Collection de set de paramètres: documentation

Thomas Hilke

October 31, 2017

Contents

1 Introduction

Ce document décrit le langage de définition de paramètres accepté par le parseur du package `parameter`. Un fichier de paramètre permet de spécifier un ensemble de clés-valeurs, ou les clés

sont des chaînes de caractères et les valeurs appartiennent à l'un des cinques types disponibles. On appellera "set de paramètres" un ensemble de clés-valeurs.

De plus, un fichier de paramètre permet de définir non pas un unique set de paramètres, mais une collection de set de paramètres, à travers de laquelle le code client pourra itérer.

La section 2 présente la syntaxe formelle acceptée par le parseur ainsi que la définition des symboles terminaux. Cette section constitue une référence concernant la syntaxe d'un fichier de paramètres. La section 3 démontre l'usage et la sémantique des différentes constructions syntaxiques à travers une série d'exemples. Finalement, la section 4 documente l'API `C++` qui permet de lire des fichiers et d'accéder aux valeurs des paramètres spécifiées de chaque élement de la collection résultante.

2 Syntaxe formelle

Dans cette partie, on définit de façon formelle la syntaxe acceptée par le lexer et le parser. Les informations présentées dans cette partie tienne lieu de réference, après le code lui-même en ce qui concerne la syntaxe et sémantique du langage.

2.1 Grammaire

La syntaxe acceptée par le parseur du fichier de paramètre est une grammaire LL(1) définie ci-dessous. Les symbols entre chevrons correspondent aux symbols non-terminaux, les symbols sans décorations correspondent aux symbols terminaux et les symbols entres guillemets simples correspondent à leur valeur litterale. Le symbol $\langle start \rangle$ est bien entendu règle de production initiale, et ϵ correspond à la production vide.

$\langle start \rangle$ ::= $\langle statment\text{-}list \rangle$

$\langle statment\text{-}list \rangle$ ::= $\langle statment \rangle$ $\langle statment\text{-}list \rangle$
| ϵ

$\langle statment \rangle$ ::= $\langle inclusion \rangle$
| $\langle parameter\text{-}definition \rangle$
| $\langle group\text{-}definition \rangle$

$\langle inclusion \rangle$ ::= 'include' literal-string

$\langle parameter\text{-}definition \rangle$ ::= key def-symbol $\langle literal\text{-}list \rangle$
| 'override' key def-symbol $\langle literal\text{-}list \rangle$

$\langle literal\text{-}list \rangle$ ::= $\langle literal \rangle$ ',' $\langle literal\text{-}list \rangle$
| ϵ

$\langle literal \rangle$ ::= key
| literal-string
| literal-boolean
| literal-integer

| literal-real
| enum-item

⟨*group-definition*⟩ ::= '[' ⟨*parameter-definition-list*⟩ ']'

⟨*parameter-definition-list*⟩ ::= ⟨*parameter-definition*⟩ ⟨*parameter-definition-list*⟩
| ϵ

On notera que cette syntaxe n'utilise pas de symbol de terminaison de ligne, et que les seuls caractères de ponctuation qui apparaissent sont la virgule ',' et les crochets '[' et ']'. Malgré tout cette syntaxe n'est pas ambigüe au sens du parser, et est intuitive à déchiffrer pour un humain.

2.2 Symbols terminaux

Les expressions régulières des symbols littéraux étant triviales, on décrit ici uniquement les symbols terminaux:

- `key` = `/[-_a-zA-Z0-9]+/`,
- `literal-boolean` = `/(true)|(false)|(yes)|(no)|(on)|(off)/`,
- `literal-string` = `/"([^"\\]|(\\")|(\\\\))*"/`,
- `literal-real` = `/[+-]?((\.\d+)|(\d+\.)|(\d+\.\d+)|(\d+))([eE][+-]?\d+)?/`,
- `literal-integer` = `/[+-]?\d+/`,
- `enum-item` = `/#[-_a-zA-Z0-9]+/`,
- `def-symbol` = `/=|:|(->)/`.

Finalement, chaque token doit être séparé par une séquence de caractères qui correspond à l'expression régulière `/(\s|(;[^\n]*\n))*/`, ce qui implique que les terminaisons de lignes précédées par le caractère point-virgule ';' sont ignorées et permettent l'insertion de commentaires.

3 Exemples et sémantique

3.1 Commentaires

Comme indiqué ci-dessus, des commentaires peuvent être insérés dans une liste de paramètres. Un commentaire commence par le caractère ';' et se termine par le retour à la ligne. Il n'y a pas de notion de commentaire multiligne: chaque nouvelle ligne de commentaire doit être précédée par un ';'. Par exemple:

```
; Ceci est un commentaire,
; ceci est une deuxieme ligne de commentaire.

var = "str"   ; Definition d'un parametre
```

On utilisera les commentaires pour décrire la sémantique d'un paramètre, ou pour donner des précisions sur la valeur spécifiée, comme les unités physiques, par exemples.

On évitera d'utiliser des commentaires pour "désactiver" une ou des définitions, en revanche. Plusieurs construction dans la grammaire sont prévues pour répondre à ce besoin, l'une d'elles étant le sujet de la partie ci-dessous.

3.2 Inclusion

Souvent, plusieurs applications partagent des mêmes groupes de paramètres. Par exemple, on peut imaginer un groupe de paramètre qui encode les propriétés thermique de certains matériaux, la description d'une géometrie, ou encore des paramètres spécifiques à une certaine méthode numérique.

Afin de minimiser la duplication du code, la syntaxe permet l'inclusion du fichiers de paramètres, par l'intermédiaire du mot-clé `include`. Supposons que le fichier `water.conf` existe et contienne:

```
density = 1.0                  ; [g/cm^3]
specific-heat = 4.182          ; [j/g/K]
dynamical-viscosity = 2.0e-3   ; [Pa s]
```

On peut ensuite réutiliser ces définitions dans un nouveau fichier de paramètre `driven-cavity.conf`:

```
include "water.conf"

box-size = 1.0 ; [m]
boundary-velocity = 0.15 ; [m/s]
```

Le chemin d'un fichier est toujours relatif au fichier qui l'inclu. Ici, il faut que le fichier `water.conf` doit se trouver dans le même dossier que le `driven-cavity.conf`. Le chemin d'un fichier inclu ne peut pas être absolu.

3.3 Definition de paramètre

Un paramètre est défini par l'association entre un identifiant (`key` tel que défini dans la grammaire) et une valeur. Une valeur appartient à l'un des cinq types prédefinis: `string`, `integer`,`real`, `boolean` et `enumeration`. Dans l'exemple suivant, on définit cinq paramètres, un de chaque type, avec un commentaire associé:

```
string-parameter = "Hello, world"  ; The classical example

int-parameter = -1234              ; The opposite of the
                                   ;  natural number 1234

real-parameter = -3.1415e+00       ; The negative pi number

boolean-parameter = yes            ; A true value. Equivalent
                                   ;  to 'true' and 'on'.

enum-parameter = #neumann          ; enum-parameter has value
                                   ;  'neumann', which the user
```

```
                                    ;  will have to validate

str = string-parameter              ; str has the same value as
                                    ;  string-parameter definied
                                    ;  above
```

Vous pouvez vous référer à la partie 2.2 pour trouver une référence de la syntaxe de chaque partie de l'exemple.

On remarque dans l'exemple ci-dessus qu'on peut définir un paramètre à partir d'un autre, `str` dans ce cas se verra associer la valeur de `string-parameter`, soit la chaîne `"Hello, world"`.

Remarque Le type du paramètre `int-parameter` est `integer`, tandis que le type du paramètre `real-parameter` est `real`. Les nombres entiers et les nombres flottant ne sont pas équivalent et ne sont pas interchangeable. Si l'API `C++` demande un `integer`, on ne peu pas spécifier un `real`, et reciproquement.

Remarque On choisira de préférence des noms de paramètres descriptifs, en caractères minuscule, et les mots séparés par des trait-d'unions. Le nom d'un paramètre constitue une partie, sinon toute la documentation qui le concerne, il importe donc de choisir soigneusement le noms des paramètres. En particulier, on évitera les abbréviations, les identifiants de moins de 5 caractères, etc.

3.4 Collection de paramètres

Il arrive souvent en analyse numérique de devoir lancer une série de calculs pour lequels un unique paramètre varie. Un exemple classique est l'étude de convergence d'un schèma numérique. Lors d'une étude de convergence, on cherche à exécuter l'algorithme pour un nombre prédéfini de la tailles de subdivisions en espace, par exemple.

La syntaxe et sémantique de définition de paramètre prévoit ce cas de figure, et permet de définir une collection de sets de paramètres.

Soit `n` le nombre de subdivisions. On peut définir le paramètre `n` de la maniére suivante, en spécifiant une liste de valeurs:

```
n = 16, 32, 64
```

ce qui correspond à la collection de sets de paramètres suivante:

$$\{\mathtt{n} = 16\}, \{\mathtt{n} = 32\}, \{\mathtt{n} = 64\}. \tag{1}$$

Le code se chargera alors d'executer l'algorithme pour chacun des sets de paramètres.

Si des listes de valeurs sont spécifiées pour plusieurs paramètres simultanément, la collection de sets de paramètres résultant correspond au produit cartésien de chaque liste. Par exemple, si les paramètres `n` et `m` sont définis par:

```
m = 100, 200, 400
n = 16, 32, 64
```

la collection de set de paramètres résultant est la suivante:

$$\{\mathtt{m}=100,\mathtt{n}=16\},\{\mathtt{m}=100,\mathtt{n}=32\},\{\mathtt{m}=100,\mathtt{n}=64\},$$
$$\{\mathtt{m}=200,\mathtt{n}=16\},\{\mathtt{m}=200,\mathtt{n}=32\},\{\mathtt{m}=200,\mathtt{n}=64\},$$
$$\{\mathtt{m}=400,\mathtt{n}=16\},\{\mathtt{m}=400,\mathtt{n}=32\},\{\mathtt{m}=400,\mathtt{n}=64\}.$$

L'ordre dans lequel le produit cartésien est effectué n'est pas spécifié, et donc l'ordre dans lequel sera effectué chaque calcul est imprévisible. Par contre, l'ordre est déterministe, et sera le même d'une execution à l'autre.

Bien entendu, il y a des situations où ce n'est pas le comportement souhaité, et ou l'on aimerait qque chaque liste de paramètres soit groupée séquentiellement. La section suivante traite de ce cas de figure.

3.5 Groupe de paramètres

Il y a des situation où le produit cartésien n'est pas la bonne approche pour décrire une collection de sets de paramètres. Pour reprendre l'exemple de l'etude de convergence d'un schéma spatio-temporel, on veut spécifier une subdivision spatiale liée à la subdivision temporelle. Si note `space-sub` et `time-sub` le nombre de subdivisions spatiale et temporelle, on veut par exemple construire le set de parametres:

$$\{\texttt{space-sub}=100,\texttt{time-sub}=16\},$$
$$\{\texttt{space-sub}=200,\texttt{time-sub}=32\},$$
$$\{\texttt{space-sub}=400,\texttt{time-sub}=64\}.$$

Pour ce faire, on utilise un groupe de paramètres, groupé par des crochets '`[`' et '`]`':

```
[
  space-sub = 100, 200, 400
  time-sub  = 16, 32, 64
]
```

Il faut bien entendu que toutes les listes dans un groupe comportent le même nombre d'éléments. Un groupe peut comportent autant de paramètres que nécessaire, et on peut bien entendu définir des paramètres supplémentaires avant et après l'occurence du groupe. Finalement, plusieurs groupe peuvent êtres définis dans un même fichier de paramètres. L'exemple suivant démontre une situation complète:

```
a = 1
[
  b = 2, 3
  c = 4, 5
]
[
  d = 6, 7
  e = 8, 9
]
```

génère la collection de sets de paramètres suivant:

$$\{\mathtt{a}=1,\mathtt{b}=2,\mathtt{c}=4,\mathtt{d}=6,\mathtt{e}=8\},$$
$$\{\mathtt{a}=1,\mathtt{b}=3,\mathtt{c}=5,\mathtt{d}=6,\mathtt{e}=8\},$$
$$\{\mathtt{a}=1,\mathtt{b}=2,\mathtt{c}=4,\mathtt{d}=7,\mathtt{e}=9\},$$
$$\{\mathtt{a}=1,\mathtt{b}=3,\mathtt{c}=5,\mathtt{d}=7,\mathtt{e}=9\}.$$

On remarque que chaque groupe participe au produit cartésien de l'ensemble des paramètres, mais pas les listes constituant chacun des groupes.

Remarque La syntaxe n'empêche pas de déclarer une liste de valeur dont les types sont hétérogènes. Ceci dit, une telle définition n'as pas beaucoup de sens d'un point du vue de l'interface `C++`, puisque le type du paramètre concernè pourrait potentiellement changer dans chaque set de la collection engendrée.

3.6 Redéfinition de paramètres

On peut redéfinir un paramètre avec une nouvelle valeur, bien que, telle quelle, cette pratique soit découragée. En particulier, un warning est émis dans le cas suivant:

```
a = 1
a = 2
```

Ce comportement permet me mettre en évidence une une erreur qui est typiquement liée à une faute de frappe où a une faute d'inattention.

Si le comportement souhaité est effectivement de redéfinir un paramètre, il faut précéder la définition par le mot-clé `override`, comme dans l'exemple suivant:

```
a = 1
override a = 2
```

Si en revanche la variable qui est redéfinie avec le mot-clé `override` n'a pas été définie, une exception est levée, comme dans le cas suivant:

```
a = 1
override b = 2
```

De nouveau, ce comportement permet d'éviter une classe d'erreurs liées à la volonté effective de redéfinir des paramètres dans certaines situations.

Remarque La syntaxe n'empêche pas de changer le type d'un paramètre par l'intermédiaire d'une redéfinition. Ceci dit, une telle redéfinition n'as pas beaucoup de sens d'un point du vue de l'interface `C++`.

3.7 Interpolation de chaine de caractère

Les paramètres dont le type est une chaîne de caractère sont souvent utilisé pour représenter des noms de fichier, ou des fragments de nom de fichier, par exemple pour spécifier la destination dans le système de fichier du résultat d'un calcul.

La syntaxe prévoit un mécanisme d'interpolation de chaîne de caractère, qui permet de spécifier une chaîne de caractère fonction des autres paramètres définis dans le set.

La valeur d'un paramètre peut être inséré dans une chaîne de caractères en spécifiant le nom de celui-ci entre accolade. Dans l'exemple qui suit, la valeur du paramètre `n` est substitué dans la chaîne de caractère qui définit le paramètre `output`:

```
n = 100
output = "solution-with-subdivisions-{n}.dat"
```

Dans ce cas, le param¸etres `output` se verra assigne la chaîne de caractère `"solution-with-subdivisions-100.dat"`. Bien entendu, ceci fonctionne également avec les collections de paramètres. Le code suivant:

```
n = 100, 200
output = "soln-w-sub-{n}.dat"
```

est fonctionnellement équivalent à:

```
[
  n = 100, 200
  output = "soln-w-sub-100.dat", "soln-w-sub-200.dat"
]
```

tout en étant plus court à écrire, et moins susceptible à l'introduction d'erreur.

3.8 Exemple complet

Pour conclure cette documentation, on donne un exemple complet inspiré du code de simulation de formation de gel en périphérie des particules d'alumine. Pour éviter de surcharger la présentation, on a omis l'ensemble des commentaires, à l'exception des unités physiques.

Le fichier de configuration principal `\texttt{cryolite-particle-remelt.conf}` contient les définitions suivantes:

```
import "config/physical/cryolite-su.conf"
import "config/model/cryolite-particle-remelt-su.conf"
import "config/numerical/cryolite-particle-remelt.conf"

output-prefix = "cryolite-particle-remelt/output"
output-transient-solution = no
output-final-solution = yes
output-transition = yes
output-beta-function = no
output-neumann-exact-solution = no
```

Le fichier principal inclu trois fichiers de configurations qui regroupent respectivement les paramètres spécifiques à la physique des matériaux, c'est-à-dire les paramètres thermocinétiques de l'alumine et du bain dans ce cas-ci, les paramètres spécifiques au modèle, soit la spécification de la géométrie, les conditions de bords, les conditions initiales, etc, et finalement les paramètres spécifiques à la méthode num´erique. On a ègalement placé ici tous les paramètres qui contrôle l'output de la simulation, et qui sont susceptibles de changer frèquemment.

Le contenu des trois fichiers inclus est donné ci-dessous. Le contenu de `config/physical/cryolite--su.conf` est:

```
electrolyte-density = 2130.0e-9 ; [kg/mm^3]
alumina-density = 2130.0e-9 ; [kg/mm^3]

electrolyte-sl-low-t = 950.0 ; [C]
electrolyte-sl-high-t = 950.0 ; [C]
electrolyte-sl-latent-heat = 5.5083e5 ; [j/kg]

solid-electrolyte-heat-capacity = 1403. ; [j/kg/K]
liquid-electrolyte-heat-capacity = 1861.3 ; [j/kg/K]
alumina-heat-capacity = 1403. ; [j/kg/K]

solid-electrolyte-diffusivity-coefficient = 2.0e-3 ; [j/s/mm/K]
liquid-electrolyte-diffusivity-coefficient = 2.0e-3 ; [j/s/mm/K]
alumina-diffusivity-coefficient = 2.0e-3 ; [j/s/mm/K]
```

Le contenu de \texttt{config/model/cryolite-\-particle-\-remelt-\-su.conf} est:

```
time-end    = 0.130  ; [s]
domain-size = .5  ; [mm]

particle-radius = 0.05  ; [mm]

particle-initial-temperature    = 150.0 ; [C]
electrolyte-initial-temperature = 955.0 ; [C]

left-bc-type   = #neumann
left-bc-value  = 0.0 ; [K/mm]

right-bc-type  = #dirichlet
right-bc-value = 955.0 ; [C]

coordinates = #spherical
```

Le contenu de `config/numerical/cryolite-particle-remelt.conf` est:

```
space-subdivisions = 4000
time-subdivisions  = 4000
```

Le fichier `cryolite-particle-remelt.conf` et les trois fichier de paramètres inclus contitues la base de paramètres par défaut du modèle simulé par le code, dans ce cas la formation de gel autour d'une particule d'alumine dans un bain électrolytique en èlements finits $\mathbb{P}_1 - \mathbb{P}_1$ sous forme enthalpique avec la formule de Chernoff.

Toutes les études de ce modèle devraient être implémentées sous forme d'un fichier de paramètres qui inclu la base `cryolite-particle-remelt.conf`, puis redéfinit certain de ces paramètres.

Considérons le cas d'une étude de convergence, impémenté par le fichier de paramètres `cryolite--particle-remelt-conv-study.conf` donné ci-dessous:

```
; Convergence study, expected order in dt and h is approximatly 0.75

include "cryolite-particle-remelt.conf"

override-output-final-solution = yes
override output-transition = no
override output-beta-function = no
override output-neumann-exact-solution = yes

override time-end = 0.5  ; [s]
override domain-size = 1000.0 ; [mm]
[
  override space-subdivisions = 100, 200, 400, 800, 1600
  override time-subdivisions = 128, 256, 512, 1024, 2048
]

override output-prefix
  = "cryolite-particle-remelt/conv/output-h-{space-subdivisions}"
```

On notera l'usage du mot-clé `override` qui garantit que les paramètres que l'on redéfinit existent au préalable. On notera également l'usage d'un groupe de paramètres pour définir le set de raffinements successifs. Et finalement, on remarquera que l'output est redirigé vers un dossier spécifique pour chaque raffinement. Pour éviter que la ligne soit trop longue, elle est répartie sur deux lignes.

4 Interface C++

La librairie `parameter` offre une unique classe `parameter::collection` pour lire des fichiers de paramètres, gérer des collections de parametres et accéder à la valeur de chaque paramètre.

Dans cette section on présente l'interface de cette classe, accompagné par des exemples. L'interface public de la classe `parameter::collection` est la suivante:

```
namespace parameter {
  class collection {
   public:
    collection();
    ~collection();

    std::size_t get_collection_size() const;
    void set_current_collection(std::size_t i);

    void read_from_file(const std::string& filename);

    void set_key_value(const std::string& key, double value);
    void set_key_value(const std::string& key, bool value);
    void set_key_value(const std::string& key, int value);
    void set_key_value(const std::string& key, const std::string& value);

    template<typename enum_type>
```

```
    enum_type get_enum_value(
      const std::string& key,
      const std::map<std::string, enum_type>& token_map) const;

    template<typename value_type>
    value_type get_value(const std::string& key) const;

    const basic_value* get_basic_value(const std::string& key) const;

    void clear();

    void print_key_values(std::ostream& stream) const;
  };
}
```

4.1 Construction

L'unique constructeur est le constructeur par défaut qui ne prend pas de paramètre:

```
parameter::collection::collection();
```

4.1.1 Exemple

```
#include <parameter.hpp>

int main(int argc, char** argv) {
  // Declaration et initialisation d'une collection p:
  parameter::collection p;

  return 0;
}
```

4.2 Lecture d'un fichier de paramètre

```
void parameter::collection::read_from_file(const std::string& filename);
```

4.2.1 Exemple

Le code suivant:

```
#include <iostream>
#include <parameter.hpp>

int main(int argc, char** argv) {
  // Declaration et initialisation d'une collection p:
  parameter::collection p;
  p.read_from_file("file.conf");
  p.print_key_values(std::cout);

  return 0;
}
```

affiche le résultat suivant dans le terminal:

```
integer everest-height = 8848
real pi = 3.1415
```

pour autant que le fichier `file.conf` contienne:

```
pi = 3.1415
everest-height = 8848
```

4.3 Insertion manuelle de paramètre

Les quatres méthodes suivantes permette d'insérer de nouvelles valeurs dans la collection de parametre dont le type correspond au type du deuxième argument:

```
void parameter::collection::set_key_value(const std::string& key, double value);
void parameter::collection::set_key_value(const std::string& key, bool value);
void parameter::collection::set_key_value(const std::string& key, int value);
void parameter::collection::set_key_value(const std::string& key, const std::string& value);
```

On note qu'il n'y a pas de méthode pour insérer une valeur de type `enum`, et il n'est pas possible non plus de définir plusieurs valeurs pour un paramètres donné.

4.3.1 Exemple

Le code suivant:

```
#include <iostream>
#include <parameter.hpp>

int main(int argc, char** argv) {
  // Declaration et initialisation d'une collection p:
  parameter::collection p;

  p.set_key_value("pi", 3.1415);
  p.set_key_value("everest-height", 8848);

  p.print_key_values(std::cout);

  return 0;
}
```

affiche le résultat suivant dans le terminal:

```
integer everest-height = 8848
real pi = 3.1415
```

4.4 Accès au paramètres

Le deux méthodes templates suivantes permette d'accéder aux valeurs des paramètres définis par la collection:

```
template<typename enum_type>
enum_type parameter::collection::get_enum_value(
  const std::string& key,
  const std::map<std::string, enum_type>& token_map) const;

template<typename value_type>
value_type parameter::collection::get_value(const std::string& key) const;
```

La méthode `get_enum_value` prend en paramètre la clè, ainsi qu'une `std::map` qui associe un élement d'un type enuméré à une chaîne de caractère.

4.4.1 Exemple: paramètre de type `real`, `integer`, `string` et `boolean`

Le code qui suit:

```
#include <iostream>
#include <parameter.hpp>

int main(int argc, char** argv) {
  // Declaration et initialisation d'une collection p:
  parameter::collection p;
  p.read_from_file("file.conf");

  std::cout << "everest-value is " << p.get_value<int>("everest-value") << std::endl;
  std::cout << "pi is " << p.get_value<double>("pi") << std::endl;

  return 0;
}
```

affiche le résultat suivant dans le terminal:

```
everest-height is 8848
pi is 3.1415
```

pour autant que le fichier `file.conf` contienne:

```
pi = 3.1415
everest-height = 8848
```

4.4.2 Exemple: paramètre de type enuméré

Le code qui suit:

```
#include <iostream>
#include <parameter.hpp>

enum class bc_type { dirichlet, neumann };

int main(int argc, char** argv) {
  std::map<std::string, bc_type> bc_type_map;
  bc_type_map["dirichlet"] = bc_type::dirichlet;
  bc_type_map["neumann"] = bc_type::neumann;

  // Declaration et initialisation d'une collection p:
  parameter::collection p;
```

```
  p.read_from_file("file.conf");

  bc_type bc(p.get_enum_value<bc_type>("bc", bc_type_map));

  switch (bc) {
  case bc_type::dirichlet:
  std::cout << "bc is dirichlet" << std::endl;
  break;

  case bc_type::neumann:
  std::cout << "bc is neumann" << std::endl;
  break;
  }

  return 0;
}
```

affiche le résultat suivant dans le terminal:

```
bc is dirichlet
```

pour autant que le fichier `file.conf` contienne:

```
bc = #dirichlet
```

4.5 Itération à travers une collection

La paire de méthodes suivante permet d'accéder à la taille de la collection de paramètres et de sélectionner l'élément courant de la collection:

```
std::size_t parameter::collection::get_collection_size() const;
void parameter::collection::set_current_collection(std::size_t i);
```

4.5.1 Exemple

Le code qui suit:

```
#include <iostream>
#include <parameter.hpp>

int main(int argc, char** argv) {

  // Declaration et initialisation d'une collection p:
  parameter::collection p;
  p.read_from_file("file.conf");

  for (std::size_t i(0); i < p.get_collection_size(); ++i) {
    p.set_current_collection(i);
    std::cout << "collection #" << i + 1 << ":" << std::endl;

    std::cout << "  n = " << p.get_value<int>("n") << std::endl;
    std::cout << "  m = " << p.get_value<int>("m") << std::endl;
  }

  return 0;
}
```

affiche le résultat suivant dans le terminal:

```
collection #1:
  n = 32
  m = 3
collection #2:
  n = 64
  m = 3
collection #3:
  n = 128
  m = 3
collection #4:
  n = 32
  m = 4
collection #5:
  n = 64
  m = 4
collection #6:
  n = 128
  m = 4
```

pour autant que le fichier `file.conf` contienne:

```
n = 32, 64, 128
m = 3, 4
```