



CPPLI : TD 5 : C : Liste circulaire simplement chaînée

Romain Absil Jonas Beleho Pierre Bettens
David Hauweele Pierre Hauweele
Nicolas Vansteenkiste * (ESI – HE2B)

Année académique 2021 – 2022

Ce TD¹ aborde l'implémentation en langage C du type abstrait de données appelé [liste circulaire simplement chaînée](#)².

Ex. 5.1 Soit :

```
struct SLNode
{
    struct SLNode * next;
    value_t      value;
};
```

Il s'agit d'un type structuré qui sert à représenter les éléments d'une [liste simplement chaînée](#)³. La signification précise de ses champs est décrite dans le fichier [slnode.h](#)⁴ reproduit en annexe [A.2](#).

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

1. https://poesi.esi-bru.be/pluginfile.php/7311/mod_folder/content/0/td05_c/td05_c_withAppendix.pdf (consulté le 13 septembre 2021).

2. https://en.wikipedia.org/wiki/Linked_list#Circular_linked_list (consulté le 13 septembre 2021).

3. https://fr.wikipedia.org/wiki/Liste_cha%C3%A9e#Liste_simplement_cha%C3%A9e (consulté le 13 septembre 2021).

4. https://poesi.esi-bru.be/pluginfile.php/7311/mod_folder/content/0/td05_c/ressource/slnode.h (consulté le 13 septembre 2021).

Le type `value_t` est un [alias de type](#)⁵ quelconque. Il est défini dans le fichier `value_t.h`⁶. On en trouve un exemple pour le type `double` en annexe [A.1](#).

Implémentez les fonctions de manipulation de `struct SLNode` suivantes, dont une [documentation](#)⁷ précise est fournie comme pour `doxygen`⁸ dans `slnode.h` (voir l'annexe [A.2](#)) :

```
struct SLNode * newSLN(value_t value);

void deleteSLN(struct SLNode * * adpSLN);

struct SLNode * nextSLN(const struct SLNode * pSLN);

void setNextSLN(struct SLNode * pSLN, struct SLNode * pNewNext);

value_t valueSLN(const struct SLNode * pSLN);

void setValueSLN(struct SLNode * pSLN, value_t newValue);
```

Ex. 5.2 Implémentez la fonction suivante facilitant l'utilisation de `struct SLNode` (voir [Ex. 5.1](#)), dont une documentation précise est fournie dans `slnode_utility.h`⁹ reproduit en annexe [A.3](#) :

```
struct SLNode * forwardSLN(struct SLNode * pSLN, size_t distance);
```

Ex. 5.3 Soit :

```
struct SLCircularList
{
    struct SLNode * entry;
};
```

Le type structuré `struct SLCircularList` sert à représenter une liste circulaire simplement chaînée. Le type de son champ `entry` est `struct SLNode *` tel que défini à l'[Ex. 5.1](#). La signification précise de ce champ est décrite dans le fichier `slcircularlist.h`¹⁰ reproduit en annexe [A.4](#).

Implémentez les fonctions de manipulation de `struct SLCircularList` suivantes, dont une documentation précise est fournie dans `slcircularlist.h` (voir l'annexe [A.4](#)) :

5. <https://en.wikipedia.org/wiki/Typedef> (consulté le 13 septembre 2021).
6. https://poesi.esi-bru.be/pluginfile.php/7311/mod_folder/content/0/td05_c/ressource/value_t.h (consulté le 13 septembre 2021).
7. https://poesi.esi-bru.be/pluginfile.php/7311/mod_folder/content/0/td05_c/ressource/html.tar.xz (consulté le 13 septembre 2021).
8. <http://www.doxygen.nl/> (consulté le 13 septembre 2021).
9. https://poesi.esi-bru.be/pluginfile.php/7311/mod_folder/content/0/td05_c/ressource/slnode_utility.h (consulté le 13 septembre 2021).
10. https://poesi.esi-bru.be/pluginfile.php/7311/mod_folder/content/0/td05_c/ressource/slcircularlist.h (consulté le 13 septembre 2021).

```
struct SLCircularList * newSLCL(void);

void deleteSLCL(struct SLCircularList * * adpSLCL);

void clearSLCL(struct SLCircularList * pSLCL);

struct SLNode * entrySLCL(const struct SLCircularList * pSLCL);

bool emptySLCL(const struct SLCircularList * pSLCL);

struct SLNode * pushSLCL(struct SLCircularList * pSLCL,
                        value_t value);

struct SLNode * insertSLCL(struct SLCircularList * pSLCL,
                          struct SLNode * pSLN,
                          value_t value);

struct SLNode * popSLCL(struct SLCircularList * pSLCL);

struct SLNode * eraseSLCL(struct SLCircularList * pSLCL,
                        struct SLNode * pSLN);
```

Ex. 5.4 Implémentez les fonctions d'utilisation de `struct SLCircularList` (voir Ex. 5.3) suivantes, dont une documentation précise est fournie dans le fichier d'en-têtes `slcircularlist_utility.h`¹¹ reproduit en annexe A.5 :

```
size_t sizeSLCL(const struct SLCircularList * pSLCL);

struct SLNode * previousSLCL(const struct SLCircularList * pSLCL,
                          const struct SLNode * pSLN);
```

A. Fichiers d'en-têtes

A.1. value_t.h

```
1  /*!
2   * \file value_t.h
3   *
4   * \brief Définition de l'alias du type contenu par un élément
5   *       de liste bi-chaînée.
```

11. https://poesi.esi-bru.be/pluginfile.php/7311/mod_folder/content/0/td05_c/ressource/slcircularlist_utility.h (consulté le 13 septembre 2021).

```
6  */
7  #ifndef VALUE_T_H
8  #define VALUE_T_H
9
10 /*!
11  * \brief Type de la valeur contenue par un élément de liste
12  *        bidirectionnelle struct DLNode.
13  */
14 typedef double value_t;
15
16 #endif // VALUE_T_H
```

A.2. slnode.h

```
1  /*!
2  * \file slnode.h
3  *
4  * \brief Définition d'un type représentant un élément de liste
5  *        simplement chaînée.
6  */
7  #ifndef SLNODE_H
8  #define SLNODE_H
9
10 #include "value_t.h"
11
12 /*!
13  * \brief Valeurs d'erreurs associées à un élément de liste.
14  */
15 enum SLNError
16 {
17     /*!
18     * \brief Erreur lors d'une allocation mémoire d'un élément de
19     *        liste.
20     */
21     ESLNMEMORYFAIL = 50
22 };
23
24 /*!
25  * \brief Structure représentant le type d'un élément d'une
26  *        liste simplement chaînée ([linked list]
27  *        (https://en.wikipedia.org/wiki/Linked\_data\_structure#Linked\_lists)).
28  */
29 struct SLNode
```

```
30 {
31     /*!
32     * \brief Adresse de l'élément suivant dans la liste.
33     *
34     * S'il n'y a pas d'élément suivant, `next` vaut `NULL`.
35     */
36     struct SLNode * next;
37
38     /*!
39     * \brief Valeur conservée par l'élément de la liste.
40     */
41     value_t      value;
42 };
43
44 /*!
45 * \brief Création d'une instance de struct SLNode.
46 *
47 * L'élément de liste créé est dans un état valide : il ne possède
48 * pas de suivant.
49 *
50 * Il est alloué dynamiquement et doit donc être détruit quand
51 * son usage n'est plus requis.
52 *
53 * Si l'allocation mémoire échoue :
54 *   + `errno` est mis à ::ESLNMEMORYFAIL ;
55 *   + `NULL` est retourné.
56 *
57 * \param value la valeur contenue dans le struct SLNode.
58 *
59 * \return l'adresse du struct SLNode créé.
60 */
61 struct SLNode * newSLN(value_t value);
62
63 /*!
64 * \brief Destruction d'une instance de struct SLNode.
65 *
66 * Le struct SLNode pointé par le pointeur de struct SLNode
67 * dont l'adresse est fournie est détruit. Ensuite, ce
68 * pointeur de struct SLNode est mis à `NULL`.
69 *
70 * Aucun maillage n'est modifié par la fonction !
71 *
72 * Si `adpSLN` est `NULL`, le comportement de la fonction est
73 * indéterminé.
```

```
74  *
75  * \param adpSLN adresse d'un pointeur de struct SLNode vers le
76  *      struct SLNode à détruire.
77  */
78  void deleteSLN(struct SLNode * * adpSLN);
79
80  /*!
81  * \brief Accès en lecture à l'élément suivant de la liste.
82  *
83  * Si `pSLN` est `NULL`, le comportement de la fonction est
84  * indéterminé.
85  *
86  * \param pSLN adresse du struct SLNode dont on désire connaître le
87  *      suivant.
88  *
89  * \return Adresse du struct SLNode suivant celui pointé par `pSLN`.
90  */
91  struct SLNode * nextSLN(const struct SLNode * pSLN);
92
93  /*!
94  * \brief Accès en écriture à l'élément suivant de la liste.
95  *
96  * Seul le maillage du struct SLNode pointé par `pSLN` est modifié
97  * par cette fonction. La
98  * mémoire n'est pas gérée ici.
99  *
100  * Si `pSLN` est `NULL`, le comportement de la fonction est
101  * indéterminé.
102  *
103  * \param pSLN adresse du struct SLNode dont on désire modifier le
104  *      suivant.
105  * \param pNewNext adresse du nouveau struct SLNode suivant celui
106  *      pointé par `pSLN`.
107  */
108  void setNextSLN(struct SLNode * pSLN, struct SLNode * pNewNext);
109
110  /*!
111  * \brief Accès en lecture à la valeur stockée dans l'élément de liste.
112  *
113  * Si `pSLN` est `NULL`, le comportement de la fonction est
114  * indéterminé.
115  *
116  * \param pSLN adresse du struct DLNode dont on désire connaître la
117  *      valeur qu'il contient.
```

```
118  *
119  * \return valeur contenue dans le struct DLNode pointé par `pSLN`.
120  */
121  value_t valueSLN(const struct SLNode * pSLN);
122
123  /*!
124  * \brief Accès en écriture à la valeur contenue dans l'élément de
125  *       liste.
126  *
127  * Si `pSLN` est `NULL`, le comportement de la fonction est
128  * indéterminé.
129  *
130  * \param pSLN adresse du struct SLNode dont on désire modifier la
131  *          valeur.
132  * \param newValue nouvelle valeur à conserver dans le struct SLNode
133  *          pointé par `pSLN`.
134  */
135  void setValueSLN(struct SLNode * pSLN, value_t newValue);
136
137  #endif // SLNODE_H
```

A.3. slnode_utility.h

```
1  /*!
2  * \file slnode_utility.h
3  *
4  * \brief Définition de fonctions d'aide à l'utilisation de
5  *       struct SLNode.
6  */
7  #ifndef SLNODE_UTILITY_H
8  #define SLNODE_UTILITY_H
9
10 #include <stddef.h>
11
12 #include "slnode.h"
13
14 /*!
15 * \brief Accès à un élément suivant en position donnée.
16 *
17 * \param pSLN adresse du struct SLNode dont on désire accéder à
18 *          un suivant.
19 * \param distance position de l'élément désiré.
20 *
```

```
21  * \return adresse de l'élément `distance` positions après celui
22  *      d'adresse `pSLN` ou `NULL` s'il n'y en a pas.
23  */
24  struct SLNode * forwardSLN(struct SLNode * pSLN, size_t distance);
25
26  #endif // SLNODE_UTILITY_H
```

A.4. slcircularlist.h

```
1  /*!
2  * \file slcircularlist.h
3  *
4  * \brief Définition d'un type représentant une liste circulaire
5  *      simplement chaînée.
6  */
7  #ifndef SLCIRCULARLIST_H
8  #define SLCIRCULARLIST_H
9
10 #include <stdbool.h>
11 #include <stddef.h>
12
13 #include "slnode.h"
14
15 /*!
16 * \brief Valeurs d'erreurs associées à une liste.
17 */
18 enum SLError
19 {
20     /*!
21     * \brief Erreur lors d'une allocation mémoire d'une liste ou
22     *      d'un de ses éléments.
23     */
24     ESLLMEMORYFAIL = 60,
25
26     /*!
27     * \brief Opération interdite car la liste est vide.
28     */
29     ESLLEMPY
30 };
31
32 /*!
33 * \brief Structure représentant une liste circulaire simplement
34 *      chaînée ([circular linked list])
```



```
35  * (https://en.wikipedia.org/wiki/Linked\_list#Circular\_linked\_list)).
36  */
37  struct SLCircularList
38  {
39      /*!
40       * \brief Entrée dans la liste circulaire.
41       */
42      struct SLNode * entry;
43  };
44
45  /*!
46   * \brief Création d'une liste circulaire simplement chaînée.
47   *
48   * La liste est créée vide, c'est-à-dire que son champ `entry`
49   * est mis à `NULL`.
50   *
51   * Si l'allocation dynamique échoue :
52   *   + `errno` est mis à ::ESLLEMEMORYFAIL ;
53   *   + `NULL` est retourné.
54   *
55   * \return adresse de la struct SLCircularList créée.
56   */
57  struct SLCircularList * newSLCL(void);
58
59  /*!
60   * \brief Destruction d'une liste circulaire simplement chaînée.
61   *
62   * La struct SLCircularList pointée par le pointeur de
63   * struct SLCircularList
64   * dont l'adresse est fournie est détruite. Ensuite, ce
65   * pointeur de struct SLCircularList est mis à `NULL`.
66   *
67   * La destruction de la liste implique la destruction de tous
68   * ses éléments.
69   *
70   * Si `adpSLCL` est `NULL`, le comportement de la fonction est
71   * indéterminé.
72   *
73   * \param adpSLCL adresse d'un pointeur de struct SLCircularList
74   *                vers la struct SLCircularList à détruire.
75   */
76  void deleteSLCL(struct SLCircularList * * adpSLCL);
77
78  /*!
```

```
79  * \brief Destruction du contenu de la liste.
80  *
81  * Tous les struct SLNode qui constituent la liste sont détruits,
82  * mais pas la liste elle-même. En fin de fonction, la liste est
83  * vide, donc son champ `entry` est mis à `NULL`.
84  *
85  * Si `pSLCL` est `NULL`, le comportement de la fonction est
86  * indéterminé.
87  *
88  * \param pSLCL adresse de la struct SLCircularList dont on désire
89  *           détruite les éléments.
90  */
91 void clearSLCL(struct SLCircularList * pSLCL);
92
93 /*!
94  * \brief Accès en lecture de l'élément d'entrée de liste.
95  *
96  * Si la liste pointée par `pSLCL` est vide, `NULL` est retourné.
97  *
98  * Si `pSLCL` est `NULL`, le comportement de la fonction est
99  * indéterminé.
100  *
101  * \param pSLCL adresse de la struct SLCircularList dont on désire
102  *           connaître le struct SLNode d'entrée.
103  *
104  * \return adresse du struct SLNode en tête de la liste pointée
105  *         par `pSLCL`.
106  */
107 struct SLNode * entrySLCL(const struct SLCircularList * pSLCL);
108
109 /*!
110  * \brief Accès en lecture de la nature vide ou non de la liste.
111  *
112  * Si `pSLCL` est `NULL`, le comportement de la fonction est
113  * indéterminé.
114  *
115  * \param pSLCL adresse de la struct SLCircularList dont on désire
116  *           savoir si elle est vide ou non.
117  *
118  * \return `true` si la liste pointée par `pSLCL` ne contient aucun
119  *         struct SLNode, `false` sinon.
120  */
121 bool emptySLCL(const struct SLCircularList * pSLCL);
122
```

```
123  /*!
124  * \brief Insertion d'un élément en entrée de liste.
125  *
126  * Si l'instanciation du struct SLNode destiné à être la
127  * nouvelle tête de liste échoue :
128  *   + la liste est laissée telle quelle ;
129  *   + `errno` est mis à ::ESLLMEMORYFAIL.
130  *
131  * Si `pSLCL` est `NULL`, le comportement de la fonction est
132  * indéterminé.
133  *
134  * \param pSLCL adresse de la liste dont on veut modifier
135  *           l'élément d'entrée.
136  * \param value valeur que doit contenir l'élément en entrée
137  *           de liste.
138  *
139  * \return adresse de la nouvelle entrée de liste... ou
140  *         l'ancienne en cas d'échec.
141  */
142  struct SLNode * pushSLCL(struct SLCircularList * pSLCL,
143                          value_t value);
144
145  /*!
146  * \brief Insertion d'une nouvelle valeur dans la liste à
147  *           l'emplacement préalable d'un élément spécifié.
148  *
149  * Si l'instanciation du struct SLNode destiné à être inséré dans
150  * la liste échoue :
151  *   + la liste est laissée telle quelle ;
152  *   + `errno` est mis à ::ESLLMEMORYFAIL.
153  *
154  * Après insertion, l'élément pointé par `pSLN` devient le suivant
155  * de celui nouvellement inséré. En d'autres termes, l'élément
156  * fraîchement inséré prend la place de celui pointé par `pSLN`,
157  * tandis que ce dernier vient se placer à sa suite.
158  *
159  * En toute généralité :
160  *
161  *     insertSLCL(pSLCL, getEntrySLCL(pSLCL), value);
162  *
163  * est équivalent à :
164  *
165  *     pushSLCL(pSLCL, value);
166  *
```

```
167  * En particulier, si `pSLCL` pointe sur une liste vide :
168  *
169  *     insertSLCL(pSLCL, NULL, value);
170  *
171  * est équivalent à :
172  *
173  *     pushSLCL(pSLCL, value);
174  *
175  * Si `pSLCL` est `NULL`, si `pSLN` est `NULL` sans que `pSLCL`
176  * ne soit vide ou si `pSLN` ne pointe pas sur un élément de la
177  * liste pointée par `pSLCL`, le comportement de la fonction est
178  * indéterminé.
179  *
180  * Remarque : il est impossible avec cette fonction d'insérer un
181  * élément entre l'entrée et l'élément précédant l'entrée sans que
182  * le nouvel élément ne devienne la nouvelle entrée de liste.
183  *
184  * \param pSLCL adresse de la liste dans laquelle on désire insérer
185  *           un nouvel élément.
186  * \param pSLN adresse de l'élément de la liste où
187  *           l'insertion doit avoir lieu.
188  * \param value valeur du nouvel élément à insérer.
189  *
190  * \return adresse du nouvel élément inséré... ou `pSLN`
191  *         en cas d'échec.
192  */
193 struct SLNode * insertSLCL(struct SLCircularList * pSLCL,
194                           struct SLNode * pSLN,
195                           value_t value);
196
197 /*!
198  * \brief Suppression de l'élément en entrée de liste.
199  *
200  * L'élément supprimé est détruit.
201  *
202  * Si la liste pointée par `pSLCL` est initialement vide :
203  *   + la liste est laissée telle quelle ;
204  *   + `errno` est mis à ::ESLLEMPY ;
205  *   + `NULL` est retourné.
206  *
207  * Si `pSLCL` est `NULL`, le comportement de la fonction est
208  * indéterminé.
209  *
210  * \param pSLCL adresse de la liste dont on veut ôter
```

```
211      *          l'élément d'entrée.
212      *
213      * \return adresse de la nouvelle entrée de liste ou
214      *      `NULL` si elle est désormais vide.
215      */
216 struct SLNode * popSLCL(struct SLCircularList * pSLCL);
217
218 /*!
219  * \brief Suppression d'un élément de la liste.
220  *
221  * L'élément supprimé est détruit.
222  *
223  * Si la liste pointée par `pSLCL` est initialement vide, rien
224  * ne se passe.
225  *
226  * Si `pSLCL` est `NULL`, si `pSLN` est `NULL` alors que la liste
227  * n'est pas vide ou si `pSLN` ne pointe pas sur un élément de la
228  * liste pointée par `pSLCL`, le comportement de la fonction est
229  * indéterminé.
230  *
231  * \param pSLCL adresse de la liste dont on désire supprimer
232  *          un élément.
233  * \param pSLN adresse de l'élément à supprimer.
234  *
235  * \return adresse de l'élément de la liste qui se trouve, après
236  *         suppression, en même position que l'élément qui a été
237  *         supprimé, c'est-à-dire `NULL` si la liste est finalement
238  *         vide ou l'adresse de l'élément suivant l'élément
239  *         qui a été supprimé.
240  */
241 struct SLNode * eraseSLCL(struct SLCircularList * pSLCL,
242                          struct SLNode * pSLN);
243
244 #endif // SLCIRCULARLIST_H
```

A.5. slcircularlist_utility.h

```
1  /*!
2  * \file slcircularlist_utility.h
3  *
4  * \brief Définition de fonctions d'aide à l'utilisation de
5  *        struct SLCircularList.
6  */
```

```
7  #ifndef SLCIRCULARLIST_UTILITY_H
8  #define SLCIRCULARLIST_UTILITY_H
9
10 #include "slcircularlist.h"
11
12 /*!
13  * \brief Accès en lecture de la taille de la liste.
14  *
15  * La taille de la liste est le nombre de struct SLNode qui la
16  * constituent. Une liste vide est donc de taille nulle.
17  *
18  * Si `pSLCL` est `NULL`, le comportement de la fonction est
19  * indéterminé.
20  *
21  * \param pSLCL adresse de la struct SLCircularList dont on désire
22  *           connaître la taille.
23  *
24  * \return nombre d'éléments de la liste pointée par `pSLCL`.
25  */
26 size_t sizeSLCL(const struct SLCircularList * pSLCL);
27
28 /*!
29  * \brief Élément précédant celui fourni.
30  *
31  * Retourne l'élément de la liste `pSLCL` précédant `pSLN`
32  * ou `NULL` si la liste est vide.
33  *
34  * Si `pSLCL` est `NULL`, si `pSLN` est `NULL` sans que `pSLCL`
35  * ne soit vide ou si `pSLN` ne pointe pas sur un élément de la
36  * liste pointée par `pSLCL`, le comportement de la fonction est
37  * indéterminé.
38  *
39  * \param pSLCL adresse de la liste où rechercher le précédent.
40  * \param pSLN adresse de l'élément dont on désire le précédent.
41  *
42  * \return adresse de l'élément précédant `pSLN` ou `NULL` si
43  *         `pSLCL` est vide.
44  */
45 struct SLNode * previousSLCL(const struct SLCircularList * pSLCL,
46                             const struct SLNode * pSLN);
47
48 #endif // SLCIRCULARLIST_UTILITY_H
```