



CPPLI : TD 7 : C++ : Conteneurs et algorithmes standards

Nicolas Vansteenkiste Romain Absil Jonas Beleho *
(ESI – HE2B)

Année académique 2019 – 2020

Ce TD¹ aborde l'étude des [conteneurs standards](#)² et des [algorithmes standards](#)³ du C++⁴.

Ex. 7.1 Testez la fonction `nvs::random_value`⁵ pour des valeurs entières comprises entre 0 et 9, ces valeurs incluses. Son code est fourni dans le fichier `random.hpp`⁶ et est reproduit en appendice A.1. Voici comment réaliser ce test :

- (a) créez une `std::array`⁷ de 10 `unsigned`;
- (b) invoquez `nvs::random_value` pour produire une valeur entière entre 0 et 9;
- (c) incrémentez l'élément de la `std::array` dont l'index est égal au résultat pseudo-aléatoire;

*Et aussi, lors des années passées : Monica Bastreggi, Stéphan Monbaliu, Anne Rousseau et Moussa Wahid.

1. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td07_cpp/td07_cpp.pdf (consulté le 1^{er} novembre 2019).

2. <https://en.cppreference.com/w/cpp/container> (consulté le 1^{er} novembre 2019).

3. <https://en.cppreference.com/w/cpp/algorithm> (consulté le 1^{er} novembre 2019).

4. <https://fr.wikipedia.org/wiki/C%2B%2B> (consulté le 1^{er} novembre 2019).

5. Il s'agit en fait de la surcharge d'une fonction et d'un modèle de fonction.

6. https://poesi.esi-bru.be/pluginfile.php/4981/mod_folder/content/0/random/random.hpp (consulté le 1^{er} novembre 2019).

7. <https://en.cppreference.com/w/cpp/container/array> (consulté le 1^{er} novembre 2019).

- (d) répétez l'invocation et l'incrémentation correspondante 10, 100, 1 000, 10 000, 100 000, etc. fois ;
- (e) établissez enfin les statistiques d'occurrences de chacune des valeurs aléatoires, de 0 à 9, et affichez les résultats.

Cela donne un résultat console comme le suivant :

```
nombre de 0 : 400000015 (10.000000 %)
nombre de 1 : 399991154 (9.999779 %)
nombre de 2 : 400006901 (10.000173 %)
nombre de 3 : 399999965 (9.999999 %)
nombre de 4 : 399993698 (9.999842 %)
nombre de 5 : 400001452 (10.000036 %)
nombre de 6 : 399999259 (9.999981 %)
nombre de 7 : 400005056 (10.000126 %)
nombre de 8 : 400000401 (10.000010 %)
nombre de 9 : 400002099 (10.000052 %)
total       : 4000000000 (100.000000 %)
```

Ex. 7.2 Écrivez la fonction de prototype :

```
void print(const std::vector<int> & data);
```

Elle affiche sur la sortie standard les éléments du `std::vector`⁸ en argument, séparés par un espace. Un passage à la ligne termine l'affichage.

Ex. 7.3 Écrivez la fonction de prototype :

```
void sort(std::vector<int> & data, bool ascending);
```

Elle trie les éléments de `data` dans l'ordre croissant ou décroissant selon que `increasing` soit `true` ou `false`⁹. Implémentez l'algorithme de tri¹⁰ de votre choix.

Testez votre fonction de tri et vérifiez son bon fonctionnement avec la fonction d'affichage de l'Ex. 7.2.

```
avant :
-4 -3 4 5 2 5 2 -4 3 0 -5 1 5 -3 2 -5 -5 -3 2 0
après :
5 5 5 4 3 2 2 2 2 1 0 0 -3 -3 -3 -4 -4 -5 -5 -5
```

Pour peupler le `std::vector` de valeurs aléatoires entre -5 et 5 , par exemple, à l'aide du modèle de fonction `nvs::random_value` présenté à l'Ex. 7.1, l'algorithme `std::generate`¹¹ est idéal :

8. <https://en.cppreference.com/w/cpp/container/vector> (consulté le 1^{er} novembre 2019).

9. Un argument d'un type énuméré serait plus explicite que le booléen utilisé ici.

10. https://fr.wikipedia.org/wiki/Algorithme_de_tri (consulté le 1^{er} novembre 2019).

11. <https://en.cppreference.com/w/cpp/algorithm/generate> (consulté le 1^{er} novembre 2019).

```
1 vector<int> data(20);
2 generate(begin(data), end(data), []()
3 {
4     return nvs::random_value(-5, 5);
5 });
```

Ex. 7.4 À l'aide de la fonction¹² standard `std::sort`¹³, triez un `std::vector` d'`unsigned` dans l'ordre :

- (a) croissant ;
- (b) décroissant : n'hésitez pas à utiliser l'objet fonction¹⁴ `std::greater`¹⁵ ;
- (c) croissant modulo 3.

Voilà à quoi la sortie console doit ressembler :

```
avant :
1 9 3 11 2 9 4 5 5 1 1 11 11 6 4 5 6 8 9 0

croissant :
0 1 1 1 2 3 4 4 5 5 5 6 6 8 9 9 9 11 11 11

décroissant :
11 11 11 9 9 9 8 6 6 5 5 5 4 4 3 2 1 1 1 0

croissant modulo 3 :
0 9 9 9 6 6 3 1 1 1 4 4 5 2 5 11 5 8 11 11
```

Ex. 7.5 Écrivez la fonction de prototype :

```
unsigned primeFactor(std::map<unsigned, unsigned> & result,
                    unsigned value)
```

Elle réalise la **décomposition en produit de facteurs premiers**¹⁶ de `value`. La valeur renvoyée est le nombre de facteurs premiers différents de `value`. En particulier si `value` vaut 0 ou 1, elle retourne 0 et ne modifie pas la `std::map`¹⁷ `result`. Dans les autres cas,

12. Il s'agit en fait d'un modèle de fonction.

13. <https://en.cppreference.com/w/cpp/algorithm/sort> (consulté le 1^{er} novembre 2019).

14. N'ayez pas peur : il s'agit simplement d'une classe (un type) dont les instances (les objets) peuvent être utilisées comme des fonctions.

15. <https://en.cppreference.com/w/cpp/utility/functional/greater> (consulté le 1^{er} novembre 2019).

16. https://fr.wikipedia.org/wiki/D%C3%A9composition_en_produit_de_facteurs_premiers (consulté le 1^{er} novembre 2019).

17. <https://en.cppreference.com/w/cpp/container/map> (consulté le 1^{er} novembre 2019).

le contenu initial de `result` est remplacé par la décomposition. Les *clés* de la `std::map` sont les facteurs premiers tandis que les *valeurs* sont les puissances respectives de ces facteurs. Seuls les facteurs premiers effectifs apparaissent dans `result` : il n'y a pas de clé à valeur nulle.

Testez votre fonction de décomposition en produit de facteurs premiers et réalisez un affichage console semblable au suivant :

```
1552521051 = 3^2 * 8629 * 19991
1552521052 = 2^2 * 67 * 569 * 10181
1552521053 = 1552521053
1552521054 = 2 * 3 * 7 * 36964787
1552521055 = 5 * 409 * 759179
1552521056 = 2^5 * 17 * 239 * 11941
1552521057 = 3 * 1009 * 512891
1552521058 = 2 * 11 * 179 * 394241
1552521059 = 83 * 18705073
1552521060 = 2^2 * 3^3 * 5 * 103^2 * 271
```

A. Fichiers d'en-têtes

A.1. random.hpp

```
1  /*!
2   * \file random.hpp
3   * \brief Définitions de fonctions conviviales pour générer des
4   *        séquences pseudo-aléatoires.
5   */
6  #ifndef RANDOM_HPP
7  #define RANDOM_HPP
8
9  #include <random>
10 #include <utility>
11 #include <limits>
12 #ifdef _WIN32
13 #include <ctime>
14 #endif
15
16 /*!
17 * \brief Espace de nom de Nicolas Vansteenkiste.
18 *
19 */
20 namespace nvs
```

```
21 {
22
23 // fonctions
24
25 /*!
26 * \brief Un générateur de nombres uniformément aléatoires.
27 *
28 * Cette fonction produit et partage un unique
29 * générateur de nombres uniformément aléatoires
30 * (_Uniform Random Number Generator_).
31 * Elle est issue de Random Number Generation in C++11
32 * [WG21 N3551]
33 * (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf),
34 * par Walter E. Brown.
35 *
36 * _Remarque_ : Sous Windows, c'est un std::mt19937 qui est
37 * retourné, sous les autres systèmes d'exploitation c'est
38 * un std::default_random_engine. La raison en est qu'avec
39 * gcc sous Windows, la première valeur retournée par
40 * un std::default_random_engine change peu en fonction de la
41 * graine plantée avec nvs::randomize. Pour s'en convaincre,
42 * exécuter nvs::random_value(1, 100000) par des instances
43 * successives d'un même programme...
44 *
45 * \return un générateur de nombres uniformément aléatoires.
46 */
47 inline auto & urng()
48 {
49 #ifdef _WIN32
50     static std::mt19937 u {};
51     // https://stackoverflow.com/a/32731387
52     // dans le lien précédent : Linux <-> gcc
53     //                               et Windows <-> msvc
54 #else
55     static std::default_random_engine u {};
56 #endif
57     return u;
58 }
59
60 /*!
61 * \brief Un peu de bruit.
62 *
63 * Cette fonction met le générateur de nombres uniformément
64 * aléatoires partagé par nvs::urng() dans un état aléatoire.
```

```
65  * Elle est issue de Random Number Generation in C++11
66  * [WG21 N3551]
67  * (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf),
68  * par Walter E. Brown.
69  */
70 inline void randomize()
71 {
72     #ifdef _WIN32
73         urng().seed(std::time(nullptr));
74         // https://stackoverflow.com/a/18908041
75     #else
76         static std::random_device rd {};
77         urng().seed(rd());
78     #endif
79 }
80
81 /*!
82  * \brief Générateur de flottants aléatoires.
83  *
84  * Les flottants produits se distribuent uniformément entre
85  * `min` et `max`, la valeur minimale comprise, la maximale non.
86  *
87  * Si `max` est strictement inférieur à `min`, les contenus de ces
88  * variables sont permutés.
89  *
90  * Cette fonction est largement inspirée par Random Number
91  * Generation in C++11 [WG21 N3551]
92  * (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf),
93  * par Walter E. Brown.
94  *
95  * _Remarque_ : Par rapport au modèle de fonction nus::random_value<T>
96  * produisant des entiers aléatoires, les arguments `min` et `max`
97  * sont inversés de sorte à avoir la valeur nulle (0) comme borne
98  * (minimale ou maximale) si la fonction est appelée avec un seul
99  * argument. Notez que cela n'a pas de réelle incidence sur la
100  * signification des paramètres puisque leurs contenus sont permutés
101  * si nécessaire.
102  *
103  * \param max la borne supérieure (ou inférieure) de l'intervalle
104  *           dans lequel les flottants sont générés.
105  * \param min la borne inférieure (ou supérieure) de l'intervalle
106  *           dans lequel les flottants sont générés.
107  *
108  * \return un flottant dans l'intervalle semi-ouvert à droite
```

```
109 *          [`min`, `max`[ (ou [`max`, `min`[ si `max` < `min`).
110 */
111 inline double random_value(double max = 1., double min = 0.)
112 {
113     static std::uniform_real_distribution<double> d {};
114
115     if (max < min) std::swap(min, max);
116
117     return d(urng(),
118             decltype(d)::param_type {min, max});
119 }
120
121 // fonctions template
122
123 /*!
124 * \brief Générateur d'entiers aléatoires.
125 *
126 * Les entiers produits se distribuent uniformément entre
127 * `min` et `max`, ces valeurs incluses.
128 *
129 * The effect is undefined if T is not one of : short, int, long,
130 * long long, unsigned short, unsigned int, unsigned long, or
131 * unsigned long long.
132 *
133 * Si `max` est strictement inférieur à `min`, les contenus de ces
134 * variables sont permutés.
135 *
136 * Cette fonction est largement inspirée par Random Number Generation
137 * in C++11 [WG21 N3551]
138 * (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf),
139 * par Walter E. Brown.
140 *
141 * \param min la valeur minimale (ou maximale) pouvant être retournée.
142 * \param max la valeur maximale (ou minimale) pouvant être retournée.
143 *
144 * \return un entier entre `min` et `max`.
145 */
146 template<typename T = int>
147 inline T random_value(T min = std::numeric_limits<T>::min(),
148                     T max = std::numeric_limits<T>::max())
149 {
150     static std::uniform_int_distribution<T> d {};
151
152     if (max < min) std::swap(min, max);
```

```
153  
154     return d(urng(),  
155             typename decltype(d)::param_type {min, max});  
156 }  
157  
158 } // namespace nvs  
159  
160 #endif // RANDOM_HPP
```