CPPLI : TD 4 : C++ : Fonctions et conteneurs standards

Nicolas Vansteenkiste Romain Absil Jonas Beleho *
(Esi – He2b)

Année académique 2017 – 2018

Ce TD [1] aborde l'étude des fonctions [2], des conteneurs standards [3], mais aussi des algorithmes standards [4] du C++ [5].
Durée : 2 séances.

1. Fonctions

Ex. 4.1 Écrivez la fonction de prototype :

```
bool isPrime(unsigned number);
```

Elle retourne `true` [6] ou `false` selon que son argument est un nombre premier [7] ou non. Répartissez prototype et code dans les fichiers `mathesi.h` et `mathesi.cpp`.

Ex. 4.2 Arrangez-vous pour produire, à l'aide de la fonction `isPrime(unsigned)` de l'Ex. 4.1, la sortie console suivante :

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

1. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td04_cpp/td04_cpp.pdf
2. http://en.cppreference.com/w/cpp/language/functions
3. http://en.cppreference.com/w/cpp/container
4. http://en.cppreference.com/w/cpp/algorithm
5. https://en.wikipedia.org/wiki/C%2B%2B14
6. http://en.cppreference.com/w/cpp/language/bool_literal
7. https://en.wikipedia.org/wiki/Prime_number

```
Les nombres premiers entre 200 et 349 :
   .   .   .   .   .   .   .   .   .   .
   . 211   .   .   .   .   .   .   .   .
   .   .   . 223   .   .   . 227   . 229
   .   .   . 233   .   .   .   .   . 239
   . 241   .   .   .   .   .   .   .   .
   . 251   .   .   .   .   . 257   .   .
   .   .   . 263   .   .   .   .   . 269
   . 271   .   .   .   .   . 277   .   .
   . 281   . 283   .   .   .   .   .   .
   .   .   . 293   .   .   .   .   .   .
   .   .   .   .   .   .   . 307   .   .
   . 311   . 313   .   .   . 317   .   .
   .   .   .   .   .   .   .   .   .   .
   . 331   .   .   .   .   . 337   .   .
   .   .   .   .   .   .   . 347   . 349
```

Pour la mise en forme, utilisez les manipulateurs de flux [8] !

Ex. 4.3 Écrivez la fonction de prototype :

```
std::pair<int, int> euclidianDivision(int dividend, int divisor);
```

Elle calcule la division euclidienne [9] du dividende `dividend` par le diviseur `divisor`. Le champ `first` de la `std::pair` [10] retournée est la division entière de `dividend` par `divisor` ; son champ `second` correspond à leur modulo.

Répartissez prototype et code dans les mêmes fichiers `mathesi.h` et `mathesi.cpp` que ceux de l'Ex. 4.1.

Rem. : Que faire si `divisor` est nul ? Levez une exception du type `std::domain_error` [11] avec un message adéquat.

Ex. 4.4 Arrangez-vous pour produire, à l'aide de la fonction de l'Ex. 4.3, la sortie console suivante :

```
euclidianDivision : division by zero
27 = 27 *  1 +  0
27 = 13 *  2 +  1
27 =  9 *  3 +  0
27 =  6 *  4 +  3
27 =  5 *  5 +  2
27 =  4 *  6 +  3
```

8. http://en.cppreference.com/w/cpp/io/manip
9. https://fr.wikipedia.org/wiki/Division_euclidienne
10. http://en.cppreference.com/w/cpp/utility/pair
11. http://en.cppreference.com/w/cpp/error/domain_error

```
27 =  3 *  7 +  6
27 =  3 *  8 +  3
27 =  3 *  9 +  0
27 =  2 * 10 +  7
27 =  2 * 11 +  5
27 =  2 * 12 +  3
27 =  2 * 13 +  1
27 =  1 * 14 + 13
27 =  1 * 15 + 12
27 =  1 * 16 + 11
27 =  1 * 17 + 10
27 =  1 * 18 +  9
27 =  1 * 19 +  8
27 =  1 * 20 +  7
27 =  1 * 21 +  6
27 =  1 * 22 +  5
27 =  1 * 23 +  4
27 =  1 * 24 +  3
27 =  1 * 25 +  2
27 =  1 * 26 +  1
27 =  1 * 27 +  0
```

Ex. 4.5 Pour chaque ligne de la fonction `main()` du source surcharge_01.cpp[12], dites si elle compile ou non. Si non, dites pourquoi ; si oui, dites quel affichage est alors produit.

```cpp
/*!
 * \file surcharge_01.cpp
 * \brief surcharge de fonctions
 */
#include <iostream>

void f(int) {
    std::cout << "f(int)" << std::endl;
}

int main() {
    f(3);
    f(4.5);
    f(true);
    f(3LL);
```

12. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td04_cpp/surcharge_01.cpp

```
16      f('T');
17      f(.3F);
18      f(5U);
19      short s {44};
20      f(s);
21      f(2e-2L);
22  }
```

Répondez à cette question d'abord sur papier, puis confrontez vos réponses aux résultats obtenus avec `gcc`.

Rappelez-vous la forme en C++ des littéraux entiers [13], flottants [14], booléens [15] et caractères [16], ainsi que les règles de choix [17] de fonction en cas de surcharge de fonction (*function overload*).

Ex. 4.6 Pour chaque ligne de la fonction `main()` du source `surcharge_02.cpp` [18], dites si elle compile ou non. Si non, dites pourquoi ; si oui, dites quel affichage est alors produit.

```
1   /*!
2    * \file surcharge_02.cpp
3    * \brief surcharge de fonctions
4    */
5   #include <iostream>
6
7   void f(int) {
8       std::cout << "f(int)" << std::endl;
9   }
10
11  void f(long double) {
12      std::cout << "f(long double)" << std::endl;
13  }
14
15  int main() {
16      f(3);
17      f(4.5);
18      f(true);
19      f(3LL);
```

13. http://en.cppreference.com/w/cpp/language/integer_literal
14. http://en.cppreference.com/w/cpp/language/floating_literal
15. http://en.cppreference.com/w/cpp/language/bool_literal
16. http://en.cppreference.com/w/cpp/language/character_literal
17. http://www-01.ibm.com/support/knowledgecenter/SSGH3R_13.1.2/com.ibm.xlcpp131.aix.doc/language_ref/implicit_conversion_sequences.html
18. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td04_cpp/surcharge_02.cpp

```
    f('T');
    f(.3F);
    f(5U);
    short s {44};
    f(s);
    f(2e-2L);
}
```

Répondez à cette question d'abord sur papier, puis confrontez vos réponses aux résultats obtenus avec `gcc`.

Ex. 4.7 Pour chaque ligne de la fonction `main()` du source `surcharge_03.cpp`[19], dites si elle compile ou non. Si non, dites pourquoi ; si oui, dites quel affichage est alors produit.

```
/*!
 * \file surcharge_03.cpp
 * \brief surcharge de fonctions
 */
#include <iostream>

void f(short) {
    std::cout << "f(short)" << std::endl;
}

void f(double) {
    std::cout << "f(double)" << std::endl;
}

int main() {
    f(3);
    f(4.5);
    f(true);
    f(3LL);
    f('T');
    f(.3F);
    f(5U);
    short s {44};
    f(s);
    f(2e-2L);
}
```

19. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td04_cpp/surcharge_03.cpp

Répondez à cette question d'abord sur papier, puis confrontez vos réponses aux résultats obtenus avec `gcc`.

Ex. 4.8 Pour chaque ligne de la fonction `main()` du source surcharge_04.cpp [20], dites si elle compile ou non. Si non, dites pourquoi ; si oui, dites quel affichage est alors produit.

```cpp
/*!
 * \file surcharge_04.cpp
 * \brief surcharge de fonctions
 */
#include <iostream>

void f(int) {
    std::cout << "f(int)" << std::endl;
}

void f(unsigned) {
    std::cout << "f(unsigned)" << std::endl;
}

void f(long long) {
    std::cout << "f(long long)" << std::endl;
}

void f(double) {
    std::cout << "f(double)" << std::endl;
}

void f(long double) {
    std::cout << "f(long double)" << std::endl;
}

int main() {
    f(3);
    f(4.5);
    f(true);
    f(3LL);
    f('T');
    f(.3F);
    f(5U);
    short s {44};
```

20. https://poesi.esi-bru.be/pluginfile.php/1320/mod_folder/content/0/td04_cpp/surcharge_04.cpp

```
36      f(s);
37      f(2e-2L);
38  }
```

Répondez à cette question d'abord sur papier, puis confrontez vos réponses aux résultats obtenus avec `gcc`.

2. Conteneurs standards

Ex. 4.9 Testez la fonction `nvs::random_value`[21] pour des valeurs entières comprises entre 0 et 9, ces valeurs incluses. Son code est fourni dans le fichier `random.hpp`[22] et est reproduit en appendice A.1. Voici comment réaliser ce test :

(a) créez une `std::array`[23] 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 ;

(d) répétez l'invocation et l'incrémentation subséquente 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. 4.10 Écrivez la fonction de prototype :

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

21. Il s'agit en fait de la surcharge d'une fonction et d'un modèle de fonction.
22. https://poesi.esi-bru.be/pluginfile.php/4981/mod_folder/content/0/random/random.hpp
23. http://en.cppreference.com/w/cpp/container/array

Elle affiche sur la sortie standard les éléments du `std::vector`[24] en argument, séparés par un espace. Un passage à la ligne termine l'affichage.

Ex. 4.11 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`[25]. Implémentez l'algorithme de tri[26] de votre choix.

Testez votre fonction de tri et vérifiez son bon fonctionnement avec la fonction d'affichage de l'Ex. 4.10.

```
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. 4.9, l'algorithme `std::generate`[27] est idéal :

```
1 vector<int> data(20);
2 auto f = []()
3 {
4     return nvs::random_value(-5, 5);
5 };
6 // ce qui précède est nécessaire car la fonction utilisée
7 // par generate ne peut pas avoir d'argument
8 // cf. http://en.cppreference.com/w/cpp/algorithm/generate
9 generate(begin(data), end(data), f);
```

Ex. 4.12 À l'aide de la fonction[28] standard `std::sort`[29], triez un `std::vector` d'`unsigned` dans l'ordre :

(a) croissant ;

(b) décroissant : n'hésitez pas à utiliser l'objet fonction[30] `std::greater`[31] ;

(c) croissant modulo 3.

Voilà à quoi la sortie console doit ressembler :

24. http://en.cppreference.com/w/cpp/container/vector
25. Un argument d'un type énuméré serait plus explicite que le booléen utilisé ici.
26. https://fr.wikipedia.org/wiki/Algorithme_de_tri
27. http://en.cppreference.com/w/cpp/algorithm/generate
28. Il s'agit en fait d'un modèle de fonction.
29. http://en.cppreference.com/w/cpp/algorithm/sort
30. N'ayez pas peur : il s'agit simplement d'une classe (un type) dont les instances (les objets) peuvent être utilisées comme des fonctions.
31. http://en.cppreference.com/w/cpp/utility/functional/greater

```
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. 4.13 Écrivez la fonction de prototype :

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

Elle réalise la décomposition en produit de facteurs premiers[32] 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`[33] `result`. Dans les autres cas, 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 la puissance 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
```

32. https://fr.wikipedia.org/wiki/D%C3%A9composition_en_produit_de_facteurs_premiers
33. http://en.cppreference.com/w/cpp/container/map

A. Fichiers d'en-têtes

A.1. `random.hpp`

```cpp
/*!
 * \file random.hpp
 * \brief Définitions de fonctions conviviales pour générer des
 *        séquences pseudo-aléatoires.
 */
#ifndef RANDOM_HPP
#define RANDOM_HPP

#include <random>
#include <utility>
#include <limits>
#ifdef _WIN32
#include <ctime>
#endif

/*!
 * \brief Espace de nom de Nicolas Vansteenkiste.
 *
 */
namespace nvs
{

// fonctions

/*!
 * \brief Un générateur de nombres uniformément aléatoires.
 *
 * Cette fonction produit et partage un unique
 * générateur de nombres uniformément aléatoires
 * (_Uniform Random Number Generator_).
 * Elle est issue de Random Number Generation in C++11
 * ([WG21 N3551]
 * (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf)),
 * par Walter E. Brown.
 *
 * _Remarque_ : Sous Windows, c'est un std::mt19937 qui est
 * retourné, sous les autres systèmes d'exploitation c'est
 * un std::default_random_engine. La raison en est qu'avec
 * gcc sous Windows, la première valeur retournée par
 * 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 successives
43  * 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 aléatoires
64  * 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
```

```
 * ‘min‘ et ‘max‘, la valeur minimale comprise, la maximale non.
 *
 * Si ‘max‘ est strictement inférieur à ‘min‘, les contenus de ces
 * variables sont permutés.
 *
 * Cette fonction est largement inspirée par Random Number Generation in
 * C++11 ([WG21 N3551]
 * (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf)),
 * par Walter E. Brown.
 *
 * _Remarque_ : Par rapport au modèle de fonction nvs::random_value<T>
 * produisant des entiers aléatoires, les arguments ‘min‘ et ‘max‘
 * sont inversés de sorte à avoir la valeur nulle (0) comme borne
 * (minimale ou maximale) si la fonction est appelée avec un seul
 * argument. Notez que cela n’a pas de réelle incidence sur la
 * signification des paramètres puisque leurs contenus sont permutés
 * si nécessaire.
 *
 * \param max la borne supérieure (ou inférieure) de l’intervalle
 *            dans lequel les flottants sont générés.
 * \param min la borne inférieure (ou supérieure) de l’intervalle
 *            dans lequel les flottants sont générés.
 *
 * \return un flottant dans l’intervalle semi-ouvert à droite
 *         [‘min‘, ‘max‘[ (ou [‘max‘, ‘min‘[ si ‘max‘ < ‘min‘).
 */
inline double random_value(double max = 1., double min = 0.)
{
    static std::uniform_real_distribution<double> d {};

    if (max < min) std::swap(min, max);

    return d(urng(),
             decltype(d)::param_type {min, max});
}

// fonctions template

/*!
 * \brief Générateur d’entiers aléatoires.
 *
 * Les entiers produits se distribuent uniformément entre
 * ‘min‘ et ‘max‘, ces valeurs incluses.
 *
```

```
 * The effect is undefined if T is not one of : short, int, long,
 * long long, unsigned short, unsigned int, unsigned long, or
 * unsigned long long.
 *
 * Si `max` est strictement inférieur à `min`, les contenus de ces
 * variables sont permutés.
 *
 * Cette fonction est largement inspirée par Random Number Generation
 * in C++11 ([WG21 N3551]
 * (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3551.pdf)),
 * par Walter E. Brown.
 *
 * \param min la valeur minimale (ou maximale) pouvant être retournée.
 * \param max la valeur maximale (ou minimale) pouvant être retournée.
 *
 * \return un entier entre `min` et `max`.
 */
template<typename T = int>
inline T random_value(T min = std::numeric_limits<T>::min(),
                      T max = std::numeric_limits<T>::max())
{
    static std::uniform_int_distribution<T> d {};

    if (max < min) std::swap(min, max);

    return d(urng(),
             typename decltype(d)::param_type {min, max});
}

} // namespace nvs

#endif // RANDOM_HPP
```