Rattrapages C-Avancé

C-Avancé

31 juillet 2022

Version 1

Fabrice BOISSIER <fabrice.boissier@epita.fr>

Copyright

Ce document est destiné à une utilisation interne à EPITA.

Table des matières

1 Consignes Générales

Les informations suivantes sont très importantes :

Le non-respect d'une des consignes suivantes entraînera des sanctions pouvant aller jusqu'à la multiplication de la note finale par 0.

Ces consignes sont claires, non-ambiguës, et ont un objectif précis. En outre, elles ne sont pas négociables.

N'hésitez pas à demander si vous ne comprenez pas une des règles.

Consigne Générale 0 : Vous devez lire le sujet.
Consigne Générale 1 : Vous devez respecter les consignes.
Consigne Générale 2 : Vous devez rendre le travail dans les délais prévus.

Consigne Générale 3 : Le travail doit être rendu dans le format décrit à la section Format de Rendu.

Consigne Générale 4 : Le travail rendu ne doit pas contenir de fichiers binaires, temporaires, ou d'erreurs (`*~`, `*.o`, `*.a`, `*.so`, `*#*`, `*core`, `*.log`, `*.exe`, binaires, ...).

Consigne Générale 5 : Dans l'ensemble de ce document, la casse (caractères majuscules et minuscules) est très importante. Vous devez strictement respecter les majuscules et minuscules imposées dans les messages et noms de fichiers du sujet.

Consigne Générale 6 : Dans l'ensemble de ce document, `login` correspond à votre login.

Consigne Générale 7 : Dans l'ensemble de ce document, `nom1-nom2` correspond à la combinaison des deux noms de votre binôme (par exemple pour Fabrice BOISSIER et Mark ANGOUSTURES, cela donnera `boissier-angoustures`).

Consigne Générale 8 : Dans l'ensemble de ce document, le caractère ␣ correspond à une espace (s'il vous est demandé d'afficher ␣␣␣, vous devez afficher trois espaces consécutives).

Consigne Générale 9 : Tout retard, même d'une seconde, entraîne la note non négociable de 0.

Consigne Générale 10 : La triche (échange de code, copie de code ou de texte, ...) entraîne **au mieux** la note non négociable de 0.

Consigne Générale 11 : En cas de problème avec le projet, vous devez contacter le plus tôt possible les responsables du sujet aux adresses mail indiquées.

Conseil : N'attendez pas la dernière minute pour commencer à travailler sur le sujet.

2 Format de Rendu

Responsable(s) du projet :	**Fabrice BOISSIER** `<fabrice.boissier@epita.fr>`
Balise(s) du projet :	**`[RATT][CAV]`**
Nombre d'étudiant(s) par rendu :	1
Procédure de rendu :	Devoir/Assignment sur Teams
Nom du répertoire :	`login-RATT-CAV`
Nom de l'archive :	`login-RATT-CAV.tar.bz2`
Date maximale de rendu :	31/07/2022 23h42
Durée du projet :	2,5 semaines
Architecture/OS :	Linux - Ubuntu (x86_64)
Langage(s) :	C
Compilateur/Interpréteur :	**`/usr/bin/gcc`**
Options du compilateur/interpréteur :	**`-W -Wall -Werror -std=c99 -pedantic`**

Les fichiers suivants sont requis :

`AUTHORS`	contient le(s) nom(s) et prénom(s) de(s) auteur(s).
`Makefile`	le Makefile principal.
`README`	contient la description du projet et des exercices, ainsi que la façon d'utiliser le projet.
`configure`	le script shell de configuration pour l'environnement de compilation.

Un fichier **`Makefile`** doit être présent à la racine du dossier, et doit obligatoirement proposer ces règles :

`all`	*[Première règle]* lance la règle `lib`.
`clean`	supprime tous les fichiers temporaires et ceux créés par le compilateur.
`dist`	crée une archive propre, valide, et répondant aux exigences de rendu.
`distclean`	lance la règle `clean`, puis supprime les binaires et bibliothèques.
`check`	lance l'éventuelle suite de tests.
`lib`	lance les règles `shared` et `static`.
`shared`	compile l'ensemble du projet avec les options de compilations exigées et génère une bibliothèque dynamique.
`static`	compile l'ensemble du projet avec les options de compilations exigées et génère une bibliothèque statique.

Votre code sera testé automatiquement, vous devez donc scrupuleusement respecter les spécifications pour pouvoir obtenir des points en validant les exercices. Votre code sera testé en générant un exécutable ou des bibliothèques avec les commandes suivantes :

```
./configure
make distclean
./configure
make
```

Suite à cette étape de génération, les exécutables ou bibliothèques doivent être placés à ces endroits :

login-RATT-CAV/libmylinkedlist.a
login-RATT-CAV/libmylinkedlist.so

L'arborescence attendue pour le projet est la suivante :

login-RATT-CAV/
login-RATT-CAV/AUTHORS
login-RATT-CAV/README
login-RATT-CAV/Makefile
login-RATT-CAV/configure
login-RATT-CAV/check/
login-RATT-CAV/src/
login-RATT-CAV/src/linkedlist.c
login-RATT-CAV/src/linkedlist.h

Vous ne serez jamais pénalisés pour la présence de makefiles ou de fichiers sources (code et/ou headers) dans les différents dossiers du projet tant que leur existence peut être justifiée (des makefiles vides ou jamais utilisés sont pénalisés, des fichiers sources hors du dossier sources sont pénalisés).
Vous ne serez jamais pénalisés pour la présence de fichiers de différentes natures dans le dossier `check` tant que leur existence peut être justifiée (des fichiers de test jamais utilisés sont pénalisés).

3 Aide Mémoire

Le travail doit être rendu au format ***.tar.bz2***, c'est-à-dire une archive **bz2** compressée avec un outil adapté (voir **`man 1 tar`** et **`man 1 bz2`**).
Tout autre format d'archive (zip, rar, 7zip, gz, gzip, ...) ne sera pas pris en compte, et votre travail ne sera pas corrigé (entraînant la note de 0).

Pour générer une archive *tar* en y mettant les dossiers *folder1* et *folder2*, vous devez taper :

```
tar cvf MyTarball.tar folder1 folder2
```

Pour générer une archive *tar* et la compresser avec GZip, vous devez taper :

```
tar cvzf MyTarball.tar.gz folder1 folder2
```

Pour générer une archive *tar* et la compresser avec BZip2, vous devez taper :

```
tar cvjf MyTarball.tar.bz2 folder1 folder2
```

Pour lister le contenu d'une archive *tar*, vous devez taper :

```
tar tf MyTarball.tar.bz2
```

Pour extraire le contenu d'une archive *tar*, vous devez taper :

```
tar xvf MyTarball.tar.bz2
```

Pour générer des exécutables avec les symboles de debug, vous devez utiliser les flags **`-g -ggdb`** avec le compilateur. N'oubliez pas d'appliquer ces flags sur *l'ensemble* des fichiers sources transformés en fichiers objets, et d'éventuellement utiliser les bibliothèques compilées en mode debug.

```
gcc -g -ggdb -c file1.c file2.c
```

Pour produire des exécutables avec les symboles de debug, il est conseillé de fournir un script **`configure`** prenant en paramètre une option permettant d'ajouter ces flags aux **`CFLAGS`** habituels.

```
./configure
cat Makefile.rules
CFLAGS=-W -Wall -Werror -std=c99 -pedantic
./configure debug
cat Makefile.rules
CFLAGS=-W -Wall -Werror -std=c99 -pedantic -g -ggdb
```

Pour produire une bibliothèque statique *libtest.a* à partir des fichiers *test1.c* et *file.c*, vous devez taper :

```
cc -c test1.c file.c
ar cr libtest.a test1.o file.o
```

Pour produire une bibliothèque dynamique *libtest.so* à partir des fichiers *test1.c* et *file.c*, vous devez taper (pensez aussi à **-fpic** ou **-fPIC**) :

```
cc -c test1.c file.c
cc test1.o file.o -shared -o libtest.so
```

Pour compiler un fichier en utilisant une bibliothèque dont le **.h** se trouve dans un dossier spécifique, par exemple la *libxml2*, vous devez taper (pensez à vous assurer que les *includes* de la bibliothèque ont été entourés de chevrons **< >**) :

```
cc -c -I/usr/include test1.c
```

Pour lier plusieurs fichiers objets ensemble et avec une bibliothèque, par exemple la *libxml2*, vous devez d'abord indiquer dans quel dossier trouver la bibliothèque avec l'option **-L**, puis indiquer quelle bibliothèque utiliser avec l'option **-l** (n'oubliez pas de retirer le préfixe *lib* au nom de la bibliothèque, et surtout, que l'ordre des fichiers objets et bibliothèques est important) :

```
cc -L/usr/lib test1.o -lxml2 file.o -o executable.exe
```

Si la bibliothèque existe en version *dynamique* (**.so**) et en version *statique* (**.a**) dans le système, l'éditeur de lien choisira en priorité la version *dynamique*. Pour forcer la version *statique*, vous devez l'indiquer dans la ligne de commande avec l'option **-static** :

```
cc -static -L/usr/lib test1.o -lxml2 file.o -o executable.exe
```

Dans ce sujet précis, vous ferez du code en C et des appels à des scripts shell qui afficheront les résultats dans le terminal (donc des flux de sortie qui pourront être redirigés vers un fichier texte).

4 Exercice 1 - Bibliothèques statique et dynamique

Nom du(es) fichier(s) :	**`Makefile`**
Répertoire :	**`login-RATT-CAV/`**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Fonctions recommandées :	**`gcc(1)`**, **`ar(1)`**

Objectif : Le but de l'exercice est de faire fonctionner l'ensemble de votre projet avec un makefile simple, et de produire une bibliothèque statique ainsi qu'une dynamique.

Une bibliothèque est un ensemble de fonctions et procédures prêtes à être utilisées (ayant éventuellement des variables statiques, ou encore faisant appel à `malloc(3)`) par un utilisateur ne connaissant pas leur implémentation. L'utilisateur développe et écrit son propre programme (utilisant un `main` pour s'exécuter) et il peut éventuellement s'appuyer sur des bibliothèques (n'ayant pas de `main`) pour réutiliser des implémentations existantes.

Vous devez écrire le (ou les) *makefile(s)* de votre projet (et éventuellement un fichier `configure`) afin de produire une bibliothèque statique nommée **`libmylinkledlist.a`** et une bibliothèque dynamique nommée **`libmylinkledlist.so`**.

Le *makefile* que vous allez écrire rendra votre projet complet et autonome. Pour cela, vous devez faire en sorte que plusieurs *cibles* soient présentent dans le makefile (celles-ci sont rappelées dans le paragraphe suivant).

Plusieurs stratégies existent pour compiler avec des makefiles, l'une d'entre elle consiste à placer un makefile par dossier afin que le makefile principal appelle les suivants avec la bonne règle (par exemple : le makefile principal va appeler le makefile du dossier *src* pour compiler le projet, mais le makefile principal appellera celui du dossier *check* lorsque l'on demandera d'exécuter la suite de tests).

Pour ce premier contact avec les makefiles, vous pouvez vous contenter d'un seul makefile à la racine exécutant des lignes de compilation toutes prêtes sans aucune réécriture ou extension.

Afin de générer des bibliothèques statiques et dynamiques, vous devez compiler vos fichiers pour produire des fichiers *objets* (des fichiers **`.o`**), mais vous ne devez pas laisser l'étape d'*édition de liens* classique se faire. À la place, la cible *static* de votre makefile devra produire une bibliothèque statique nommée **`libmylinkledlist.a`**, et la cible *shared* devra produire une bibliothèque dynamique nommée **`libmylinkledlist.so`**.

Pour générer une bibliothèque statique (*static library* en anglais), on utilise la commande **`ar`** qui créée des *archives*. Pour produire la bibliothèque statique *libtest.a* à partir des fichiers *test1.c* et *file.c*, on utilisera ces commandes :

```
cc -c test1.c file.c
ar cr libtest.a test1.o file.o
```

Pour générer une bibliothèque dynamique (*shared library* en anglais), on utilise l'option **-shared** du compilateur. Pour produire la bibliothèque dynamique *libtest.so* à partir des fichiers *test1.c* et *file.c*, on utilisera ces commandes :

```
cc -c test1.c file.c
cc test1.o file.o -shared -o libtest.so
```

La première commande génère des fichiers objets, et la deuxième les réunit dans un seul fichier. Il arrivera sur certains systèmes qu'il soit nécessaire d'ajouter l'option **-fpic** ou **-fPIC** (*Position Independent Code*) pour générer des bibliothèques dynamiques. Gardez en tête que *toutes* les bibliothèques **doivent** être préfixées par un `lib` dans le nom des fichiers **.a** et **.so** générés. Mais, lorsque l'on utiliser une bibliothèque dans un projet, on doit ignorer ce préfixe lors de la compilation et l'édition de liens.

Pour utiliser une bibliothèque dynamique sur votre système, plusieurs méthodes existent selon le système où vous vous trouvez.

— L'une d'entre elle consiste à ajouter le dossier contenant la bibliothèque à la variable d'environnement **LD_LIBRARY_PATH** (comme pour la variable **PATH**). Selon votre système, il peut s'agir de la variable d'environnement **DYLD_LIBRARY_PATH**, **LIBPATH**, ou encore **SHLIB_PATH**. Pour ajouter le dossier courant comme dossier contenant des bibliothèques dynamiques en plus d'autres dossiers, vous pouvez taper :
export LD_LIBRARY_PATH=.:/usr/lib:/usr/local/lib

— Une autre méthode consiste à modifier le fichier **/etc/ls.so.conf** et/ou ajouter un fichier contenant le chemin vers votre bibliothèque dans **/etc/ld.so.conf.d/** puis à exécuter **/sbin/ldconfig** pour recharger les dossiers à utiliser.

— Enfin, vous pouvez demander les droits administrateur et installer votre bibliothèque dans **/usr/lib** ou **/usr/local/lib**.

Évidemment, à long terme, l'objectif est d'avoir une bibliothèque installée dans le répertoire **/usr/lib**. Mais, lors du développement, il est préférable d'utiliser la variable **LD_LIBRARY_PATH**. Pour vous assurer du fonctionnement de votre exécutable, vous pouvez utiliser **ldd(1)** avec le chemin complet vers un exécutable. **ldd** vous indiquera quelles bibliothèques sont requises, et où celui-ci les trouve. Si l'une d'entre elle n'est pas trouvée, **ldd** vous en informera :

```
ldd /usr/bin/ls
ldd my_main
```

Pour rappel voici les cibles demandées pour le **Makefile** principal à la racine de votre projet :

`all`	*[Première règle]* lance la règle `lib`.
`clean`	supprime tous les fichiers temporaires et ceux créés par le compilateur.
`dist`	crée une archive propre, valide, et répondant aux exigences de rendu.
`distclean`	lance la règle `clean`, puis supprime les binaires et bibliothèques.
`check`	lance le(s) script(s) de test.
`lib`	lance les règles `shared` et `static`.
`shared`	compile l'ensemble du projet avec les options de compilations exigées et génère une bibliothèque dynamique.
`static`	compile l'ensemble du projet avec les options de compilations exigées et génère une bibliothèque statique.

5 Exercice 2 - Liste chaînée

Nom du(es) fichier(s) :	**linkedlist.c**
Répertoire :	**login-RATT-CAV/src/**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	640
Fonctions autorisées :	**malloc(3)**, **free(3)**, **memcpy(3)**, **printf(3)**

Objectif : Le but de l'exercice est d'implémenter une liste chaînée en C.

Les fonctions demandées dans cet exercice devront se trouver dans une bibliothèque nommée **libmylinkedlist**. Après un appel à la commande `make` à la racine du projet, il faut que votre chaîne de compilation produise à la racine de votre projet une version statique de la bibliothèque (qui se nommera **libmylinkedlist.a**) ainsi qu'une version dynamique de la bibliothèque (qui se nommera **libmylinkedlist.so**).

Vous devez écrire plusieurs fonctions permettant de créer, utiliser, vider, et libérer une liste. Un fichier **linkedlist.h** contenant toutes les fonctions exportables à implémenter vous est fourni en annexe. Vous devez déclarer une structure **my_linkedlist** et l'ajouter dans **linkedlist.h**. N'oubliez pas de déclarer également une structure qui contiendra les éléments de la liste chaînée. Pour les premières étapes, vous devrez implémenter une version simplifiée de la file qui ne prend en charge que des entiers positifs.

Conceptuellement, les fonctions manipulant des files de type **my_linkedlist*** devront pouvoir gérer ces 3 cas :

Vous devez implémenter les fonctions suivantes :

```
my_linkedlist *ll_create(void);
void ll_delete(my_linkedlist *ll);

int ll_length(my_linkedlist *ll);

int ll_head(my_linkedlist *ll);
int ll_tail(my_linkedlist *ll);
int ll_get(int pos, my_linkedlist *ll);

int ll_clear(my_linkedlist *ll);
int ll_is_empty(my_linkedlist *ll);

int ll_insert(int elt, int pos, my_linkedlist *ll);
int ll_replace(int elt, int pos, my_linkedlist *ll);
int ll_remove(int pos, my_linkedlist *ll);

int ll_search(int elt, my_linkedlist *ll);
my_linkedlist *ll_reverse(my_linkedlist *ll);
void ll_print(my_linkedlist *ll);
```

Liste des fonctions pour une liste chaînée

`my_linkedlist *ll_create(void)`

Cette fonction crée une liste vide. En cas d'erreur (pas assez de mémoire), elle renvoie un pointeur **NULL**.

`void ll_delete(my_linkedlist *ll)`

Cette fonction vide une liste de l'ensemble de ses éléments, et détruit la structure restante. Si le paramètre donné est **NULL**, la fonction ne fait rien.

`int ll_length(my_linkedlist *ll)`

Cette fonction renvoie la longueur de la liste (c'est-à-dire le nombre d'éléments actuellement dans la liste). Si le paramètre donné est **NULL**, la fonction renvoie -1.

`int ll_head(my_linkedlist *ll)`

Cette fonction renvoie l'élément en tête de liste (position 0). Si la liste donnée en paramètre est **NULL**, la fonction renvoie -1. Si la liste donnée en paramètre est vide, la fonction renvoie -2.

`int ll_tail(my_linkedlist *ll)`

Cette fonction renvoie l'élément en queue de liste (dernière position). Si la liste donnée en paramètre est **NULL**, la fonction renvoie -1. Si la liste donnée en paramètre est vide, la fonction renvoie -2.

`int ll_get(in pos, my_linkedlist *ll)`

Cette fonction renvoie l'élément stocké à la position *pos* de la liste. Si la liste donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la liste donnée en paramètre est vide, la fonction renvoie -2.

`int ll_clear(my_linkedlist *ll)`

Cette fonction vide une liste de l'ensemble de ses éléments, sans détruire la structure de la liste. La fonction renvoie le nombre d'éléments supprimés de la mémoire. Si le paramètre donné est **`NULL`**, la fonction renvoie -1. Si la liste donnée en paramètre est vide, la fonction renvoie 0.

`int ll_is_empty(my_linkedlist *ll)`

Cette fonction teste si une liste est vide ou non. Si la liste est vide, la fonction renvoie 1. Si la liste n'est pas vide, la fonction renvoie 0. Si la liste donnée en paramètre est **`NULL`**, la fonction renvoie -1.

`int ll_insert(int elt, int pos, my_linkedlist *ll)`

Cette fonction ajoute un élément dans la liste à l'emplacement *pos*. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire la tête de la liste), cette position sera numérotée 0. L'ancien élément qui était présent à cette position est décalé d'un cran en arrière (vers la queue). Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. Si l'emplacement n'existe pas et est positif, on ajoute l'élément en queue. Si l'emplacement n'existe pas et est négatif, on ajoute l'élément en tête. En cas de succès, la fonction renvoie 0. Si la liste donnée en paramètre est **`NULL`**, la fonction renvoie -1. S'il y a un problème de mémoire, la fonction renvoie -3.

`int ll_replace(int elt, int pos, my_linkedlist *ll)`

Cette fonction remplace un élément dans la liste à l'emplacement *pos*, et renvoie l'élément qui était présent à cet endroit. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire la tête de la liste), cette position sera numérotée 0. Si le nombre donné en paramètre est inférieur à 0, la fonction renvoie -4. Si l'emplacement n'existe pas et est positif, on remplace l'élément en queue. Si l'emplacement n'existe pas et est négatif, on remplace l'élément en tête. Si la liste donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la liste donnée en paramètre est vide, la fonction renvoie -2.

`int ll_remove(int pos, my_linkedlist *ll)`

Cette fonction supprime l'élément stocké à la position *pos* de la liste. Si la liste donnée en paramètre est **`NULL`**, la fonction renvoie -1. Si la liste donnée en paramètre est vide, la fonction renvoie -2.

`int ll_search(int elt, my_linkedlist *ll)`

Cette fonction recherche un élément dans la liste et renvoie sa position. La première position est celle où l'élément le plus ancien a été placé (c'est-à-dire la tête de la liste),

cette position sera numérotée 0. Si l'élément n'est pas trouvé, la fonction renvoie -4. Si la liste donnée en paramètre est **NULL**, la fonction renvoie -1.

my_linkedlist *ll_reverse(my_linkedlist *ll)

Cette fonction inverse la position de tous les éléments de la liste. Le premier élément devient le dernier, l'avant dernier devient le deuxième, etc. En cas de succès, la fonction renvoie le pointeur vers l'éventuelle nouvelle adresse en mémoire de la structure de la liste inversée. En cas de problème mémoire, on renvoie **NULL**, et l'ancienne liste doit rester à son ancienne adresse mémoire sans subir la moindre modification. Si la liste donnée en paramètre est **NULL**, la fonction renvoie **NULL**.

void ll_print(my_linkedlist *ll)

Cette fonction affiche le contenu de la liste. Le format d'affichage attendu implique d'afficher un seul élément par ligne, suivi d'un retour à la ligne. L'élément en tête de file sera affiché en premier. Si la liste donnée en paramètre est vide, seul un retour à la ligne est affiché. Si la liste donnée en paramètre est **NULL**, rien n'est affiché.

```
$ ./my_linked_list
42
5
13

$
```

Exemple d'affichage du cas normal : liste chaînée contenant 42, 5, 13 dans cet ordre précis

```
$ ./my_linked_list

$
```

Exemple d'affichage d'une liste chainée vide

```
$ ./my_linked_list
$
```

Exemple d'affichage d'un pointeur NULL

6 Exercice 3 - Suite de tests

Nom du(es) fichier(s) :	**`check.sh`**
Répertoire :	**`login-RATT-CAV/check/`**
Droits sur le répertoire :	750
Droits sur le(s) fichier(s) :	750
Fonctions recommandées :	**`diff(1)`**, **`find(1)`**, **`printf(3)`**

Objectif : Le but de l'exercice est de construire une suite de tests pour valider le fonctionnement de la liste chaînée.

Vous devez écrire plusieurs tests démontrant que votre implémentation de la liste chaînée fonctionne. Pour cela, vous devrez faire plusieurs programmes en C qui utilisent votre liste chaînée, et un script shell qui compare les résultats.
Dans les exercices précédents, vous n'avez pas écrit de fonction *main* pour la bonne raison qu'il s'agissait d'implémenter une structure (une liste chaînée). Il est maintenant temps de réutiliser tous les tests que vous avez écrits lors du développement de votre liste chaînée ! Vous devez néanmoins penser aux cas un peu plus complexes que le cas général.

Un exemple avec 3 programmes C de test serait :
- Un programme qui imprime l'état de la liste chaînée à différents moments, et imprime certains retours de fonctions.
- Un programme qui imprime l'état d'une liste chaînée vide, appelle plusieurs fonctions, et imprime les retours de fonctions.
- Deux programmes témoins qui serviront de vérité absolue sur le comportement attendu (ou les sorties textes que des programmes parfaits devraient imprimer).
- Chaque programme et son témoin testeront les mêmes valeurs dans le même ordre, afin que le même comportement soit visible d'un point de vue utilisateur.

Votre fichier **`main.c`** pourra contenir un scénario précis à suivre qui imprimera différentes valeurs dans le terminal, et vous pouvez éventuellement comparer cela à une sortie texte que vous avez manuellement préparée dans un fichier. Ainsi, le fichier préparé manuellement sera stocké en dur dans le dossier *check* et sera comparé avec un **`diff`** pour confirmer que tout se passe bien, ou au contraire que des problèmes existent lors de certains tests.

Pour utiliser une bibliothèque statique, lors du développement, vous devez disposer de la bibliothèque (le **`.a`** ou **`.so`**) et du fichier **`.h`** associé à l'interface (l'API) de votre bibliothèque, c'est-à-dire aux fonctions exportées.

Lors de la phase de *compilation*, vous devrez parfois ajouter un paramètre *INCLUDE* indiquant le dossier où trouver les *headers* de la bibliothèque (par exemple pour la *libxml2*, on ajoutera ce flag : **`-I/usr/include/libxml2`**). Ce flag **`-I`** permet d'ajouter un dossier dans lequel chercher les fichiers d'en-têtes propres aux bibliothèques (donc les **`.h`** entourés de **`< >`** dans vos *includes*). Vous pouvez cumuler plusieurs flags **`-I`** à la suite.

Lors de la phase d'*édition de liens*, vous devrez parfois ajouter un paramètre *LIBRARY* indiquant le nom de la bibliothèque à utiliser (par exemple pour la *libxml2*, on ajoutera ce flag : **`-lxml2`**). Ce flag **`-l`** (littéralement : *tiret L minuscule*) permet d'indiquer à

l'éditeur de lien qu'il doit utiliser une bibliothèque dynamique dont le nom est donné en paramètre (attention : le préfixe *lib* est supprimé des noms de bibliothèques). Attention, le placement de la bibliothèque par rapport aux fichiers objets (les fichiers **`.o`**) dans la ligne de commande a une importance.

Vous devrez également indiquer dans quel dossier trouver les bibliothèques grâce au flag **`-L`** (par exemple, pour chercher une bibliothèque dans le dossier courant, on ajoutera le flag : **`-L.`**). Vous pouvez cumuler plusieurs flags **`-L`** et **`-l`** à la suite.

Si vous disposez de la même bibliothèque avec une version statique et une dynamique, l'*éditeur de liens* choisira la version dynamique par défaut, mais vous pouvez forcer la statique grâce au flag **`-static`**.

La méthode idéale consiste à produire plusieurs scénarios numérotés et décrits, puis d'afficher quels tests ont réussi, et lesquels ont échoué. Pour cet exercice, vous n'êtes pas obligés de descendre à ce niveau de précision. Une simple petite suite de tests est suffisante, mais n'oubliez pas de vérifier autre chose que le cas général.

N'hésitez pas à vous appuyer sur le fait que votre projet produit des bibliothèques ! Vous pouvez développer un programme entier qui affiche une suite de tests avec un script shell très simple appelant ce programme, ou, vous pouvez utiliser un script shell plus complet qui fait de nombreux tests en s'appuyant sur un ou des programmes C très simples.