TP – DEV : Listes chaînées

4 au 6 décembre 2023
Erwan Kerrien

Rappel : si vous avez des questions sur ce TP ou sur le cours, n'hésitez pas à m'envoyer un mail à Erwan.Kerrien@inria.fr (je consulte plus rarement mon mail Erwan.Kerrien@univ-lorraine.fr).

Le TP est à rendre pour la fin de la session sur arche. Les questions notées sont en bas de page 1. Pour répondre à ces questions, vous devez me rendre soit un fichier `liste.c` , soit les fichiers nommés `liste.c`, `liste.h`, `element.c`, `element.h`, `Makefile` et `test_liste.c`. Respectez bien les noms, casse comprise ! Ces fichiers devront être correctement commentés. Vous pouvez, si vous le souhaitez et si vous le jugez utile, m'envoyer un texte de commentaires. Ne pas le faire ne vous exposera à aucun retrait de points. Ces fichiers sont à déposer sur arche.

1 Objectifs du TP

Ce TP a pour objectif d'implémenter les fonctions sur la sorte `Liste` construites lors des TD9&10.

On partira de la sorte que vous avez écrite lors du TP précédent. Au besoin vous pouvez reprendre la correction qui vous est fournie sur arche.

Dans une première partie (section 2), on met tout notre code dans un seul fichier nommé `liste.c`. Vous pourrez rendre ce fichier seul si vous le souhaitez. **Ce travail suffit comme rendu du TP.**

Pour celles et ceux qui iront assez vite, vous pourrez réaliser la deuxième partie (section 3) qui réorganise ce premier code suivant les principes de la programmation modulaire (principes que vous devrez suivre pour vos SAÉs). Dans ce deuxième cas, vous me rendrez les fichiers `element.c`, `element.h`, `liste.c`, `liste.h`, `Makefile`, ainsi qu'un fichier de programme `test_liste.c`.

Et pour celles et ceux qui veulent aller encore plus loin, je décris (plus brièvement du coup) quelques exercices supplémentaires que vous pouvez faire pour vous améliorer.

2 Les fonctions de la liste chaînée

En partant de ce que vous avez écrit lors du dernier TP, ou bien en reprenant la correction qui vous en est fournie, écrivez toutes les fonctions sur les listes qui sont commentées. Un bon ordre à suivre est celui donné dans le TD, mais vous pouvez suivre le vôtre. Basez-vous sur le travail d'algorithmique fait pendant le TD9&10. Vous me renverrez votre version du fichier `liste.c`.

Voici la liste exhaustive des fonctions à écrire (ordre suggéré) :

- `Longueur`
- `Afficher`
- `Rechercher` et `EstDans`
- `Dernier`
- `Supprimer`
- `Concatener`
- `Ajouter`
- `Inverser`
- `Copier`
- `Vider`

Faites bien attention au prototype de chaque fonction qui devra suivre celui indiqué dans la définition de la sorte. Vous ne devez utiliser que les fonctions écrites pour les types Element et Liste (par exemple utiliser `SuccLire` ou `SuccModif` au lieu `l->next`, ou encore `ElementComparer` au lieu de `E1 == E2`...) : votre code doit être indépendant de la manière dont les types Element et Liste ont été définis.

Éléments de notation :

- le nom de fichier (`liste.c`), le nom des types, constantes, celui des fonctions, ainsi que leur prototype (type de sortie et nombre et types des paramètres) ne doivent pas être changés : mes scripts automatiques seront impitoyables.
- assurez-vous a minima que le fichier que vous m'envoyez compile. Faites attention en particulier aux typos que vous insérez parfois dans votre code en le commentant au dernier moment.
- enfin, j'attends un code commenté : la qualité et la pertinence des commentaires entrera dans la notation.

3 Programmation modulaire

Jusqu'à présent, vous n'avez travaillé que sur un seul fichier source. Il suffisait donc de lancer la commande `gcc <fichier>.c -o <executable>` pour compiler le fichier `<fichier>.c` et générer le fichier exécutable `<executable>`.

Dans le cas d'un plus gros projet, il est impossible de ne travailler que sur un seul fichier : non seulement sa taille en rendrait la lecture extrêmement difficile, mais en plus, impaginez le nombre de conflits qui seraient à régler si des dizaines de développeurs travaillaient sur le même fichier !

Une génération d'exécutable en deux parties

Dès que le projet prend une certaine ampleur, la bonne pratique est donc de travailler sur plusieurs fichiers. La génération d'un fichier exécutable se fait alors selon deux étapes : la compilation et l'édition de liens.

- **La compilation** a pour but de vérifier la syntaxe du code source et de générer le code objet associé. À ce stade, on n'intègre pas de code extérieur. Tout appel à une fonction définie dans un autre fichier sera remplacé par un code d'appel qui ne nécessite que de connaître prototype de la fonction (c'est-à-dire son nom, le nombre, type et ordre des arguments, et le type de sortie). La commande pour générer un fichier objet à partir d'un fichier source est `gcc -c <fichier>.c` : cette commande génère le fichier objet `<fichier>.o`
- **l'édition de liens** a pour but de regrouper tous les fichiers objets en un fichier exécutable. C'est à cette étape qu'est vérifiée, pour chaque fonction appelée, qu'un code existe bien parmi tous les fichiers objets pour la définir (et donc décrire son exécution). Ce code peut être dans un fichier objet que vous avez généré mais également dans une bibliothèque externe disponible sur votre système. Un exemple est la libraire mathématique dont le code objet est disponible physiquement dans le fichier `/usr/lib/x86_64-linux-gnu/libm.so` sur un système linux.
  La commande pour générer un fichier exécutable à partir de plusieurs fichiers objets est : `gcc -o <executable> <fichier1>.o <fichier2>.o <fichier3>.o`
  Dans l'exemple précédent, on considère trois fichiers objets, mais bien sûr on peut en mettre plus ou moins. Pour indiquer un lien avec une bibliothèque, il faut l'indiquer avec l'option `-l` à laquelle on acolle le nom du fichier, sans `lib`, ni `.so`. Par exemple pour la bibliothèque mathématique, l'option sera `-lm` (correspondant au fichier `libm.so`).
  La commande pour générer un fichier exécutable est dans ce cas :
  `gcc -o <executable> <fichier1>.o <fichier2>.o <fichier3>.o -lm`
  Ici encore, on peut lier plusieurs bibliothèques, et chacun d'elle donnera lieu à l'ajout d'une option `-l`.

Découpage en modules

Une bonne organisation des fichiers d'un projet va donc définir un ensemble de modules et un fichier source principal. Dans notre cas, on a un type Element, auquel est associée une constante element_invalide et un ensemble de fonctions `ElementAfficher`, `ElementEstValide`, `ElementComparer`, `ElementCopie`, `ElementDetruire` : on va donc créer un module Element. Pour ce faire, on crée deux fichiers :

- **un fichier header**, que l'on va nommer `element.h` : ce fichier contient la définition du type, la constante, ainsi que le prototype de chaque fonction, qui se termine par un ; (point-virgule) et **sans le code de la fonction**. On y met aussi toutes les instructions `#include` qui sont nécessaires.
- **un fichier source** qui commence par une instruction `#include "element.h"`. Cela permet d'inclure automatiquement la définition du type Element, de la constante element_invalide et de déclarer toutes les fonctions. Ensuite, on ne met que le code des fonctions (avec leur prorotype), c'est-à-dire la partie de votre code initial où vous avez définie ces fonctions.

Le fichier `element.h` contiendra donc :

```
#ifndef ELEMENT_H
#define ELEMENT_H

#include <stdbool.h> // type booléen (bool: true/false): renvoyé par ElementEstValide et ElementCompare
#include <stdlib.h> // pour la constante NULL

// Définit un type Element, comme char*
typedef char* Element;

// la constante element_invalide
const static Element element_invalide = (Element)NULL;
```

```
12
13 // - les opérations (fonctions):
14
15 // Vérifie la validité d'un élément
16 // @param E: Element à vérifier
17 // @return true si E est valide, faux sinon
18 bool ElementEstValide(Element E);
19
20 // Affiche un élément
21 // @param E: élément à afficher
22 // @Note affiche <INVALIDE> en cas d'élément invalide
23 void ElementAfficher(Element E);
24
25 // Compare deux éléments
26 // @param E1, E2: les deux éléments à comparer
27 // @return true si E1 et E2 sont égaux, faux sinon
28 // @Note si E1 et E2 sont tous les deux invalides, la fonction renvoie true
29 bool ElementComparer(Element E1, Element E2);
30
31 // Effectue une copie profonde d'un élément (donc allocation de mémoire)
32 // @param E: élément à copier
33 // @return nouvel élément avec le même contenu que E.
34 //         Renvoie element_invalide si E n'est pas valide
35 // @Note l'élément renvoyé peut être détruit par appel à la fonction free.
36 //       Voir aussi la fonction ElementDetruire qui le fait proprement
37 Element ElementCopie(Element E);
38
39 // Destruction d'un élement (libération de mémoire)
40 // @param E: élément à détruire
41 // @return element_invalide
42 // @Note fonctionne même si E est invalide
43 Element ElementDetruire(Element E);
44
45 #endif // ELEMENT_H
```

On remarquera en début de fichier les deux lignes 1 et 2 qui définissent en fait une variable permettant d'indiquer que le fichier `element.h` a été inclu lors de la compilation. Cela évite les inclusions multiples, qui peuvent vite arriver dans de gros projets, et qui poseraient des problèmes de redéfinition de type/constantes et redéclarations de fonctions que le compilateur ne sait pas bien gérer. Ces deux lignes se terminent par le `#endif` de la dernière ligne (`ELEMENT_H` est mis en commentaire pour se souvenir de quelle instruction `#if` il s'agit de terminer ici). Prenez l'habitude de faire ça systématiquement afin d'éviter un certain nombre de soucis.

Par ailleurs, le fichier `stdbool.h` est inclu car nous utilisons le type `bool` pour déclarer les fonction `ElementEstValide` et `ElementComparer`. En revanche les fichiers `stdio.h` (pour `printf`) et `stdlib.h` (pour la constante `NULL` et les fonctions `malloc` et `free`) ne sont pas inclus : ils ne le seront que dans le fichier `element.c` dans lequel les fonctions et constantes impliquées seront effectivement utilisées. N'incluez toujours que le nombre minimal de fichiers pour garder un code qui compile vite.

Enfin, vous remarquerez que la constante element_invalide est définie en ligne 11 avec le mot-clé `static` : ce mot-clé indique que la variable ne sera déclarée qu'une seule fois, lors du premier passage du code par cette ligne. Ce mot-clé a d'autres impacts importants que je vous incite à rechercher sur internet. Les explications détaillées du mot-clé `static` dépassent le périmètre de ce cours d'introduction à C.

Exercice : Travail à faire (optionnel)

- recopiez le code des fonctions concernant le type Element dans un fichier `element.c`
- de manière similaire, créez le module Liste avec un fichier `liste.h` qui contiendra les définitions de type, ainsi que la constante liste_vide, dans lequel vous ajouterez les prototypes de toutes les fonctions associées à une Liste. Mettez ensuite le code de ces fonctions dans un fichier `liste.c`. Le fichier `liste.h` devra faire un `#include "element.h"` puisque celui-ci est utilisé dans le prototype des fonctions d'une Liste.
- créez un fichier `test_liste.c` qui commencera par inclure le fichier `liste.h` (qui lui-même inclut le fichier `element.h`, donc inutile de l'inclure explicitement à nouveau) et ne contiendra qu'une fonction `main` où mettrez un

code de test de vos fonctions.

Compilation modulaire automatisée

Pour compiler votre projets, il faut donc d'abord compiler le module Element, puis le module Liste, et enfin le programme principal. Et en plus il faut finir par une étape d'édition de liens avant d'avoir l'exécutable. Autrement dit, voici les commandes à lancées pour compiler le programme

```
> gcc -c element.c
> gcc -c liste.c
> gcc -c test_liste.c
> gcc -o test_liste test_liste.o liste.o element.o
```

Vous pouvez tester si votre refactorisation de codes en modules a été bien faite, en lançant ces commandes.

C'est cependant un peu pénible à lancer à chaque test. Par ailleurs, peut-être n'est-il pas nécessaire de tout recompiler à chaque fois (par exemple si vous ne modifiez que le fichier `liste.c`, il suffit de recompiler `liste.o` puis de refaire l'édition de liens). Pour vous aider à automatiser cela, il y a l'outil `make` (installable par `sudo apt install make` dans une fenêtre de commande linux). La commande `make` cherche automatiquement un fichier nommé `Makefile` dans le dossier courant et compile le projet suivant les instructions données. Ici encore, ce cours d'introduction n'est le lieu pour rentrer dans les détails de `make`. Je vous mets à disposition le `Makefile` suivant : copiez-le dans le dossier de votre projet (qui contient `element.h`, `element.c`, `liste.h`, `liste.c` et `test_liste.c`). Dans un terminal, placez-vous dans ce dosser et entrez la commande `make` : cela devrait compiler le projet.

```
1 all: test_liste
2
3 element.o: element.h
4 liste.o: liste.h element.h
5 test_liste.o: liste.h element.h
6 test_liste: test_liste.o liste.o element.o
7 clean:
8         $(RM) -rf *.o test_liste
```

Quelques explications toutefois. `make` fonctionne par cibles (*targets* en anglais) de compilation, indiquées par un nom suivi de :. Les résolutions de ces cibles sont récursives. Par exemple, le `Makefile` fourni commence par définir la cible `all` : c'est la cible qui résolue par défaut par un appel à `make`. `all` demande de résoudre la cible `test_liste`. Elle ne fait rien d'autre (pas de ligne supplémentaire pour la cible `all`). Du coup, `make` va résoudre la cible `test_liste` en ligne 6.

Cette cible dépend de fichiers objets `.o` : `make` va donc par défaut faire une édition de liens avec ces fichiers en entrée, mais auparavant, il faut regénérer chaque fichier qui a été modifié depuis la dernière génération de `test_liste`.

Pour cela, il va résoudre les cibles correspondant à chaque fichier objet (lignes 3,4 et 5). En face de chaque cible, on voit qu'elle dépend d'un fichier `.c` et d'un ou plusieurs fichiers `.h` : `make` comprend donc qu'il faut compiler ce fichier `.c`, et qu'il n'a besoin de le faire que si un des fichiers indiqués en dépendance a été modifié depuis la dernière génération du fichier objet.

Par résolution récursive des cibles, le fichier exécutable `test_liste` est ainsi regénéré, avec un minimum de compilations.

Une dernière cible est indiquée ligne 7 : il n'y a pas de dépendance, mais en revanche une commande spécifique est liée (ligne 8). Si vous entrez `make clean` dans le terminal, cette cible sera résolue, ce qui lance la commande : celle-ci efface tous les fichiers objets du dossier courant, ainsi que le fichier exécutable `test_liste`.

4 Extension #1 : récursivité

Les fonctions ci-dessus sont en général plutôt écrites en récursif alors que la correction du TD les donnent en itératif. Donnez-en une version recursive.

5 Extension #2 : `g_slist`

En pratique, vous n'aurez que rarement besoin d'implémenter une liste chaînée et vous en trouverez une version dans la librairie standard de votre langage de travail. En C, ce n'est pas le cas, mais il existe une librairie qui fait figure

de standard en environnement POSIX (Linux) : la `glib` qui définit notamment les listes chaînées grâce à la structure `g_slist` et les fonctions associées.

Prenez connaissance de la documentation à la page `https://docs.gtk.org/glib/struct.SList.html`.

La documentation pour compiler un code avec la glib est sur la page `https://docs.gtk.org/glib/compiling.html`.

Écrivez un programme exploitant les `g_slist` pour stocker une liste de réels.