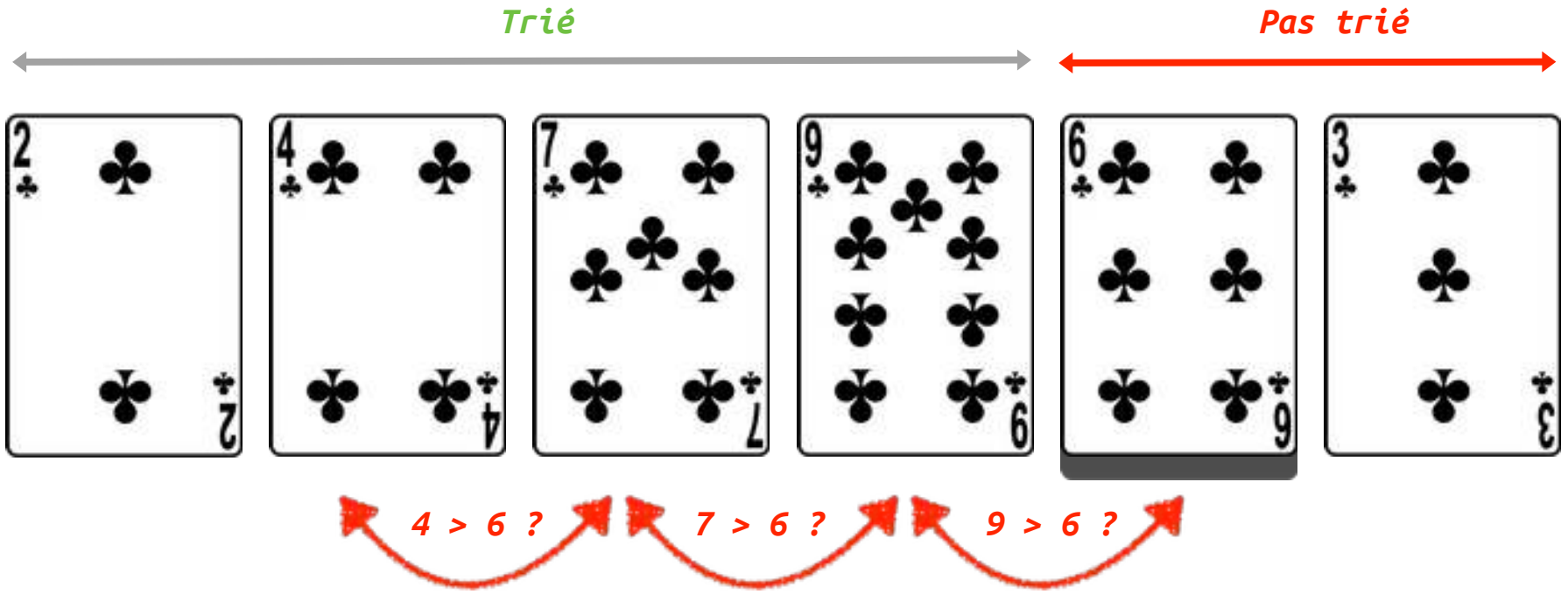
Tri par insertion : animation



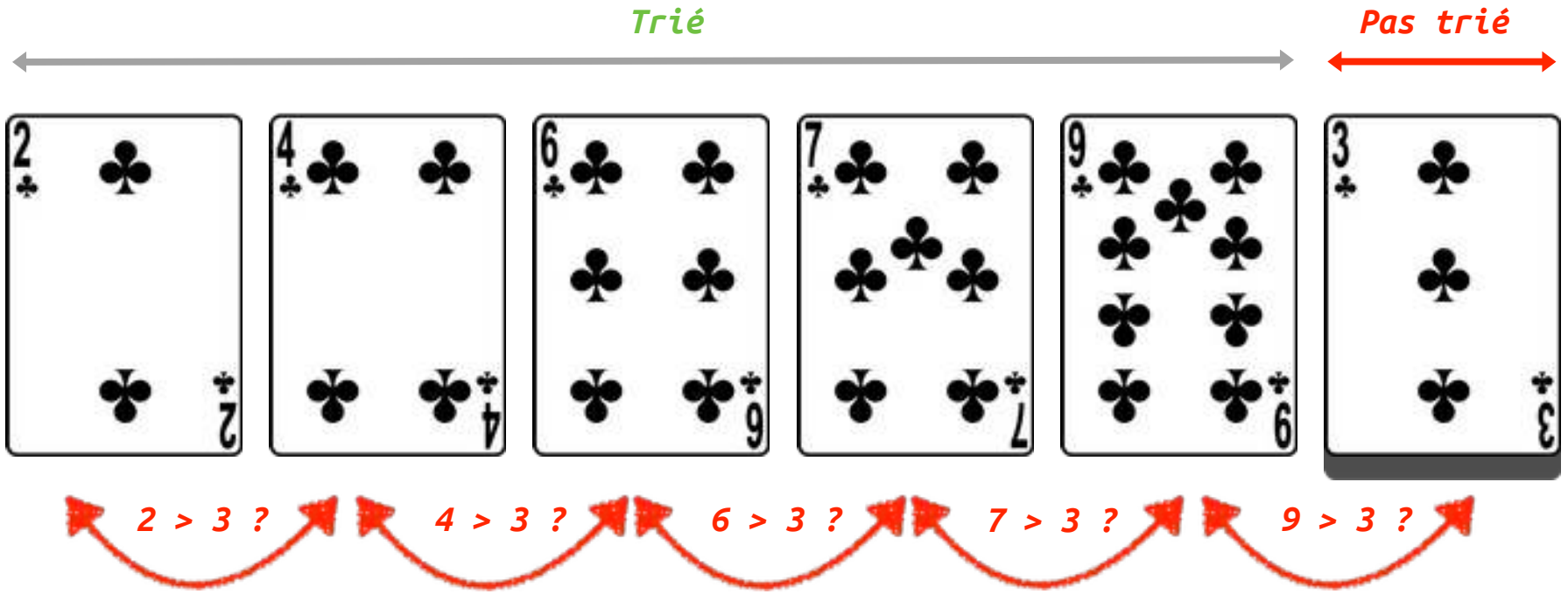
Tri par insertion : animation



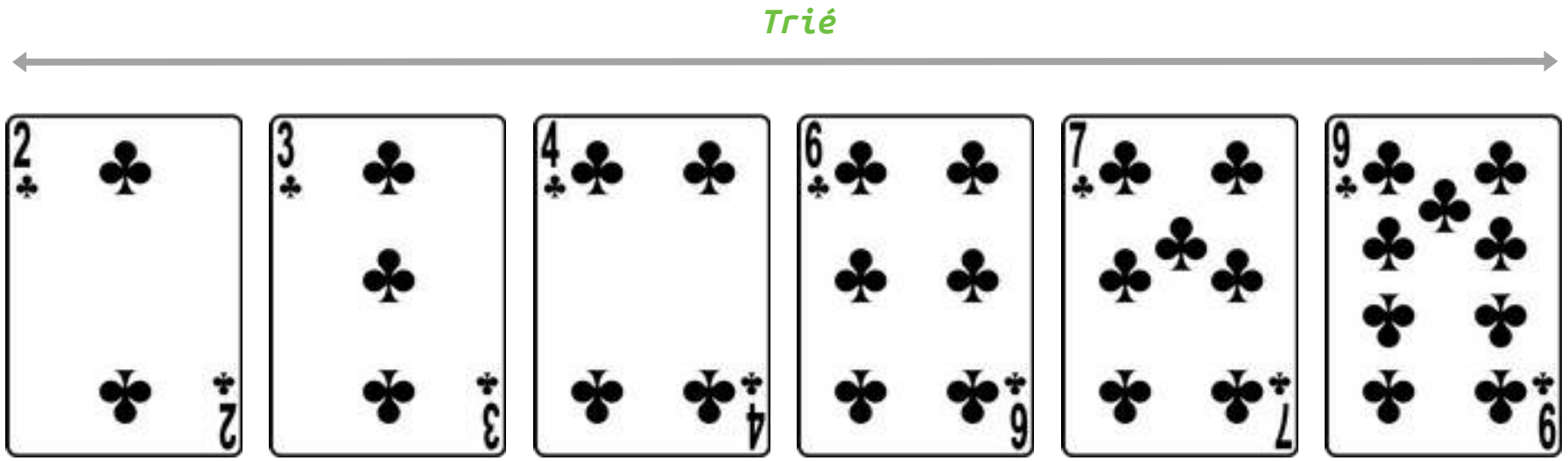
Tri par insertion : animation



Tri par insertion : animation



Tri par insertion : animation



Tri par insertion : pseudo-code et exemple

```
fonction insertSort(A, n)

   pour i de 2 à n boucler
      tmp ← A(i)
      j ← i
      tant que j-1 ≥ 1 et A(j-1) > tmp boucler
         A(j) ← A(j-1)
         décrémenter j de 1
      fin tant que
      A(j) ← tmp
   fin pour i
```

```
void insertSort(vector<int>& v) {
   int tmp;
   size_t j;
   for (size_t i = 1; i < v.size(); ++i) {
      tmp = v[i];
      j = i;
      while (j >= 1 && v[j - 1] > tmp) {
         v[j] = v[j - 1];
         --j;
      }
      v[j] = tmp;
   }
}

int main() {
   vector<int> v{9, 5, 2, 6, 7, 3, 4, 1, 8};
   insertSort(v);
   for (int n : v)
      cout << n << ' '; // 1 2 3 4 5 6 7 8 9
}
```

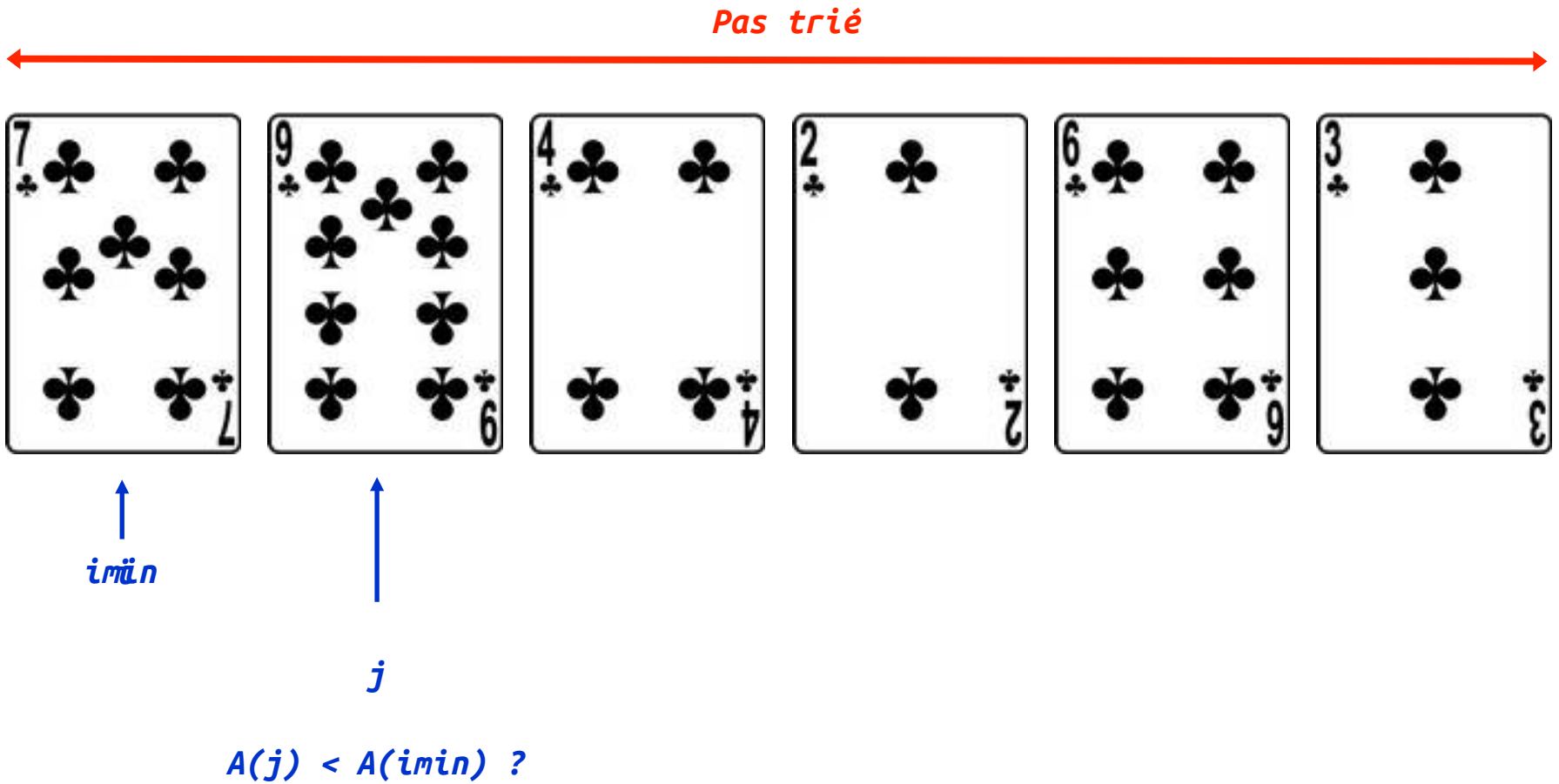
Tri par sélection : principe

Source : *Algorithmes et structures de données avec Ada, C++ et Java*, A. Guerid, P. Breguet, H. Röthlisberger, PPUR

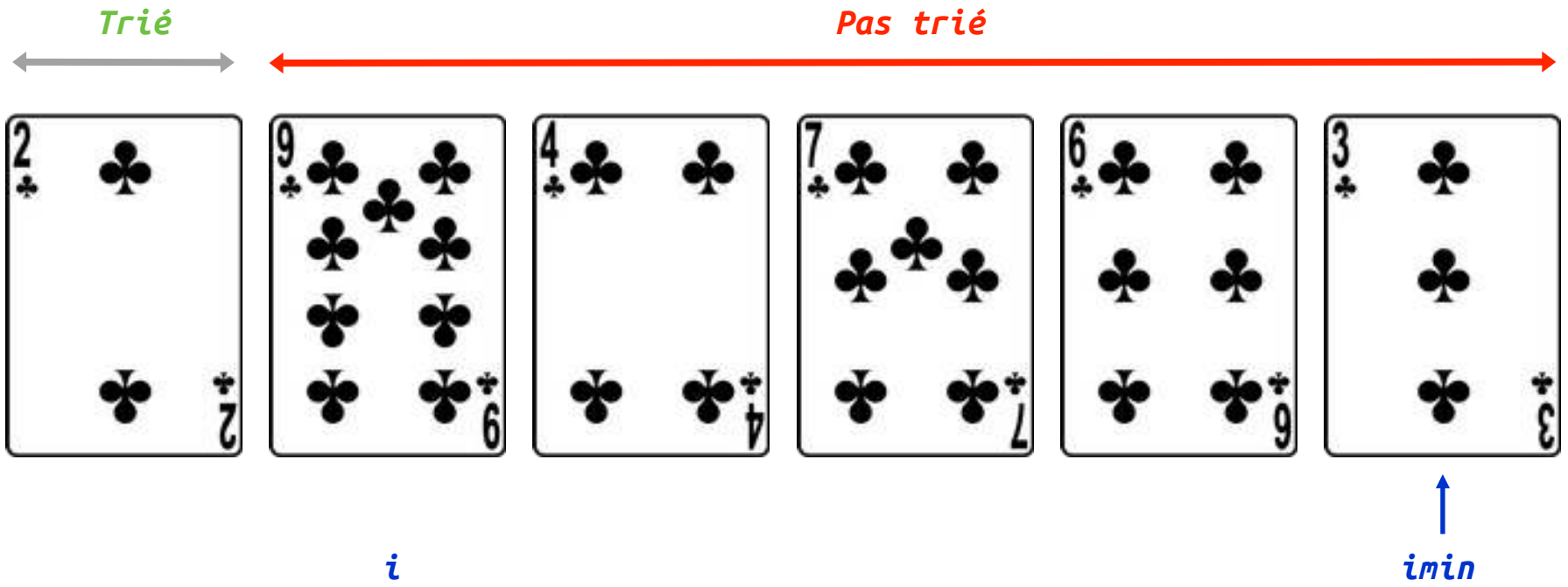
- Le tri par sélection est une autre variante du tri à bulles.
  Elle permet d'économiser encore plus d'affectations en ramenant les permutations au nombre d'éléments à trier.

- **Principe de fonctionnement**
  - Au *i* ième parcours, on considère que les *i* – 1 premiers éléments sont déjà triés.
  - L'idée est de chercher le plus petit des éléments non encore triés : il suffit alors d'une seule permutation pour le placer à la suite des éléments déjà triés qui sont tous plus petits (ou égaux) que lui.

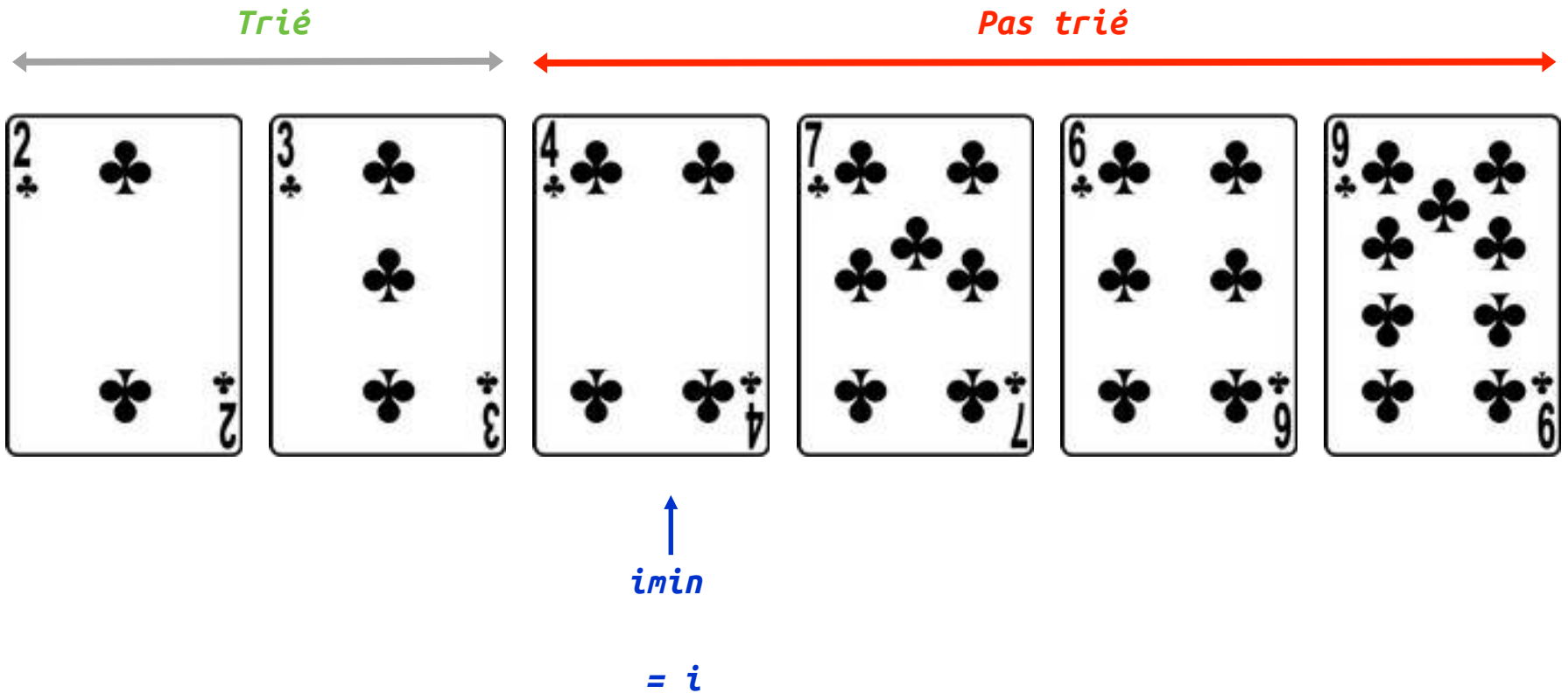
Tri par sélection : animation



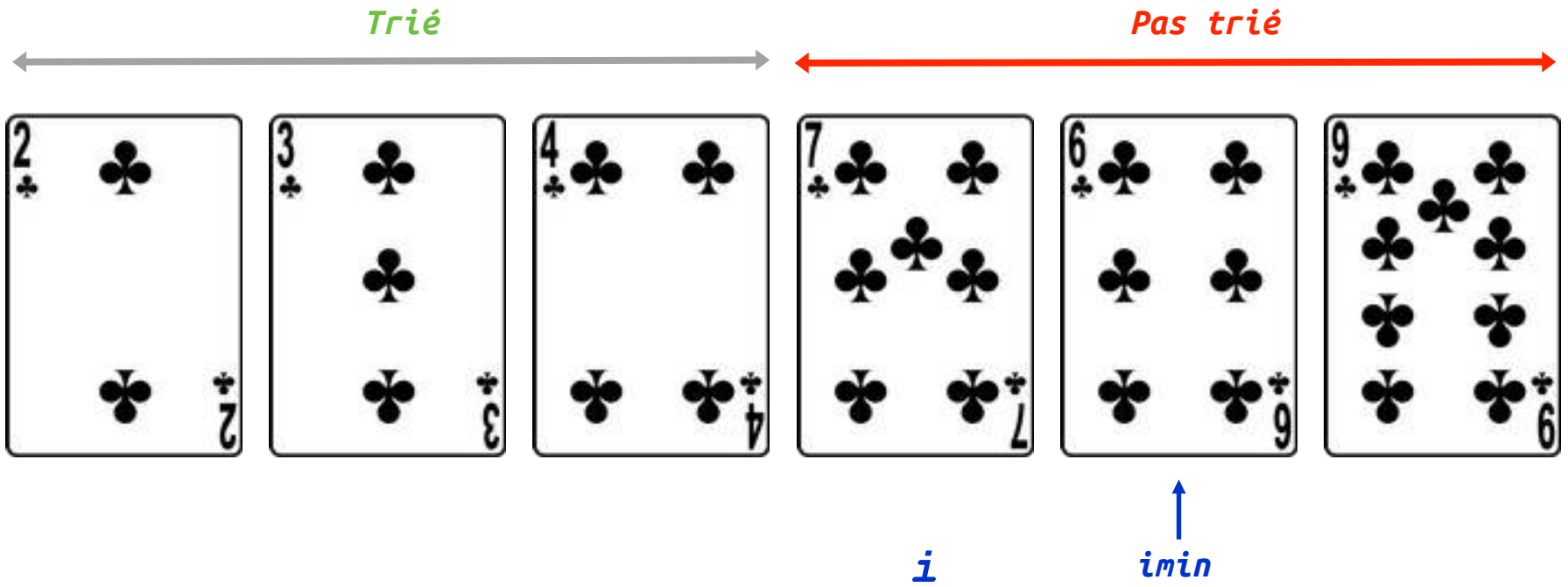
Tri par sélection : animation



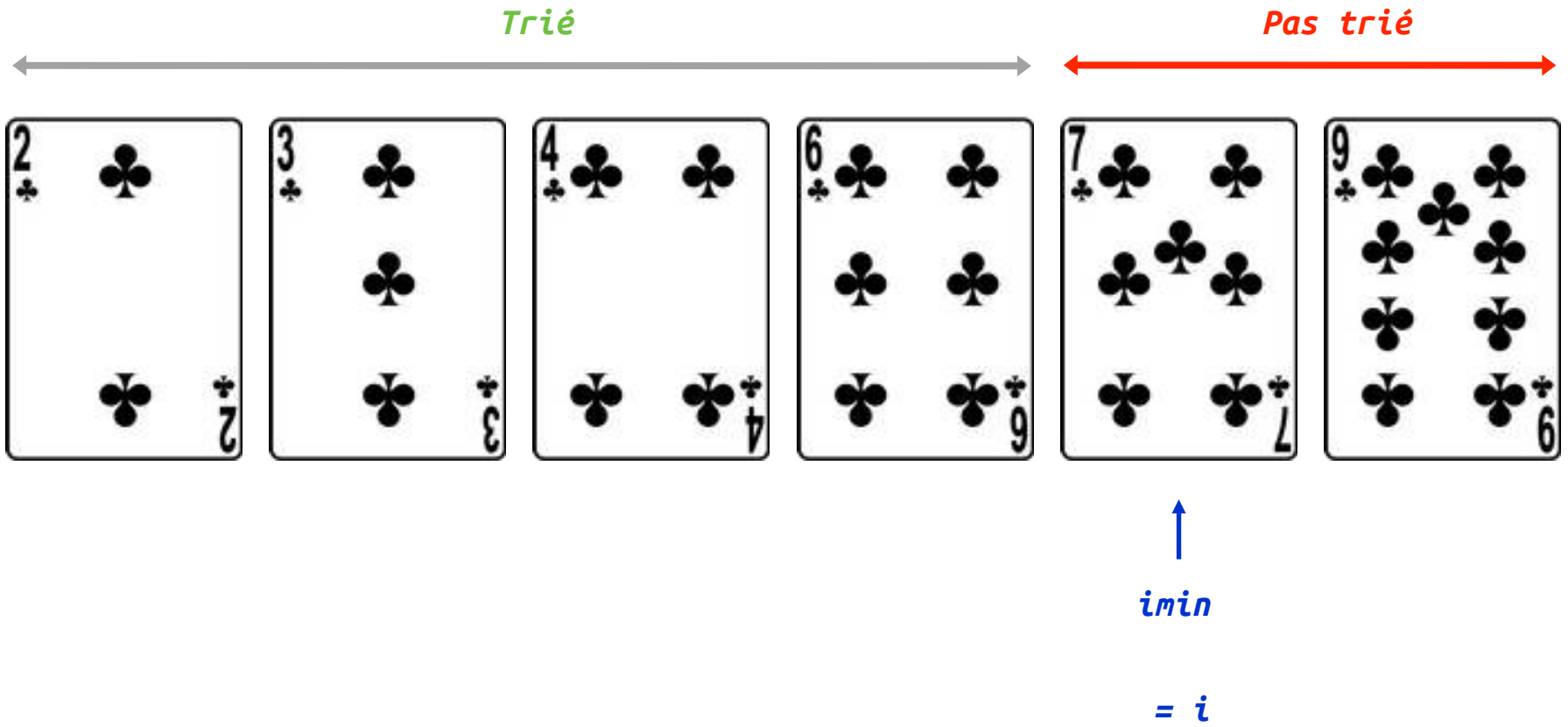
Tri par sélection : animation



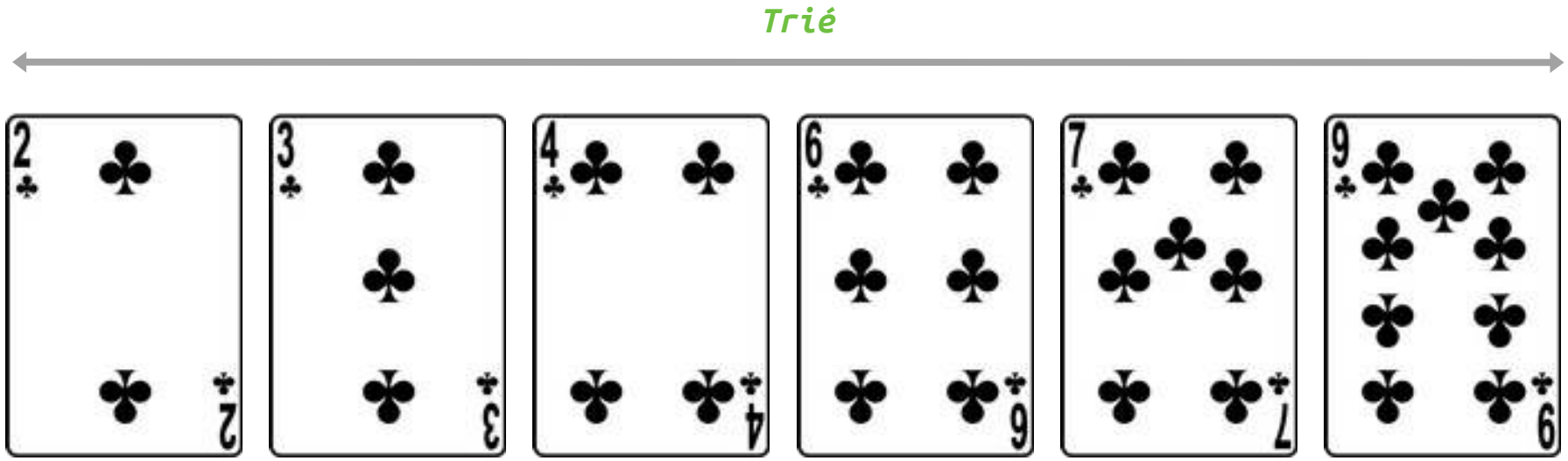
Tri par sélection : animation



Tri par sélection : animation



Tri par sélection : animation



Tri par sélection : pseudo-code et exemple

```
fonction selectionSort(A, n)

   pour i de 1 à n-1 boucler
      imin ← i
      pour j de i+1 à n boucler
         si A(j) < A(imin), alors
            imin ← j
         fin si
      fin pour j
      permuter A(i) et A(imin)
   fin pour i
```

```
void selectionSort(vector<int>& v) {
   if (v.size() > 0) {
      size_t iMin;
      for (size_t i = 0; i < v.size() - 1; ++i) {
         iMin = i;
         for (size_t j = i + 1; j < v.size(); ++j) {
            if (v[j] < v[iMin])
               iMin = j;
         }
         swap(v[i], v[iMin]); // permutation
      }
   }
}

int main() {
   vector<int> v{9, 5, 2, 6, 7, 3, 4, 1, 8};
   selectionSort(v);
   for (int n : v)
      cout << n << ' '; // 1 2 3 4 5 6 7 8 9
}
```

Complexité : résumé

- Le tableau ci-dessous résume les principales caractéristiques de nos trois algorithmes :

	Complexité		
Tri	**Meilleur cas**	**Pire cas**	**Stable**
à bulles	n	n^2	Oui
par insertion	n	n^2	Oui
par sélection	n^2	n^2	Non

  - Aucun des algorithmes ci-dessus ne requiert de tableau auxiliaire.
  - Des mesures comparatives montrent que le tri par insertion est généralement plus rapide que le tri par sélection qui, lui-même, est plus rapide que le tri à bulles.

7. Les librairies
<algorithm> et <numeric>

<algorithm>

- La librairie `<algorithm>` définit des **fonctions** (recherche, tri, comptage, manipulation, …) pour des **opérations sur des « plages » d'éléments**.
- Une plage est définie abstraitement comme `[premier, dernier)`, où `dernier` n'appartient pas à la plage, mais fait référence à l'élément suivant le dernier élément à inspecter ou modifier.
- Typiquement, on traite donc un `vector v` en fournissant en paramètres de début et fin de plage `v.begin()` et `v.end()`, mais on peut aussi utiliser d'autres itérateurs ou modifier ceux-ci en ajoutant des décalages.
- On peut aussi traiter un tableau `int tab[N]`, en fournissant simplement en paramètres `tab` et `tab + N`

sort — Efficace, simple, à privilégier

```
void sort(first, last);
void sort(first, last, compare);
```

- Trie les éléments de la plage `[first,last)` du plus petit au plus grand
  - `first` et `last` définissent la plage
  - `compare` est une fonction binaire renvoyant un booléen qui indique si le premier paramètre est plus petit que le second

```
vector<double> v(n);
double t[10]
// trie v et t par valeurs croissantes
sort(v.begin(), v.end());
sort(t, t+10);
```

- La norme C++11 impose aux implémentations de garantir que la complexité en temps du tri est dans tous les cas O(*N*·log(*N*)) où *N* = *last* – *first*
  - n'impose pas d'algorithme ☾ p.ex., GCC utilise « Introsort » (voir `stl_algo.h`)
  - n'impose pas que le tri soit stable (et Introsort ne l'est pas)
    - `stable_sort` garantit un tri stable de complexité O(*N*·log(*N*)) si la mémoire suffit

Non-modification de la séquence

- Voyons tout d'abord une série d'algorithmes (donc des fonctions dans `<algorithm>`) ne modifiant pas la plage d'éléments reçus en paramètres.
- Ils effectuent tous une boucle de type :

```
for (auto i = first; i != last; ++i)
  ...
```

  et garantissent qu'à aucun moment ils ne font d'affectation sur `*i`.

- Par contre, ils peuvent appeler des fonctions en passant `*i` en paramètre, ce qui peut éventuellement le modifier selon la fonction appelée, comme nous le verrons pour `for_each`.

Notation utilisée

- Dans ce qui suit, nous utilisons une notation très simplifiée pour présenter les prototypes des fonctions de `<algorithm>`. Par exemple, nous écrivons :

```
for_each (first, last, fn);
```

alors que la notation exacte est en fait :

```
template <class InputIterator, class UnaryFunction>
UnaryFunction for_each(InputIterator first, InputIterator last, UnaryFunction f);
```

- La notation simplifiée et les exemples fournis vous permettront d'utiliser ces algorithmes avec des conteneurs de type `vector` ou `array`.
- Vous comprendrez mieux la notation exacte après avoir vu la généricité (*chapitre 8*) et les différents types d'itérateurs (cours ASD).

for_each

```
for_each (first, last, fn);
```

- Applique une fonction à une série d'éléments
  - `first` et `last` définissent la plage à traiter
  - `fn` est la fonction à appliquer.
- Les codes suivants sont équivalents

```
for_each(first, last, fn);
```

```
for (auto i = first; i != last; ++i) {
   fn(*i);
}
```

- La séquence peut être modifiée si `fn` reçoit son paramètre par référence,[1] mais ce n'est pas l'intention première.
  Il est plus explicite d'utiliser `transform` pour cela (voir plus loin).

[1] Cela est visible dans son prototype, pas dans l'appel.

for_each (exemple)

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

void afficher(int i)     {cout << i << ' ';}
void incrementer(int& i) {++i;}

int main() {
   vector<int> v{2, 3, 5, 7, 11};
   for_each(v.begin(), v.end(), afficher);
   cout << endl;
   for_each(v.begin(), v.end(), incrementer);
   for_each(v.rbegin(), v.rend(), afficher);
}
```

```
2 3 5 7 11
12 8 6 4 3
```

count, count_if

```
int count    (first, last, value);
int count_if (first, last, unaryPredicate);
```

- Retournent le nombre d'éléments satisfaisant un critère donné
  - `first` et `last` définissent la plage à traiter
  - `value` est une valeur à *matcher* (nombre d'éléments égaux cette valeur)
  - `unaryPredicate` est une fonction unaire retournant un booléen

- Ces fonctions (algorithmes) sont respectivement équivalentes à :

```
int cnt = 0;
for (auto i = first; i != last; ++i)
   if (*i == value)
      cnt++;
return cnt;
```

```
int cnt = 0;
for(auto i = first; i != last; ++i)
   if (unaryPredicate(*i))
      cnt++;
return cnt;
```

count, count_if (exemple)

```
bool estImpair(int i) {return i % 2;}

int main() {
   vector<int> v{ 2, 3, 5, 7, 11};

   cout << "Le chiffre 7 apparait "
        << count(v.begin(), v.end(), 7)
        << " fois" << endl;

   cout << "Il y a "
        << count_if(v.begin(), v.end(), estImpair)
        << " nombres impairs" << endl;
}
```

```
Le chiffre 7 apparait 1 fois
Il y a 4 nombres impairs
```

all_of, any_of, none_of

```
bool all_of  (first, last, unaryPredicate);
bool any_of  (first, last, unaryPredicate);
bool none_of (first, last, unaryPredicate);
```

- Vérifient si le prédicat `unaryPredicate` est vrai pour tous / au moins un / aucun éléments de la plage `[first,last)`
  - `first` et `last` définissent la plage à traiter
  - `unaryPredicate` une fonction unaire renvoyant un booléen

- Exemple

```
vector<int> v{2, 3, 5, 7, 11};
cout << boolalpha
     << "Tous impairs: "
     << all_of(v.begin(), v.end(), estImpair)
     << "\nAu moins un impair: "
     << any_of(v.begin(), v.end(), estImpair)
     << "\nAucun impair: "
     << none_of(v.begin(), v.end(), estImpair);
```

```
Tous impairs: false
Au moins un impair: true
Aucun impair: false
```

equal

```
bool equal (first1, last1, first2);
bool equal (first1, last1, first2, binaryPredicate);
```

- Retourne *true* si les éléments sont les mêmes dans deux plages :
  l'une est `[first1,last1)` et l'autre est la plage commençant à `first2` avec le même nombre d'éléments que la première
  - `first1` et `last1` définissent la première plage
  - `first2` définit le début de la seconde plage
  - `binaryPredicate` est la fonction de comparaison : une fonction binaire renvoyant *true* si deux éléments sont égaux, *false* sinon

equal (exemple)

```
bool memeParite(int i, int j) {return i % 2 == j % 2;}

int main() {
   vector<int> v{2, 3, 5, 7, 11};
   vector<int> w{4, 5, 7, 11, 3};

   cout << boolalpha
        << "v et w sont egaux: "
        << equal(v.begin(), v.end(), w.begin())

        << "\nv et w sont de meme parite: "
        << equal(v.begin(), v.end(), w.begin(), memeParite)
        << endl;
}
```

```
v et w sont egaux: false
v et w sont de meme parite: true
```

Les opérateurs de comparaison

- Notons que pour la comparaison simple entre vecteurs, il est possible (et plus simple) d'utiliser les opérateurs de comparaison `<`, `>`, `<=`, `>=`, `==`, `!=`.
  Ils sont définis pour les classes `vector` et `array` et fonctionnent comme ceux de `string`.

- Par exemple, `v1 < v2` est équivalent à :

```
bool plusPetit(const vector<int>& v1, const vector<int>& v2) {
   auto i1 = v1.begin(), i2 = v2.begin();
   for ( ; i1 != v1.end() and i2 != v2.end(); ++i1, ++i2) {
      if (*i1 < *i2) return true;
      if (*i1 > *i2) return false;
      // *i1 == *i2, il faut vérifier le suivant
   }
   return (v1.size() < v2.size());
}
```

Les opérateurs de comparaison (exemple)

```
void afficher(const vector<int>& v) {
   cout << "[ ";
   for (auto i = v.begin(); i != v.end(); ++i) {
      if (i != v.begin()) cout << ", ";
      cout << *i;
   }
   cout << " ]";
}
void comparer(const vector<int>& v1, const vector<int>& v2) {
   afficher(v1);
   cout << (v1 == v2 ? " = " : v1 < v2 ? " < " : " > ");
   afficher(v2);
   cout << endl;
}
int main() {
    comparer({1, 2, 3}, {1, 2, 3});
    comparer({1, 1, 3}, {1, 2, 3});
    comparer({1, 3, 1}, {1, 2, 3});
    comparer({1, 2, 3, 4}, {1, 2, 3});
    comparer({1, 2, 2, 4}, {1, 2, 3});
    comparer({1, 2, 4, 4}, {1, 2, 3});
}
```

```
[ 1, 2, 3 ] = [ 1, 2, 3 ]
[ 1, 1, 3 ] < [ 1, 2, 3 ]
[ 1, 3, 1 ] > [ 1, 2, 3 ]
[ 1, 2, 3, 4 ] > [ 1, 2, 3 ]
[ 1, 2, 2, 4 ] < [ 1, 2, 3 ]
[ 1, 2, 4, 4 ] > [ 1, 2, 3 ]
```

Recherche

```
Iterator find        (first, last, value);
Iterator find_if     (first, last, unaryPredicate);
Iterator find_if_not (first, last, unaryPredicate);
```

- Retourne un itérateur du même type que `first` vers le premier élément qui a la même valeur que `value` / pour lequel `unaryPredicate` renvoie `true` / `false`
  - `first` et `last` définissent la plage
  - `value` est la valeur recherchée
  - `unaryPredicate` est une fonction unaire renvoyant un booléen
- Retourne `last` si la recherche est infructueuse.
  Attention, le comportement est différent de celui de `string::find`. Nous reviendrons sur ce point dans le chapitre 6.
- `find_first_of`, `adjacent_find`, `search`, `find_end`, et `search_n` proposent d'autres fonctionnalités de recherche de valeurs multiples, adjacentes, de séquences depuis le début ou la fin, ou des valeurs répétées

Recherche (exemple)

```
vector<int> v{2, 3, 5, 7, 11};
const int N = 3, M = -3;

auto it = find(v.begin(), v.end(), N);

if (it != v.end()) {
   cout << N << " trouvé et remplacé par " << M << endl;
   *it = M;
} else {
   cout << N << " pas trouvé \n";
}

for (int val : v) {
   cout << val << ' ';
}
```

```
3 trouvé et remplacé par -3
2 -3 5 7 11
```

min_element, max_element

```
Iterator min_element (first, last);
Iterator min_element (first, last, compare);
Iterator max_element (first, last);
Iterator max_element (first, last, compare);
```

- Une fonctionnalité particulière de recherche consiste à trouver le minimum / maximum d'une séquence
  - `first` et `last` définissent la plage
  - `compare` est une fonction binaire renvoyant un booléen qui indique si le premier paramètre est plus petit que le second

min_element, max_element (exemple)

```
bool abs_compare(int a, int b) {
    return abs(a) < abs(b);
}

int main() {
    vector<int> v{3, 1, -14, 1, 5, 9};

    auto resultat = max_element(v.begin(), v.end());
    cout << "élément max : " << *resultat
         << " , en position: "
         << distance(v.begin(), resultat) << endl;

    resultat = max_element(v.begin(), v.end(), abs_compare);
    cout << "élément max (absolu): " << *resultat
         << " , en position: "
         << distance(v.begin(), resultat);
}
```

```
élément max : 9 , en position: 5
élément max (absolu): -14 , en position: 2
```

Fonction définie dans `<iterator>`. Ici équivalente à : `resultat - v.begin()`

Modification de la séquence

- Voyons maintenant une série d'algorithmes qui modifient explicitement la séquence d'éléments qu'ils traitent
  - `transform`, l'équivalent modifiant de for_each
  - `fill` / `generate`, qui remplissent une séquence
  - `copy` et ses dérivés qui copient une séquence
  - `remove` et ses dérivés qui suppriment des éléments
  - `replace` et ses dérivés qui remplacent des éléments
  - `reverse` / `rotate` / `shuffle` qui réordonnent les éléments
  - `sort` et ses dérivés qui les trient

Attention !

- **La librairie <algorithm> fonctionne uniquement en recevant des itérateurs en paramètres.**
- Elle n'a donc pas accès au conteneur, mais peut uniquement parcourir ses données.
- Elle ne peut donc pas modifier les taille et capacité du conteneur, donc
  - Pour les méthodes remplissant le conteneur, il faut que l'emplacement pour les éléments existe avant d'appeler la fonction.
  - Pour les méthodes supprimant des éléments, il faut typiquement appeler `vector::erase(…)` après celles-ci. Pour ce faire, elles retournent un itérateur indiquant ce qu'il est possible de supprimer.

transform

```
Iterator transform (first1, last1, d_first, unary_op);
Iterator transform (first1, last1, first2, d_first, binary_op);
```

- Applique la fonction donnée à une ou deux séquences et stocke le résultat dans la plage commençant à d_first, qui peut être l'une ou l'autre des séquences originales
  - `first1` et `last1` définissent la première plage
  - `first2` définit le début de la seconde plage
  - `d_first` définit le début de la plage destination
  - `unaryOp / binaryOp` est une fonction unaire / binaire renvoyant une valeur stockable dans la séquence commençant en `d_first`

transform (exemples)

```
string s("hello");
transform(s.begin(), s.end(), s.begin(), ::toupper);
cout << s;
```

```
HELLO
```

Il existe plusieurs versions de cette fonction :
une dans std:: et une autre dans l'espace de nom global (noté ::).
Il est nécessaire de spécifier la version pour éviter une ambiguïté.

```
int mult(int a, int b) {
   return a * b;
}

int main() {
   vector<int> v{2, 3, 5, 7, 11};
   vector<int> w(v.size()); // pour stocker le résultat de transform
   transform(v.begin(), v.end(), v.rbegin(), w.begin(), mult);
   for (int val : w)
      cout << val << ' ';
}
```

```
22 21 25 21 22
```

fill / generate

```
void     fill       (first, last, value);
Iterator fill_n     (first, count, value);
void     generate   (first, last, generator);
Iterator generate_n (first, count, generator);
```

- Remplissent une plage avec une valeur donnée ou le résultat d'une fonction génératrice
  - `first` et `last` définissent la plage d'éléments
  - `first` et `count` définissent autrement une plage d'éléments, comme une plage de `count` éléments commençant à `first`
  - `value` est la valeur à copier dans la plage
  - `generator` est une fonction sans paramètre renvoyant les valeurs à copier dans la plage

```
vector<int> v(10);
// remplit le vector de nombres aléatoires
generate(v.begin(), v.end(), rand);
```

copy

```
Iterator copy          (first, last, d_first);
Iterator copy_if       (first, last, d_first, unaryPredicate);
Iterator copy_n        (first, count, d_first);
Iterator copy_backward (first, last, d_last);
```

- Copie une plage d'éléments dans une autre, éventuellement seulement certains sélectionnés par un prédicat
  - `first` et `last` ou `first` et `count` définissent la plage d'éléments à copier
  - `d_first` définit le début de la destination.
  - `d_last` définit la fin de la destination.
  - `unaryPredicate` est une fonction unaire qui retourne vrai s'il faut copier l'élément passé en paramètre
- Retourne un itérateur vers l'élément suivant le dernier élément copié dans la destination

copy (exemple)

```
bool estImpair(int i) {return i % 2;}

vector<int> v{2, 3, 6, 7, 11, 16};

vector<int> w1(v.size());
copy(v.begin(), v.end(), w1.begin());

vector<int> w2(v.size());
auto it = copy_if(v.begin(), v.end(), w2.begin(), estImpair);
w2.erase(it, w2.end());

vector<int> w3(v.size());
copy_n(v.begin(), 5, w3.begin());

vector<int> w4(10);
copy_backward(v.begin(), v.end(), w4.end());
```

```
w1: 2 3 6 7 11 16
```

```
w2: 3 7 11
```

```
w3: 2 3 6 7 11 0
```

```
w4: 0 0 0 0 2 3 6 7 11 16
```

remove

```
Iterator remove         (first, last, value);
Iterator remove_if      (first, last, unaryPredicate);
Iterator remove_copy    (first, last, d_first, value);
Iterator remove_copy_if (first, last, d_first, unaryPredicate);
```

- Supprime les éléments de [first, last) répondant à des critères spécifiques
  - `first` et `last` définissent la plage d'éléments à traiter
  - `value` définit la valeur à supprimer
  - `unaryPredicate` est une fonction unaire qui retourne vrai s'il faut supprimer l'élément passé en paramètre
  - `d_first` définit le début de la destination pour la copie
- La suppression se fait par déplacement des éléments suivants et écrasement de l'élément à supprimer
- Retourne la nouvelle fin de la plage

remove (exemple)

```
bool estImpair(int i) {return i % 2;}

vector<int> v{2, 3, 6, 2, 11, 16};
auto it = remove(v.begin(), v.end(), 2);

v.erase(it, v.end());

v = {2, 3, 6, 2, 11, 16};
it = remove_if(v.begin(), v.end(), estImpair);
v.erase(it, v.end());

v = {2, 3, 6, 2, 11, 16};
vector<int> w(v.size());
it = remove_copy(v.begin(), v.end(), w.begin(), 11);
w.erase(it, w.end());
```

```
v: 3 6 11 16 11 16
```

```
v: 3 6 11 16
```

```
v: 2 6 2 16
```

```
w: 2 3 6 2 16
```

replace

```
void     replace         (first, last, oldVal, newVal);
void     replace_if      (first, last, unaryPredicate, newVal);
Iterator replace_copy    (first, last, d_first, oldVal, newVal);
Iterator replace_copy_if (first, last, d_first, unaryPredicate, newVal);
```

- Remplace les éléments de [first, last) répondant à des critères spécifiques par une nouvelle valeur
  - `first` et `last` définissent la plage d'éléments à traiter
  - `oldVal` définit la valeur à remplacer
  - `newVal` définit la valeur qui la remplace
  - `unaryPredicate` est une fonction unaire qui retourne vrai s'il faut remplacer l'élément passé en paramètre
  - `d_first` définit le début de la destination pour la copie
- La version avec copie retourne la fin de la plage destination

reverse, rotate, shuffle

```
void reverse         (first, last);
Iterator reverse_copy (first, last, d_first);
```

- Inverse l'ordre des éléments dans `[first,last)`

```
Iterator rotate      (first, n_first, last);
Iterator rotate_copy (first, n_first, last, d_first);
```

- Effectue une rotation des éléments de `[first,last)` pour que `n_first` soit le nouveau premier élément et `n_first-1` le dernier.

```
void shuffle (first, last, generator);
```

- Réordonne les éléments de `[first,last)` dans un ordre aléatoire

et plus encore…

- Cette présentation ne couvre en fait qu'une partie des fonctions proposées par la bibliothèque `<algorithm>`.
- Vous trouverez encore plus de détails sur http://fr.cppreference.com/w/cpp/algorithm
- Le cours ASD présentera d'autres fonctions de cette librairie.

<numeric>

- La librairie `<numeric>` définit 5 algorithmes pour effectuer des opérations sur des séquences numériques. Leur flexibilité permet de les utiliser également dans d'autres contextes.
  - **`accumulate`** - calcule la somme des éléments
  - **`adjacent_difference`** – calcule les différences de toutes les paires d'éléments successifs
  - **`inner_product`** – calcule le produit scalaire de deux séquences
  - **`partial_sum`** – calcule les sommes partielles des sous-séquences
  - **`iota`** – remplit la séquence de valeurs incrémentales
- Voyons un exemple : `accumulate`

accumulate

```
T accumulate (first, last, T init);
T accumulate (first, last, T init, operation);
```

- Calcule la somme de `init` et des éléments de la plage `[first,last)`.
  Une autre opération que la somme peut être utilisée.
- Retourne la somme du même type que `init`

accumulate (exemple)

```
#include <numeric>
…
int mult(int a, int b) {return a * b;}

string dash(const string& a, int b) {
   return a.empty() ? to_string(b) : a + '-' + to_string(b);
}

int main() {
   vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
   int somme   = accumulate(v.begin(), v.end(), 0);
   int produit = accumulate(v.begin(), v.end(), 1, mult);
   string s    = accumulate(v.begin(), v.end(), string{}, dash);

   cout << "somme:   " << somme << endl
        << "produit: " << produit << endl
        << "chaine:  " << s << endl;
 }
```

```
somme:   55
produit: 3628800
chaine:  1-2-3-4-5-6-7-8-9-10
```

8. Résumé

Résumé

- C++ propose plusieurs alternatives pour stocker une séquence de données dans un tableau. Les tableaux classiques (« à la C ») et les conteneurs `vector`, et `array`.
- On peut toujours accéder aux éléments sans contrôle de borne avec l'opérateur `[]`. `vector` et `array` proposent également la méthode `at()` qui lève une exception en cas de dépassement de borne.
- Le passage d'un tableau classique en paramètre de fonction est délicat. La taille doit être fournie dans un paramètre annexe. Le passage des conteneurs s'effectue comme avec toute autre variable.

Résumé

- Il est utile de séparer les notions de taille (le nombre d'éléments) et de capacité (le nombre d'emplacements disponibles en mémoire) d'un tableau classique.
- Insérer et supprimer un ou des éléments ailleurs qu'en fin de tableau nécessite une boucle qui parcourt le tableau.
- Le conteneur `vector` est un tableau dont la capacité peut varier si nécessaire.

Résumé

- L'accès aux éléments de `vector` (et `array`) se fait via
  - L'opérateur `[]` combiné avec la méthode size()
  - La méthode `at()`
  - La boucle `for( : )` sur un conteneur
  - Les itérateurs avec `begin()` et `end()`
- On peut modifier la taille d'un `vector` explicitement (`resize`, `clear`) ou implicitement (`push_back`, pop_back, `insert`, `erase`).
- On peut gérer la capacité d'un `vector` avec `reserve`, `capacity`, `shrink_to_fit`.
- Il est également possible de définir `vector` (et `array`) à plusieurs indices. `typedef` ou `using` simplifient le code.

Résumé

- Il existe divers algorithmes de tri.
  - Ici, seuls le tri à bulles, le tri par insertion et le tri par sélection ont été examinés.
  - D'autres algorithmes de tri seront présentés dans le cours ASD.
- Chaque algorithme se caractérise par diverses propriétés :
  sa complexité (en temps), sa stabilité, son efficacité mémoire…
  - La complexité d'un algorithme de tri peut être en $O(n)$, $O(n \cdot \log(n))$ ou $O(n^2)$
  - Elle traduit le lien de proportionnalité entre le temps d'exécution de l'algorithme et le nombre d'éléments *n* à trier.
  - Les tri à bulles, par insertion et par sélection sont tous en $O(n^2)$.

Résumé

- La librairie `<algorithm>` fournit de nombreux algorithmes qui s'appliquent sur des plages d'éléments définies entre deux itérateurs `[first,last)`.
- Le passage de fonctions en paramètre permet une grande flexibilité de ces algorithmes.
- Les algorithmes n'ont pas accès aux conteneurs mais uniquement à la plage d'éléments. Il faut donc redimensionner explicitement ces conteneurs avant ou après l'application des algorithmes tels que `transform`, `generate`, `remove`,...
- La librairie `<numeric>` offre des fonctionnalités additionnelles.