Méthodologie de la programmation : tests en C (exemple des listes)

Michel Billaud

20 novembre 2022

Table des matières

Première version 20 novembre 2022. Corrections typos 21 nov.

1 Pourquoi ce document

Le langage C n'est pas jeune, a plein de défauts, et est souvent très mal enseigné, surtout au regard des enjeux actuels : produire du code qui n'a pas trop d'erreurs.

Pour cela, il convient de sensibiliser les débutants[1] qui découvrent la programmation en C[2] (les malheureux). Un point qui est très souvent négligé, c'est l'idée de tester systématiquement le code que l'on écrit.

Mieux :

- d'**automatiser** les tests, pour qu'ils s'exécutent à chaque modification des sources, et pas seulement quand on n'aura que ça à faire d'y penser ;
- d'écrire les tests **avant** le code. Ca permet de réfléchir à ce que le code est censé faire, avant dêtre mentalement encombré par les détails de comment on envisage de le réaliser.

Pour enseigner ça, il n'est pas utile de montrer un "framework de test" à des débutants. Les bibliothèques industrielles c'est très utile pour les professionnels, mais là on s'adresse à des débutants qui sont déjà largement perdus dans les bases de C. Consacrer du temps à appréhender l'utilisation d'un framework (une mystérieuse usine à gaz qui offre plein de possibilités), c'est autant d'heures en moins consacrées aux bases de la programmation et de l'algorithmique.

En pratique, ils pourront à utiliser un framework le moment venu si jamais ils en ont besoin. Et ils apprendront d'autant plus vite qu'ils auront maîtrisé les bases de la programmation, sans être ralentis par l'apprentissage d'une usine à gaz.

Bref, ce document montre

- ce que fait `assert` ;
- comment on l'utilise dans le cadre d'exercices de programmation ;

sur un exemple classique : quelques exercices sur les listes chaînées.

[1] Pour leurs enseignants c'est souvent trop tard.

[2] L'idée de commencer à apprendre la programmation par un langage aussi rustique que bourré de défauts est clairement une abomination dans la deuxième décennie du troisième millénaire.

2 Utiliser la macro `assert()`

2.1 Utilisation basique

2.1.1 Un exemple de code source

Examinez le code suivant :

```c
// demo-assert.c

#include <stdio.h>
#include <assert.h>

int main()
{
    printf("Début\n");
    assert(2+2 == 4);
    assert(2+3 == 6);
    assert(3+3 == 6);
    printf("Fin\n");
}
```

Il contient

- 3 appels à `assert()`, avec comme paramètre une condition ;
- une directive d'inclusion de `assert.h`, où `assert()` est déclarée.

2.1.2 Exécution

Compilez et exécutez, il se produit

```
$ gcc demo-assert.c -o demo-assert
./demo-assert
Début
demo-assert: demo-assert.c:8: main: Assertion `2+3 == 6' failed.
Abandon
```

Vous remarquez que

- le programme affiche un message d'erreur à propos d'une condition condition (assertion) qui s'est trouvée fausse à l'exécution ;
- le premier test (vérification `2+2 == 4`) a réussi, silencieusement ;
- l'exécution est abandonnée au premier échec (sinon le message "Fin" aurait été affiché.

En résumé : à l'exécution, l'appel `assert(condition)`

- évalue la condition ;
- si la condition est fausse, affiche le code source de la condition, et termine le programme ;
- sinon, passe à la suite.

La macro prédéfinie `assert` fait partie du standard C. Les curieux trouveront dans l'annexe une implémentation simple.

2.2 Astuce d'utilisation

Une astuce traditionnelle est d'ajouter une chaîne de caractères qui sert de commentaire dans l'assertion. Exemple

```
assert("cas de base" && fib(1) == 1);
```

Du point de vue du langage C

- la chaîne de caractère est un pointeur non nul,
- converti en booléen comme paramètre de `&&`, ce pointeur non nul correspond à `true`,
- et par conséquent, l'assertion est équivalente à `fib(1) == 1`.

La présence de la chaîne ne change donc rien au test qui est effectué. Par contre, pour le programmeur qui teste, la chaîne s'affiche

```
fib: fib.c:13: tests_fib: Assertion `"cas de base" && fib(1) == 1' failed.
                                    -------------
```

et fournit **immédiatement** une indication supplémentaire, plutôt que d'avoir à aller consulter le fichier source à la recherche d'un commentaire expliquant la raison du test.

Ça parait peu de choses, mais la compréhension des erreurs est une activité intellectuelle intense, dans laquelle une petite tâche annexe comme "ouvrir un autre fichier pour aller voir les commentaires" est une distraction qui fait perdre de la concentration. Et ce n'est vraiment pas le moment.

Donc, on peut recommander de mettre un message dans les assertions.

3 Un projet traditionnel : les listes chainées simples

Ici on se place dans le cadre d'un projet traditionnel, écrire - à titre d'exercices -un certain nombre de fonctions qui agissent sur une liste de nombres.

Nous allons le faire, en suivant une méthodologie de tests systématiques.

3.1 Démarrage

Une liste est une suite de noeuds chaînés entre eux. Nous définissions deux types de données[3] :

```
typedef struct Node {
    int value;
    struct Node *next;
} Node;

typedef struct List {
    struct Node *first;
} List;
```

[3] On utilise ici `typedef` pour simplifier l'écriture des déclarations de structures.

Il faudra écrire des fonctions pour ajouter, enlever, etc. dans des listes.

Pour homogénéiser les fonctions, on va leur donner des noms

- en anglais,
- qui commencent par le préfixe `sl` (pour simple list),
- dont le premier paramètre est toujours un pointeur sur une `List`

3.2 Fonctions de base : liste vide

Commençons par le début : une fonction (`sl_init`) qui initialise une liste. Pour vérifier qu'elle fonctionne, on écrit aussi une fonction qui regarde si la liste est vide.

Si vous ne voyez pas comment faire, regardez la section "Indications" à la fin du document.

```
void test_init()
{
    List l;
    sl_init(&l);
    assert("liste vide après initialisation" && sl_is_empty(&l));
}
```

Exercice : écrire la fonction

```
void sl_init(List * list)
{
   ...
}
```

Attention : on se permet d'appeler le paramètre `list` alors qu'il ne contient pas une structure `List`, mais **l'adresse** d'une telle structure.

Exercice : en ayant inclus `stdbool.h`, écrire :

```
bool  sl_is_empty(const List * list)
{
   ...
}
```

On spécifie `const` parce que tester si une liste est vide ne doit pas modifier la liste.

Si vous n'y arrivez pas, voir dans l'annexe "Solutions".

3.3 Ajouter un élément au début

Pour cette fonction, on peut imaginer le test suivant

- mise en place
  - créer une liste vide
  - ajouter la valeur 10 au début
  - ajouter la valeur 30 au début
  - ajouter la valeur 20 au début
- vérifications

- la liste a 3 éléments
- le premier est 20 (le dernier ajouté)
- le second est 30
- le troisième est 10

Le test pourrait s'écrire

```
void test_add_first()
{
    List list;
    sl_init(&list);

    sl_add_first(&list, 10);
    sl_add_first(&list, 30);
    sl_add_first(&list, 20);

    assert("après 3 ajouts" && sl_size(&list) == 3);

    assert("premier de la liste"   && sl_value_at(&list, 0) == 20);
    assert("second de la liste"    && sl_value_at(&list, 1) == 30);
    assert("troisième de la liste" && sl_value_at(&list, 2) == 10);
}
```

en suivant la convention -raisonnable en C- d'indicer les éléments de la liste à partir de 0.

Exercice : partir du code suivant

```
void sl_add_first(List *list, int value)
{
    assert("A faire" && false);
}

int sl_size(const List *list)
{
    assert("A faire" && false);
}

int sl_value_at(const List *list, int index)
{
    assert("A faire" && false);
}
```

Ici `assert` est utilisée pour faire des "stubs" (bouchons), c'est-à-dire écrire des fonctions syntaxiquement correctes (donc compilables), mais qui font juste acte de présence, sans faire le boulot.

Le travail devient un cycle : compiler le source (qui est correct), exécuter, s'occuper de la première erreur qui apparaît, recommencer.

On peut aussi utiliser une macro `TODO` pour faciliter l'écriture des stubs :

```
int sl_size(const List *list)
{
```

```
    TODO("calcul de la taille d'une liste");
}
```

Ce n'est pas un macro standard, on peut se la définir soi-même en s'inspirant du code d'`assert`. Voir code en annexe.

3.4 Ajout à la fin

On donne inévitablement comme exercice l'ajout d'un élément en fin d'une liste chaînée simple. L'objectif est de faire manipuler les chaînages[4].

Voici le test

```
void test_add_last()
{
    List list;
    sl_init(& list);

    sl_add_last(&list, 11);
    assert("après 1 ajout" && sl_size(&list) == 1);
    assert("après 1 ajout" &&sl_value_at(&list, 0) == 11);

    sl_add_last(&list, 333);
    assert("après 2 ajouts" && sl_size(&list) == 2);
    assert("après 2 ajouts" &&sl_value_at(&list, 1) == 333);

    sl_add_last(&list, 2);
    assert("après 3 ajouts" && sl_size(&list) == 3);
    assert("après 3 ajouts" && sl_value_at(&list, 2) == 2);
}
```

Exercice : écrire la fonction `sl_add_last`

```
void sl_add_last(List *list, int value) {

    // A Faire

}
```

3.5 Comparaison liste / tableau

Vérifier qu'une liste contient bien les valeurs attendues serait bien plus simple ainsi :

```
void test_add_last()
{
    List list;
    sl_init(& list);
    sl_add_last(&list, 11);
```

[4] En pratique, si on a besoin souvent d'ajouter des valeurs à la fin d'une liste, c'est qu'une liste chaînée simple n'est pas la structure de données qui convient. Envisagez par exemple une liste avec un pointeur qui donne un accès direct au dernier élément (et peut-être une liste à double chaînage si on a besoin de retirer le dernier).

```
    sl_add_last(&list, 333);
    sl_add_last(&list, 2);

    int expected[] = { 11, 333, 2};                    // ici
    assert("après 3 ajouts à la fin"
           && sl_list_contains(&list, expected, 3));

}
```

Exercice : écrire la fonction

```
bool sl_list_contains(const List *list, int values[], int nb_values)
{


}
```

3.6 Remplissage d'une liste depuis un tableau

L'écriture des tests fait souvent apparaître des besoins, au niveau des fonctions sur le type de données.

C'était le cas ci-dessus pour `sl_list_contains`, qui nous permet de tester facilement qu'une liste contient les valeurs attendues.

Dans les tests, nous avions besoin d'ajouter des éléments, ce qui nous faisions laborieusement à la petite cuiller, un par un.

Pour simplifier, on peut imaginer une fonction qui ajoute un tableau de valeurs à la fin d'une liste.

Exemple de test :

- on crée une liste avec les valeurs 11, 22, 33
- on y ajoute 44 et 55
- on vérifie qu'on a bien 11, 22, 33, 44 et 55, dans cet ordre.

```
void test_add_values() {
    List list;
    sl_init(& list);

    int v1[] = {11, 22, 33};
    sl_add_values(&list, v1, 3);

    assert("construction liste" && sl_list_contains(&list, v1, 3));

    int v2[] = {44, 55};
    sl_add_values(&list, v2, 2);

    int expected[] = { 11, 22, 33, 44, 55};
    assert("ajout tableau" && sl_list_contains(&list, expected, 5));
}
```

Exercice : écrire la fonction

```
void sl_add_values(List *list, int values[], int nb_values)
{
   // A faire


}
```

4 Indications pour les exercices

4.1 Fonctions de base

Pour l'initialisation, il suffit d'affecter le membre `first` de la structure dont l'adresse est reçue en paramètre.

```
list->first = .... ;
```

Tester si la liste est vide se ramène à tester si `first` contient `NULL`.

4.2 Ajouter un élément au début de la liste

- allouer un nouveau maillon, y mettre la valeur ;
- ce maillon est maintenant le premier :
- la chaîne de maillons qui existait est dans le suivant de ce maillon.

4.3 Ajouter à la fin

- Trouver le dernier maillon de la liste (si il existe)
- créer un nouveau maillon avec la valeur à ajouter
- accrocher le nouveau maillon
  - soit comme premier de la liste si elle était vide
  - soit comme suivant du dernier maillon.

Pour trouver le dernier maillon, vous pouvez faire une boucle avec une variable `n`

- qui pointe successivement sur tous les maillons de la chaîne (comme pour le calcul de la taille),
- dans le corps de la boucle, vous notez l'ancienne valeur de `n` dans une variable `last`.

Ainsi, à la sortie de la boucle, on a dans `last` l'adresse du dernier maillon.

A voir : initialisation de `last` ?

4.4 Comparaison liste tableau

Vous pouvez par exemple faire une boucle avec un index qui varie de 0 à la taille du tableau moins 1.

Dans cette boucle, vous ferez aussi progresser un pointeur le long de la liste, et vous comparerez la valeur désignée par l'indice dans le tableau, et celle de l'élément pointé.

Attention :

- si ce pointeur est `NULL` *dans* la boucle, la liste est trop courte.
- si il n'est pas `NULL` *après* la boucle, la liste est trop longue.

4.5 Remplissage d'une liste depuis un tableau

La solution de facilité consiste à faire une boucle sur le tableau, et appeler `sl_add_last` pour chaque élément.

Ça ferait l'affaire pour une fonction qu'on n'appellerait que pour des tests. Vous pouvez commencer par faire comme ça.

Mais il y a un problème : pour le premier élément, `sl_add_last` n'exécute pas sa boucle. Pour le second, elle fait 1 tour. Pour le troisième, deux tours. Etc.

Si on fait le total, pour ajouter 100 éléments, ça fera 0+1+2+….+99 = 4950 tours. Avec de grosses listes, Le temps nécessaire grandit comme le **carré de la taille** de la liste. Pour de grosses listes, c'est trop.

Pour faire ça bien, dans `sl_add_values`, il faut se rappeler du dernier élément ajouté.

5 Solution des exercices

5.1 Fonctions de base

```
#include <stdbool.h>
#include <assert.h>

// ... mettre les déclarations ici

void sl_init(List *list)
{
    list->first = NULL;
}

bool sl_is_empty(const List *list)
{
    return list->first == NULL;
}

void test_init()
{
    List l;
    sl_init(&l);
    assert("liste vide après initialisation" && sl_is_empty(&l));
}

int main()
{
    printf("# Tests\n");
    test_init();
```

```
    printf("# OK\n");
}
```

5.2 Ajouter un élément au début

```
void sl_add_first(List *list, int value)
{
    Node *n = malloc(sizeof(Node));
    // A traiter : cas où l'allocation échoue
    n->value = value;
    n->next = list->first;
    list->first = n;
}

int sl_size(const List *list)
{
    int size = 0;
    for (Node *n = list->first; n != NULL; n = n->next) {
        size += 1;
    }
    return size;
}

int sl_value_at(const List *list, int index)
{
    Node *n = list->first;
    for (int i = 0; i < index; i++) {
        n = n->next;
    }
    return n->value;
}
```

5.3 Ajouter à la fin

```
void sl_add_last(List *list, int value)
{
    // recherche du dernier maillon
    Node *last = NULL;
    for (Node *n = list->first; n != NULL; n = n->next) {
        last = n;
    }

    // allocation nouveau maillon
    Node *new_node = malloc(sizeof(Node));
    new_node->value = value;
    new_node->next = NULL;

    // accrochage
    if (last == NULL) {
        // au début (la liste était vide)
```

```
        list->first = new_node;
    } else {
        // après le dernier
        last->next = new_node;
    }
}
```

5.4 Comparaison liste / tableau

```
bool sl_list_contains(const List *list, int values[], int nb_values)
{
    Node *node = list->first;
    for (int i = 0; i < nb_values; i++) {
        if (node == NULL) {
            return false;
        }
        if (node->value != values[i]) {
            return false;
        }
        node = node -> next;
    }
    return node == NULL;
}
```

5.5 Remplissage d'une liste depuis un tableau

Pour ajouter plusieurs éléments à la fin, on commence par localiser le dernier maillon. Les éléments du tableau s'accrocheront

- soit par le champ `next` de ce maillon si il existe ;
- soit par le champ `first` de la liste si la liste était vide.

Comme ces deux champs sont de même type, on peut uniformiser en notant l'adresse de ce champ.

```
void sl_add_values(List *list, int values[], int nb_values)
{
    // recherche de l'adresse du pointeur auquel on accrochera le
    // premier élément supplémentaire
    Node **addr_ptr = & list->first;
    for (Node *n = list->first; n != NULL; n = n->next) {
        addr_ptr = &(n->next);
    }

    // ajout des éléments du tableau
    for (int i = 0; i < nb_values; i++) {
        Node *new_node = malloc(sizeof(Node));
        new_node->value = values[i];
        new_node->next = NULL;
        // chainage du maillon
        *addr_ptr = new_node;
```

```
        // c'est à lui qu'on accrochera les suivants
        addr_ptr = & new_node->next;
    }
}
```

6 Compléments

6.1 Implémentation d'assert

Voici une version d'assert

```
// my-assert.c

#include <stdio.h>
#include <stdlib.h>

#define my_assert(condition) { \
    if(!(condition)) { \
       fprintf(stderr, "Dans %s ligne %d (fonction %s) l'assertion '%s' est fausse.\n",\
        __FILE__, __LINE__, __func__, #condition); \
       exit(1); \
    }\
}

int main()
{
    printf("Début\n");
    my_assert(2+2 == 4);
    my_assert(2+3 == 6);
    my_assert(3+3 == 6);
    printf("Fin\n");
}
```

qui produit presque la même chose que le programme du début.

```
$ gcc my-assert.c   -o my-assert
./my-assert
Début
Dans my-assert.c ligne 18 (fonction main) l'assertion '2+3 == 6' est fausse.
Abandon.
```

Explications

- `my_assert` ne peut pas être une fonction. C'est une **macro** pour afficher le code source de la condition, le nom du fichier et le numéro de ligne où elle est appelée, etc.
- `__FILE__`, `__LINE__` et `__func__` sont des macros standards du langage C.

6.2 Implémentation de `TODO`

Les mêmes mécanismes peuvent servir à réaliser une macro `TODO` qui accélère l'écriture des stubs.

Exemple :

```
// my-todo.c

#include <stdio.h>
#include <stdlib.h>

#define TODO(message) { \
       fprintf(stderr,  "Dans %s ligne %d (fonction %s) "\
                        "code manquant : %s.\n", \
        __FILE__, __LINE__, __func__, #message ); \
       exit(1); \
    }

// exemple d'utilisation
int fib(int n)
{
    if (n <=1) {
        return n;
    } else {
        TODO("valeurs > à 2");
    }
}

int main()
{
    printf("fib(0) = %d\n", fib(0));
    printf("fib(1) = %d\n", fib(1));
    printf("fib(2) = %d\n", fib(2));
}
```

Exécution

```
./my-todo
fib(0) = 0
fib(1) = 1
Dans my-todo.c ligne 19 (fonction fib) code manquant : "valeurs > à 2".
```