Programmation Orientée Objet : Structures de données abstraites et Bibliothèques

Jean-Cédric Chappelier

Laboratoire d'Intelligence Artificielle
Faculté I&C

Objectifs des deux derniers cours

L'objectif de ces deux derniers cours est :

1. de compléter le cours d'ICC du premier semestre sur un point qui (à mon avis) lui manquait :
   les **structures de données abstraites** :
   - *listes chaînées*
   - *piles*

2. de vous présenter (sommairement) un certain nombre d'**outils** standards existant en C++ (« bibliothèque standard »)

   Le but ici n'est pas d'être exhaustif, mais simplement de vous :
   - informer de l'existence des **principaux** outils
   - faire prendre conscience d'aller **lire/chercher dans la documentation** les éléments qui peuvent vous être utiles

Jean-Cédric Chappelier
& Jamila Sam

Pourquoi modéliser les données ?

L'élaboration d'un algorithme est grandement facilitée par l'utilisation de **structures de données abstraites**, de plus haut niveau, et de fonctions de manipulations associées.

Une structure de données doit *modéliser au mieux les informations* à traiter pour en formaliser les traitements (complément des algorithmes).

Choisir les bons modèles de données *fait partie* du choix de bons algorithmes : algorithmes et structures de données abstraites sont intimement liés.

Jean-Cédric Chappelier
& Jamila Sam

C'est quoi une « structure de données abstraite » ?

Une **structure de données abstraite** (S.D.A.) est un ensemble organisé d'informations (ou données) reliées logiquement et pouvant être manipulées non seulement individuellement, mais aussi comme un tout.

Exemples généraux :

tableau (au sens général du terme)
contenu : divers éléments de types à préciser
interactions : demander la taille du tableau, accéder (lecture/écriture) à chaque élément individuellement, ...

vecteur (au sens général, pas C++) : formalisation mathématique d'espace vectoriel sur un corps $\mathscr{K}$
contenu : n coordonnées (éléments de $\mathscr{K}$)
interactions : les propriétés élémentaires définissant un espace vectoriel

Jean-Cédric Chappelier
& Jamila Sam

C'est quoi une « structure de données abstraite » ?

Une **structure de données abstraite** (S.D.A.) est un ensemble organisé d'informations (ou données) reliées logiquement et pouvant être manipulées non seulement individuellement, mais aussi comme un tout.

Exemple informatique élémentaire :

Vous connaissez déjà des structures de données abstraites, très simples : les **types élémentaires**.

Par exemple, un `int`
interactions : affectation, lecture de la valeur, `+`, `-`, `*`, `/`

Spécifications des structures de données abstraites

Une S.D.A. est caractérisée par :

- son contenu
- les interactions possibles (manipulation, accès, ...)

Du point de vue informatique, une structure de données abstraite peut être spécifiée à deux niveaux :

- niveau fonctionnel / logique : spécification formelle des données et des **algorithmes** de manipulation associés
- niveau physique (programmation) : comment est implémentée la structure de données abstraite dans la mémoire de la machine

  ☞ déterminant pour l'efficacité des **programmes** utilisant ces données.

Spécifications des S.D.A. [2]

Au niveau formel (modèle), on veut généraliser cette idée « d'objets » manipulables par des opérateurs propres, sans forcément en connaître la structure interne et encore moins l'implémentation.

Par exemple, vous ne pensez pas un `int` comme une suite de 32 bits, mais bien comme un « entier » (dans un certain intervalle) avec ses opérations propres : `+`, `-`, `*`, `/`

Une structure de données abstraite définit une abstraction des données et **cache les détails de leur implémentation**.

abstraction : identifier précisément les **caractéristiques** de l'entité (par rapport à ses applications), et en décrire les **propriétés**.

Jean-Cédric Chappelier
& Jamila Sam

Spécifications des S.D.A. [3]

Une structure de données abstraite modélise donc l'« **ensemble des services** » désirés (interface) plutôt que l'organisation intime des données (détails d'implémentation)

On identifie usuellement 4 types de « services » :

1. les sélecteurs ou accesseurs, qui permettent « d'interroger » la S.D.A.
2. les modificateurs, qui modifient la S.D.A.
3. les itérateurs, qui permettent de parcourir la structure
4. les constructeurs (pour l'initialisation)

Exemple : *tableau dynamique*
modifieur : ajout d'un élément (`push_back(a)`)
sélecteur : lecture d'un élément (`t[i]`)
sélecteur : le tableau est-il vide ? (`t.empty()`)
itérateur : index d'un élément (`[i]` ci-dessus)

Divers exemples de S.D.A.

Il y a beaucoup de structures de données abstraites en Informatique :

- tableaux (d'au moins quatre sortes ; cf MOOC 1er semestre)
- listes
- piles
- files d'attente (avec ou sans priorité)
- tables associatives (dites « de hachage »)
- multi-listes
- arbres (pleins de sorte...)
- graphes

Beaucoup sont offertes dans les *bibliothèques* de C++.
Vous avez déjà vu :

- les tableaux (de taille fixe, dynamiques) ;
- les chaînes de caractères.

Divers exemples de S.D.A.

Dans ce cours, nous allons tout d'abord détailler les **deux** plus fondamentales après les *tableaux* :

- les **listes**
- et les **piles**

puis nous en présenterons rapidement d'autres, ainsi que d'autres aspects des bibliothèques de C++.

Liste chaînées

Les listes chaînées sont (comme les tableaux)
des SDA **séquentielles**, c'est-à-dire stockant des séquences d'éléments.

Par contre, dans une liste chaînée, *l'accès direct à un élément n'est pas possible* (contrairement aux tableaux).

Spécification logique :
Ensemble d'éléments *successifs* (sans d'accès direct), ordonnés ou non

Interactions :

- accès au premier élément (sélecteur)
- accès à l'élément suivant d'un élément (sélecteur)
- modifier l'élément courant (modificateur)
- insérer/supprimer un élément après(/avant) l'élément courant (modificateur)
- tester si la liste est vide (sélecteur)
- parcourir la liste (itérateur)

Jean-Cédric Chappelier
& Jamila Sam

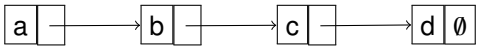
Listes

Exemple concret :

visionneuse stéreo (essayez d'accéder à la 3e image directement, sans passer par la 2e !)

Exemple informatique :

(a (b (c (d ()))))



Une liste peut être vu comme une structure récursive :

liste = valeur + liste OU liste = vide

Jean-Cédric Chappelier
& Jamila Sam

Réalisations d'une liste

Implémentations possibles :

- tableau dynamique (`vector`) (mais inconvénient 1 ci-après)
- classe :

```
class Element;
typedef Element* ListeChainee;

class Element {
private:
  type_el valeur;
  ListeChainee suite;
};
```

Note : Ce « truc » (prédéclaration), utilisé ici pour clarifier les concepts, est très utile en cas de dépendances cycliques entre données : `A` utilise des (pointeurs sur des) `B`s lesquels utilisent des (pointeurs sur des) `A`s.

Jean-Cédric Chappelier
& Jamila Sam

Pourquoi les listes dynamiques ?

Les tableaux sont un type de données très utile en programmation
mais présentent **2 limitations** :

1. les données sont *contiguës* (les unes derrières les autres)
   et donc l'insertion d'un nouvel élément au milieu du tableau demande la recopie (le décalage) de tous les éléments suivants.
   $\Longrightarrow$ **insertion en** $\mathcal{O}(n)$
2. augmenter la taille (lorsqu'elle n'est pas fixée) peut nécessiter la création d'un nouveau tableau
   $\Longrightarrow$ $\mathcal{O}(n)$

Jean-Cédric Chappelier
& Jamila Sam

Complexité optimale des opérations élémentaires

	Liste	tableau
accès à un élément précis :	$\mathcal{O}(n)$	$\mathcal{O}(1)$
insérer un élément (quand on connait sa place !) :	$\mathcal{O}(1)$	$\mathcal{O}(n)$
supprimer un élément :	$\mathcal{O}(1)$	$\mathcal{O}(n)$
parcourir la S.D.A. :	$\mathcal{O}(n)$	$\mathcal{O}(n)$
calculer la longueur :	$\mathcal{O}(n)$	$\mathcal{O}(n)$

(voire $\mathcal{O}(1)$ si le stockage de cette valeur est effectué, en particulier si « longueur » a été spécifiée dans les « services » de la SDA.
C'est par exemple le cas pour le `vector` et les `array` où `size()` est $\mathcal{O}(1)$.
Ce n'est pas forcément le cas pour `list` ou `forward_list` car avoir `size()` en $\mathcal{O}(1)$ peut avoir d'autres conséquences ; et ce ne serait en tout cas plus une liste chaînée telle que décrite dans ce cours !)

Jean-Cédric Chappelier
& Jamila Sam

Listes chaînées en C++

Les listes (simplement) chainées existent depuis C++11 dans la bibliothèque `forward_list`.

Note : Les listes *doublement* chainées existent depuis C++98 : `list`

(quelques) méthodes des listes chaînées :

`Type& front()`	retourne le premier élément de la liste
`void push_front(Type)`	ajoute un élément en tête de liste
`void pop_front()`	supprime le premier élément
`void insert(iterator, Type)`	insertion avant un élément de la liste désigné par un itérateur

Exemple :

```
#include <forward_list>

forward_list<int> ma_liste({ 6, 1, 5, -23, 3 });

for (auto element : ma_liste) {
  cout << element << endl;
}

ma_liste.push_front(877);
```

Piles

Spécification :
Une pile est une structure de données abstraite dynamique à 1 point d'accès, contenant des éléments homogènes, et permettant :

- d'ajouter une valeur à la pile (empiler ou push) ;
- de lire la **dernière** valeur ajoutée ;
- d'enlever la dernière valeur ajoutée (dépiler ou pop) ;
- de tester si la pile est vide.

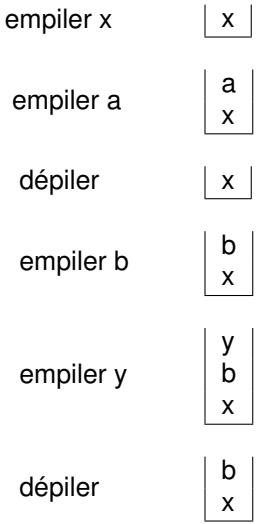
On ne « connaît » donc de la pile **que** le dernier élément empilé (son sommet).

Exemples concrets :

- une pile d'assiettes
- une tour dans le jeu des tours de Hanoï

Jean-Cédric Chappelier
& Jamila Sam

Piles : exemple d'utilisation



Exemples d'utilisation des piles

Le problème des parenthèses : étant donnée une expression avec des parenthèses, est-elle bien ou mal parenthésée ?

$$((a+b)\times c-(d+4)\times(5+(a+c)))\times(c+(d+(e+5\times g)\times f)\times a)$$

(correct)

$$(a+b)($$

(incorrect)

Encore un peu plus complexe : différentes parenthèses
Exemple avec [et (

([])[()(()[])] ☞ correct
([)] ☞ incorrect

Autres exemples d'utilisation des piles (non traités ici) :

- tours de Hanoï
- notation postfixée (ou « polonaise inverse ») :

  $4\ 2\ +\ 5\ \times$

  (☞ $5\times(4+2)$)

Jean-Cédric Chappelier & Jamila Sam

Vérification de parenthésage

Tant que lire caractère c
  Si c est `(` ou `[`
    empiler c
  Sinon
    Si c est `)` ou `]`
      Si pile vide
        ÉCHEC
      Sinon
        c' ← lire la pile
        Si c et c' correspondent
          dépiler
        Sinon
          ÉCHEC
Si pile vide
  OK
Sinon
  ÉCHEC

Exemple

Entrée : `([)]`

empile `(`

empile `[`

lu = `)`, top = `[`
$\rightarrow$ ne correspond pas
$\Longrightarrow$ ERREUR

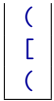
Deuxième Exemple

Entrée : `([()])`

empile `(`

empile `[`

empile `(`



lu `)` → correspond ⟹ dépile

lu `]` → correspond ⟹ dépile

lu `)` → correspond ⟹ dépile

pile vide ⟹ **OK**

Piles (et files) en C++

Pour utiliser les piles en C++ : `#include <stack>`

Les **files d'attente** sont des piles où c'est le premier arrivé (empilé) qui est dépilé le premier. Elles sont définies dans la bibliothèque `<queue>`.

Une pile de type *type* se déclare par `stack<`*type*`>` et une file d'attente par `queue<`*type*`>`. Par exemple :

```
stack<double> une_pile;
queue<char> attente;
```

Méthodes :

Type `top()`	accède au premier élément (sans l'enlever)
`void push(`*Type*`)`	empile/ajoute
`void pop()`	dépile/supprime
`bool empty()`	teste si la pile/file est vide

Jean-Cédric Chappelier
& Jamila Sam

Code C++ de l'exemple

```
#include <stack>

bool check(string const& s) {
  stack<char> p;
  for (auto const& c : s) {
    if ((c == '(') or (c == '['))
      p.push(c);
    else if (c == ')') {
      if ((not p.empty()) and (p.top() == '('))
        p.pop();
      else
        return false;
    } else if (c == ']') {
      if ((not p.empty()) and (p.top() == '['))
        p.pop();
      else
        return false;
    }
  }
  return p.empty();
}
```


Jean-Cédric Chappelier
& Jamila Sam

Bibliothèque standard

La bibliothèque standard (d'outils) C++ **facilite la programmation** et permet de la rendre **plus efficace**, si tant est que l'on connaisse bien les outils qu'elle fournit.

Cette bibliothèque est cependant vaste et complexe, mais elle peut dans la plupart des cas s'utiliser de façon très simple, facilitant ainsi la réutilisation des structures de données abstraites et des algorithmes sophistiqués qu'elle contient.

La bibliothèque standard C++17 est formée de 96 « paquets » :

- 32 « d'origine » (C++89)
- 43 « nouveaux » : 19 de C++11, 1 de C++14, 8 de C++17 et 15 de C++20
- et 21 bibliothèques du C

Contenu de la bibliothèque standard

La bibliothèque standard C++ contient 32 « paquets » du C++ d'origine :

`<algorithm>`	plusieurs algorithmes utiles
`<bitset>`	gestions d'ensembles de bits
`<complex>`	les nombres complexes
`<deque>`	tableaux dynamiques avec `push_front`
`<exception>`	diverses fonctions aidant à la gestion des exceptions
`<fstream>`	manipulation de fichiers
`<functional>`	objets fonctions
`<iomanip>`	manipulation de l'état des flots
`<ios>`	définitions de base des flots
`<iosfwd>`	anticipation de certaines déclarations de flots
`<iostream>`	flots standards
`<istream>`	flots d'entrée
`<iterator>`	itérateurs
`<limits>`	diverses bornes concernant les types numériques
`<list>`	listes doublement chaînées
`<locale>`	contrôles liés au choix de la langue

Contenu de la bibliothèque standard (2)

`<map>`	tables associatives clé–valeur ordonnées
`<memory>`	gestion mémoire pour les containers
`<new>`	gestion mémoire
`<numeric>`	fonctions numériques
`<ostream>`	flots de sortie
`<queue>`	files d'attente
`<set>`	ensembles ordonnés
`<sstream>`	flots dans des chaînes de caractères
`<stack>`	piles
`<stdexcept>`	gestion des exceptions
`<streambuf>`	flots avec tampon (buffer)
`<string>`	chaînes de caractères
`<typeinfo>`	information sur les types
`<utility>`	divers utilitaires
`<valarray>`	tableaux orientés vers les valeurs
`<vector>`	tableaux dynamiques

Contenu de la bibliothèque standard (3)

19 « paquets » de C++11 :

`<array>`	tableaux de taille fixe
`<atomic>`	expression atomique
`<chrono>`	heures et chronomètres
`<condition_variable>`	concurence (multi-thread)
`<forward_list>`	listes simplement chaînées
`<future>`	concurence (multi-thread)
`<initializer_list>`	listes d'initialisation
`<mutex>`	concurence (multi-thread)
`<random>`	nombres aléatoires
`<ratio>`	constantes rationnelles ($\mathbb{Q}$)
`<regex>`	expressions régulières
`<scoped_allocator>`	allocation mémoire
`<system_error>`	erreurs système
`<thread>`	concurence (multi-thread)
`<tuple>`	*n*-uples
`<type_traits>`	caractéristiques de types
`<typeindex>`	utiliser les types comme index de containers
`<unordered_map>`	tables associatives non ordonnées
`<unordered_set>`	ensembles non ordonnés

Contenu de la bibliothèque standard (4)

1 « paquet » de C++14 :

`<shared_mutex>`	programmation concurrente

8 « paquets » de C++17 :

`<any>`	« fourre-tout » (une valeur de n'importe quel type)
`<charconv>`	conversion de séquence de caractères en valeur numérique
`<execution>`	programmation parallèle
`<filesystem>`	gestion du système de fichiers
`<memory_resource>`	gestion mémoire polymorphique
`<optional>`	valeur possiblement absente
`<string_view>`	« vues » sur des chaînes de caractères
`<variant>`	« fourre-tout » parmi des types choisis

Jean-Cédric Chappelier
& Jamila Sam

Contenu de la bibliothèque standard (5)

15 « paquets » de C++20 :

`<barrier>`	« barrières » (pour les threads)
`<bit>`	manipulation de bits
`<compare>`	« 3-way comparison » et ordres (mathématiques)
`<concepts>`	« *equationnal reasoning* » (propriétés syntaxiques et sémantiques)
`<coroutine>`	co-routines (fonctions en attente)
`<format>`	formater plus facilement du texte
`<latch>`	« verrous » (pour les threads)
`<numbers>`	diverses constantes
`<ranges>`	« intervalles » d'itérateurs
`<semaphore>`	« sémaphores » (pour les threads)
`<source_location>`	informations sur le code source
`<span>`	non-owning view over a contiguous sequence of objects
`<stop_token>`	pour stopper les threads
`<syncstream>`	sorties (flots) synchrones
`<version>`	informations sur les outils disponibles

Jean-Cédric Chappelier
& Jamila Sam

Contenu de la bibliothèque standard (6)

Il existe aussi dans les outils standards les 21 « paquets» venant du langage C :

`<cassert>`	test d'invariants lors de l'exécution
`<cctype>`	diverses informations sur les caractères
`<cerrno>`	code d'erreurs retournés dans la bibliothèque standard
`<cfenv>`	manipulation des règles de gestion des nombres en virgule flottante
`<cfloat>`	diverses informations sur la représentation des réels
`<cinttypes>`	`int` de taille fixée (C99)
`<climits>`	diverses informations sur la représentation des entiers
`<clocale>`	adaptation à diverses langues
`<cmath>`	diverses définitions mathématiques
`<csetjmp>`	branchement non locaux
`<csignal>`	contrôle des signaux (processus)
`<cstdarg>`	nombre variable d'arguments
`<cstddef>`	diverses définitions utiles (types et macros)
`<cstdio>`	entrées sorties de base
`<cstdint>`	sous-partie de `cinttypes`
`<cstdlib>`	diverses opérations de base utiles
`<cstring>`	manipulation des chaînes de caractères à la C

Contenu de la bibliothèque standard (7)

`<ctime>`	diverses conversions de date et heures
`<cuchar>`	char de 16 ou 32 bits
`<cwchar>`	utilisation des caractères étendus
`<cwctype>`	classification des codes de caractères étendus

Outils standards

On distingue plusieurs types d'outils.

Parmi les principaux :

- les containers de base
- les containers avancés (appelés aussi « adaptateurs »)
- les itérateurs
- les algorithmes
- les outils numériques
- les traitements d'erreurs
- les chaînes de caractères
- les flots

Outils standards (2)

Les outils les plus utilisés par les débutants sont :

- les chaînes de caractères (`string`) ✔
- les flots (`stream`) ✔
- les tableaux dynamiques (`vector`) [container] ✔
- les listes chaînées (`list`) [container avancé] ✔
- les piles (`stack`) [container avancé] ✔
- les algorithmes de tris (`sort()`)
- les algorithmes de recherche (`find()`)
- les itérateurs (`iterators`)

Plan

Présentons maintenant certains des outils standards de façon plus détaillée :

- `set`/`unordered_set` [container]
- `iterator`
- `map`/`unordered_map` [container]
- `sort`
- `find`
- `complex`
- `cmath`
- `random`

Jean-Cédric Chappelier
& Jamila Sam

Containers

Comme le nom l'indique, les containers sont des structures de données abstraites servant à **contenir** (« collectionner ») **d'autres objets**.

Vous en connaissez déjà plusieurs : tableaux, listes chaînées, piles et files d'attentes.

Il en existe plusieurs autres, parmi lesquels les ensembles (`set`, `unordered_set`) et les tables associatives (`map`, `unordered_map`).

Les `set` permettent de gérer des **ensembles** au sens mathématique du terme (finis et ordonnés) : collection d'éléments où chaque élément n'est présent qu'une seule fois.

Les **tables associatives** sont une généralisation des tableaux où les indexes ne sont pas forcément des entiers.

Imaginez par exemple un tableau que l'on pourrait indexer par des chaînes de caractères et écrire par exemple `tab["Informatique"]`

Containers (2)

(Presque) Tous les containers contiennent les méthodes suivantes :

`bool empty()` : le containers est-il vide ?

`size_t size()` : nombre d'éléments contenus dans le container
(sauf `forward_list`)

`void clear()` : vide le container
(sauf `array`, `stack`, `queue` et `priority_queue`)

`iterator erase(`it`)` : supprime du container l'élément pointé par it. it est un itérateur (généralisation de la notion de pointeur, voir quelques transparents plus loin)
(sauf `forward_list`, `array`, `stack`, `queue` et `priority_queue`)

Ils possèdent également tous (sauf `stack`, `queue` et `priority_queue`) les méthodes `begin()`, `cbegin()`, `end()` et `cend()` que nous verrons avec les itérateurs.

Jean-Cédric Chappelier
& Jamila Sam

Tableaux dynamiques : petit complément

Pour accéder directement à un élément d'un tableau dynamique (`vector`) on utilise l'opérateur `[]` : `tab[i]`.

Il existe une autre méthode pour cet accès : `at(`*n*`)` qui, à la différence de `[`*n*`]`, lance l'exception `out_of_range` (de la bibliothèque `<stdexcept>`) si `n` n'est pas un index correct.

Exemple :

```
#include <vector>
#include <stdexcept>
...
vector<int> v(5,3); // 3, 3, 3, 3, 3
int n(12);
try {
   cout << v.at(n) << endl;
}
catch (out_of_range) {
   cerr << "Erreur : " << n << " n'est pas correct pour v"
        << endl
        << "qui ne contient que " << v.size() << " éléments."
        << endl;
}
```

Ensembles

Les ensembles (au sens mathématique) sont implémentés dans la bibliothèque `<set>`. Ils contiennent des éléments d'un même type, ordonnés par `operator<`.

(Pour des éléments (d'un même type) non ordonnés, c.-à-d. *sans* `operator<`, on utilisera un `unordered_set`.)

On déclare un ensemble comme les autres containers, en spécifiant le type de ses éléments, par exemple :

```
set<char> monensemble;
```

Les ensembles sont une SDA **non-indexée** : l'accès direct à un élément n'est pas possible.

(quelques) méthodes des ensembles :

`insert(`*Type*`)`	insère un élément s'il n'y est pas déjà
`erase(`*Type*`)`	supprime l'élément (s'il y est)
`find(`*Type*`)`	retourne un itérateur indiquant l'élément recherché

À noter que la bibliothèque `<algorithm>` fournit des fonctions pour faire la réunion, l'intersection et la différence d'ensembles.

Ensembles – Exemple

```
#include <set>
...
set<char> voyelles;

voyelles.insert('a');
voyelles.insert('b');
voyelles.insert('e');
voyelles.insert('i');
voyelles.erase('b');
voyelles.insert('e'); // n'insere pas 'e' car il y est déjà
```

Comment parcourir cet ensemble ?

```
for (size_t i(0); i < voyelles.size(); ++i)
    cout << voyelles[i] << endl;
```

ne fonctionne pas car c'est une SDA non-indexée.

Jean-Cédric Chappelier
& Jamila Sam

Ensembles – parcours

Comment parcourir cet ensemble ?

Depuis C++11 c'est facile :

```
for (auto const v : voyelles) {
    cout << v << endl;
}
```

Il y a aussi un autre moyen, plus avancé :

☞ utilisation d'**itérateurs**

Jean-Cédric Chappelier
& Jamila Sam

Itérateurs

Les **itérateurs** sont une SDA généralisant aux **containers** d'une part les accès par index et d'autre part les pointeurs.

Ils permettent :

- de parcourir de façon itérative les containers
- d'indiquer (c.-à-d. de pointer sur) un élément d'un container

Il existe en fait 7 sortes d'itérateurs, mais nous ne parlons ici que de la plus générale, qui permet de tout faire : lecture et écriture du container, aller en avant ou en arrière (accès quelconque en fait).

Itérateurs (2)

Un itérateur associé à un container *C<type>* se déclare simplement comme

C<type>`::iterator` *nom*`;`

ou *C<type>*`::const_iterator` *nom*`;`

si l'on ne modifie pas le container lors du parcours.

Exemples :

```
vector<double>::iterator i;
set<char>::const_iterator j; // en lecture seule
```

Il peut s'initialiser grâce aux méthodes `begin()`, `cbegin()`, `end()` ou `cend()` du container, voire d'autres méthodes spécifiques, comme par exemple `find()` pour les containers non-séquentiels.

Exemples :

```
vector<double>::iterator i(monvect.begin());
set<char>::const_iterator j(monset.find(monelement));
```

L'élément indiqué par l'itérateur `i` est simplement `*i`, comme pour les pointeurs.

Retour sur l'exemple des ensembles

Pour parcourir notre ensemble précédent, nous pouvons donc faire :

```
for (set<char>::const_iterator i(voyelles.cbegin());
     i != voyelles.cend(); ++i) {
    cout << *i << endl;
}
```

Exemple d'utilisation de `find()` :

```
set<char>::const_iterator i(voyelles.find('c'));
if (i == voyelles.cend()) {
    cout << "pas trouvé" << endl;
} else {
    cout << *i << " trouvé" << endl;
}
```


Jean-Cédric Chappelier
& Jamila Sam

Code complet de l'exemple

```
#include <set>
#include <iterator>
#include <iostream>
using namespace std;

int main() {
  set<char> voyelles;

  voyelles.insert('a');
  voyelles.insert('b');
  voyelles.insert('e');
  voyelles.insert('i');
  voyelles.insert('a'); // ne fait rien car 'a' y est déjà
  voyelles.erase('b');  // supprime 'b'

  // parcourt l'ensemble
  for (auto const v : voyelles)  cout << v << endl;

  // recherche d'un élément
  set<char>::const_iterator element(voyelles.find('c'));
  if (element == voyelles.cend())
    cout << "je n'ai pas trouvé." << endl;
  else
    cout << *element << " trouvé !" << endl;

  return 0;
}
```


Jean-Cédric Chappelier
& Jamila Sam
EPFL

Suppression d'un élément d'un container

On a vu que tout container possédait une méthode

```
iterator erase(it)
```

permettant de supprimer un élément, mais...

Attention ! On **ne** peut **pas** continuer à utiliser l'itérateur it !
(plus exactement : `erase` rend invalide tout itérateur et référence situé(e) au dela du premier point de suppression)

Exemple d'**erreur** classique : ceci **n'**est **pas** correct («`Segmentation fault`») :

```
vector<double> v;
...
for (vector<double>::iterator i(v.begin()); i != v.end(); ++i)
    if (cond(*i)) v.erase(i);
```

(avec `bool cond(double);`)

pas plus que :

```
for (vector<double>::iterator i(v.begin()); i != v.end(); ++i)
    if (cond(*i)) i = v.erase(i);
```


Jean-Cédric Chappelier
& Jamila Sam
EPFL

Suppression d'un élément d'un container (2)

Ce qu'il faut faire c'est :

```
vector<double>::iterator next;
for (vector<double>::iterator i(v.begin()); i != v.end(); i = next) {
  if (cond(*i)) { next = v.erase(i); }
  else          { next = i + 1;      }
}
```

ou mieux en utilisant `remove_if` (ou `remove`) de `<algorithm>` :

```
v.erase(remove_if(v.begin(), v.end(), cond), v.end());
```

mais qui sont de toutes façons «**coûteux**» ($\mathcal{O}(\texttt{v.size()}^2)$)

En effet, un tableau dynamique **n'est pas la bonne SDA** si l'on veut détruire un élément au milieu **et** garder l'ordre (☞ *listes chaînées*)

Note : si l'on ne tient pas à garder l'ordre, on peut toujours faire :

```
for (size_t i(0); i < v.size(); ++i)
    if (cond(v[i])) {
        swap(v[i], v.back());
        v.pop_back();
        --i;
    }
```

Tables associatives

Les **tables associatives** sont une généralisation des tableaux où les indexes ne sont pas forcément des entiers.

Imaginez par exemple un tableau que l'on pourrait indexer par des chaînes de caractères et écrire par exemple `tab["Informatique"]`

On parle d'« associations clé–valeur »

Les tables associatives sont définies dans la bibliothèque `<map>`.

Elles nécessitent deux types pour leur déclaration :
le type des « clés » (les indexes) et le type des éléments indexé.

Par exemple, pour indexer des nombres réels par des chaînes de caractères on déclarera :

```
map<string,double> une_variable;
```

Si l'ordre (`operator<`) des clés n'importe pas, on utilisera une `unordered_map`.

Tables associatives : exemple : VERSION C++98

```cpp
#include <map>
#include <string>
#include <iostream>
using namespace std;

int main() {
  map<string,double> moyenne;

  moyenne["Informatique"]       = 5.5;
  moyenne["Physique"]           = 4.5;
  moyenne["Histoire des maths"] = 2.5;
  moyenne["Analyse"]            = 4.0;
  moyenne["Algèbre"]            = 5.5;

  // parcours de tous les éléments
  for (map<string,double>::iterator i(moyenne.begin());
       i != moyenne.end(); ++i) {
    cout << "En " << i->first << ", j'ai " << i->second
         << " de moyenne." << endl ;
  }

  // recherche
  const map<string,double>& m = moyenne; // pour forcer le const
  cout << "Ma moyenne en Informatique est de ";
/* cout << m["Informatique"] << endl; // ne compile pas parce que m[] est non const :
                                      //   il insère si l'élément n'est pas présent.  */
  cout << m.find("Informatique")->second << endl;

  return 0;
```

Tables associatives : exemple VERSION C++11

```cpp
#include <map>
#include <string>
#include <iostream>
using namespace std;

int main() {
  map<string,double> moyenne;

  moyenne["Informatique"]       = 5.5;
  moyenne["Physique"]           = 4.5;
  moyenne["Histoire des maths"] = 2.5;
  moyenne["Analyse"]            = 4.0;
  moyenne["Algèbre"]            = 5.5;

  // parcours de tous les éléments
  for (auto i : moyenne) {
    cout << "En " << i.first << ", j'ai " << i.second
         << " de moyenne." << endl ;
  }


  // recherche
  const map<string,double>& m = moyenne;
  cout << "Ma moyenne en Informatique est de "
       << m.at("Informatique") << endl;

  return 0;
}
```

Tables associatives : exemple VERSION C++17

```cpp
#include <map>
#include <string_view>
#include <iostream>
using namespace std;

int main() {
  map<string_view,double> moyenne;

  moyenne["Informatique"]       = 5.5;
  moyenne["Physique"]           = 4.5;
  moyenne["Histoire des maths"] = 2.5;
  moyenne["Analyse"]            = 4.0;
  moyenne["Algèbre"]            = 5.5;

  // parcours de tous les éléments
  for (auto&& [ cours, note ] : moyenne) {
    cout << "En " << cours << ", j'ai " << note
         << " de moyenne." << endl ;
  }


  // recherche
  const map<string_view,double>& m = moyenne;
  cout << "Ma moyenne en Informatique est de "
       << m.at("Informatique") << endl;

  return 0;
}
```

Algorithmes

La bibliothèque `algorithm` (c.-à-d. `#include <algorithm>`) fournit l'implémentation d'un grand nombre d'algorithmes généraux :

- de séquencement
  quelques exemples : `for_each`, `find`, `copy`
- de tris
  `sort`, mais aussi bien d'autres
- numériques
  `inner_product`, `partial_sum`, `adjacent_difference`
- ...

3 exemples ici :

- `find`
- `copy` et les `output_iterators`
- `sort`

pour les autres : référez-vous à la documentation

find()

`find()` implémente un algorithme général permettant de faire des recherches dans (une partie d')un container.
Son prototype général est :

```
iterator find(iterator debut, iterator fin, Type valeur);
```

qui cherche `valeur` entre `debut` (inclu) et `fin` (exclu). Elle retourne un itérateur sur le contenu correspondant à la valeur recherchée ou `fin` si cette valeur n'est pas trouvée.

Exemple :

```
list<int> uneliste;

uneliste.push_back(3);
uneliste.push_back(1);
uneliste.push_back(7);

list<int>::iterator result(find(uneliste.begin(), uneliste.end(), 7));

if (result != uneliste.end()) cout << "trouvé";
else                          cout << "pas trouvé";
cout << endl;
```

copy()

`copy()` implémente un algorithme général pour copier (une partir d')un container dans un autre.

Son prototype général est :

```
OutputIterator copy(InputIterator debut, InputIterator fin,
                    OutputIterator resultat);
```

qui copie le contenu compris entre `debut` (inclus) et `fin` (exclus) vers `resultat` (inclus) et les positions suivantes (itérateurs).

La valeur de retour est `resultat + (fin - debut)`.

Attention ! Notez bien que cela *copie* des éléments, mais ne fait pas d'insertion : il **faut** absolument que `resultat` ait (c.-à-d. pointe sur) la place nécessaire !

Exemple :

```
copy(unensemble.begin(), unensemble.end(), untableau.begin());
```

Notez que l'on peut ainsi copier des données d'une SDA dans une autre SDA d'un autre type.

copy() (2)

`copy()` peut être très utile pour afficher le contenu d'un container sur un flot en utilisant un `ostream_iterator` (je ne donne qu'un exemple ici :)

```
copy(container.begin(), container.end(),
     ostream_iterator<int>(cout, ", "));
```

`container` contenant ici des `int`.
Son contenu sera affiché sur `cout`, séparé par des « `,  ` ».

Jean-Cédric Chappelier
& Jamila Sam

copy() – Exemple complet

```
#include <iostream>
#include <set>
#include <vector>
#include <iterator>
using namespace std;

int main() {
  set<double> unensemble, unautre;

  unensemble.insert(1.1);  unensemble.insert(2.2);
  unensemble.insert(3.3);

  // copy(unensemble.begin(), unensemble.end(), unautre.begin());
  // ne fonctionnerait pas ("assignment of read-only location")
  // car, ici, unautre n'a pas la taille suffisante.

  vector<double> untableau(unensemble.size()); // prévoit la place

  copy(unensemble.begin(), unensemble.end(), untableau.begin());

  // output
  cout << "untableau = ";
  copy(untableau.begin(), untableau.end(),
       ostream_iterator<double>(cout, ", "));
  cout << endl;
  return 0;
}
```

sort()

`sort()` permet de trier des SDA implémentées sous forme de containers

La version la plus simple de tri est (il y en a d'autres) :

```
void sort(iterator debut, iterator fin)
```

qui utilise `operator<` des éléments contenus dans la partie du container indiquée par `debut` et `fin`
(les objets qui y sont stockés doivent donc posséder cet opérateur)

Exemple :

```
list<double> uneliste;
...
sort(uneliste.begin(), uneliste.end());
```

Exemple plus avancé (qui affiche « 1 2 4 5 7 8 ») :

```
constexpr size_t N(6);
int montableau[N] = { 1, 4, 2, 8, 5, 7 };
sort(montableau, montableau + N);
copy(montableau, montableau + N, ostream_iterator<int>(cout, " "));
cout << endl;
```


Jean-Cédric Chappelier
& Jamila Sam

Nombres Complexes

La bibliothèque `<complex>` définit les nombres complexes.

Ils se déclarent par `complex<double>`. Ils possèdent un constructeur à 2 paramètres permettant de préciser les parties réelle et imaginaire, p.ex.

```
complex<double> c(3.2,1.4), i(0,1);
```

Par contre, il n'existe pas de constructeur permettant de créer un nombre complexe à partir de ses coordonnées polaires.

En revanche, la fonction `polar`, qui prend comme paramètres la norme et l'argument du complexe à construire, permet de le faire. Cette fonction renvoie le nombre complexe nouvellement construit :

```
c = polar(sqrt(3.0), M_PI / 12.0);
```

Les méthodes des nombres complexes sont `real()` qui retourne la partie réelle, `imag()` qui retourne la partie imaginaire, et bien sûr les operateurs usuels.

Nombres Complexes (2)

Ce qui est plus inattendu c'est que les opérations de norme, argument, et conjugaison n'ont pas été implémentées sous forme de méthodes, mais de **fonctions** :

`double abs(const complex<double>&)`	retourne la **norme** (au sens français du terme) du nombre complexe
`double norm(const complex<double>&)`	retourne le **carré** de la norme
`double arg(const complex<double>&)`	retourne l'argument du nombre complexe
`complex<double> conj(const complex<double>&)`	retourne le complexe conjugué

La bibliothèque fournit de plus les extensions des fonctions de base (trigonométriques, logarithmes, exponentielle) aux nombres complexes.

Détails de `<cmath>`

Quelques fonctions définies dans la bibliothèque `<cmath>` :

`abs`	valeur absolue
`acos`	arccos
`asin`	arcsin
`atan`	arctan
`ceil`	$\lceil x \rceil$, entier supérieur
`cos`	cos
`cosh`	cosinus hyperbolique
`exp`	exp
`floor`	$\lfloor x \rfloor$, entier inférieur
`log`	ln, logarithme népérien
`log10`	log, logarithme en base 10
`pow(x,y)`	$x^y = \exp(y \ln x)$ — (préférez la multiplication pour les faibles puissances entières)
`sin`	sin
`sinh`	sinus hyperbolique
`sqrt`	$\sqrt{}$
`tan`	tan
`tanh`	tangente hyperbolique

Détails de <cmath> (2)

Quelques constantes :

	non ISO, mais souvent disponible avec `#define _USE_MATH_DEFINES` (*avant* les `#include`)	depuis C++20, avec : `#include <numbers>` `using namespace std::numbers;`
π	`M_PI`	`pi`
e	`M_E`	`e`
$\sqrt{2}$	`M_SQRT2`	`sqrt2`
$\ln(2)$	`M_LN2`	`ln2`
$\ln(10)$	`M_LN10`	`ln10`
$\log_2(e)$	`M_LOG2E`	`log2e`
$\frac{1}{\pi}$	`M_1_PI`	`inv_pi`
$\sqrt{3}$	—	`sqrt3`

... (et encore plein d'autres)

Nombres aléatoires

La génération de nombres au hasard sur ordinateur se fait avec des *générateurs* dit « pseudo-aléatoires » qui pour une valeur initiale donnée (appelée « **graine** » [« *seed* » en anglais]) donnent toujours la même séquence « aléatoire » (suivant une *distribution* de probabilités choisie).

Utiliser la même graine peut être utile pour déverminer un programme utilisant des nombres aléatoires.

Pour avoir une série de nombres aléatoires différente à chaque utilisation du programme, il faut utiliser une graine différente à chaque fois.

[Même si ce n'est pas terrible,] Cela se fait souvent en utilisant comme graine la valeur de l'horloge de l'ordinateur à cet instant.

Une autre solution consiste à tirer la graine (voire la séquence elle-même) depuis un **périphérique** matériel suffisement **aléatoire** (« *random device* ») : (micro-)déplacement de la souris, température du processeur, ...

Nombres aléatoires (2)

Dans la bibliothèque `<random>` (C++11), il existe différents **générateurs** de nombres pseudo-aléatoires et différentes **distributions** de probabilités.

Les deux doivent être combinés pour pouvoir effectuer une série de tirage.

Ci-après un exemple simple pour tirer de façon uniforme un nombre aléatoire entier entre `min` et `max`.

Jean-Cédric Chappelier
& Jamila Sam

Nombres aléatoires : exemple

```cpp
#include <iostream>
#include <functional> // pour bind()
#include <random>
using namespace std;

int main()
{
  // par exemple (dé ?)
  constexpr int min(1), max(6);

  // distribution uniforme entre min et max
  uniform_int_distribution<int> distribution(min, max);

  random_device rd;          // utilisé ici pour la graine
  unsigned int graine(rd()); // par exemple, ou sinon de votre choix

  // choix du générateur et initialisation (graine)
  default_random_engine generateur(graine);

  auto tirage(bind(distribution, generateur)); // ésotérisme

  for (unsigned int i(1); i <= 10; ++i) { // 10 tirages
    cout << tirage() << endl;
  }
  return 0;
}
```


Jean-Cédric Chappelier & Jamila Sam

Ce que j'ai appris

- Les bases de la **formalisation des données** : les structures de données abstraites
- Les deux structures de données abstraites les plus utilisées en informatique (en plus des tableaux et des types élémentaires) :
  les **listes chaînées** et les **piles**
- Qu'il existe beaucoup d'outils prédéfinis dans la bibliothèque standard de C++

  Le but n'est évidemment pas de les connaître tous par cœur, mais de **savoir qu'ils existent** pour penser aller chercher dans la documentation les informations complémentaires.