Logiciel de base
Première année par alternance

Responsable : Christophe Rippert
Christophe.Rippert@Grenoble-INP.fr

Ecriture de programmes modulaires

Introduction

On va voir dans cette partie comment écrire des programmes modulaires, découpés en plusieurs modules distincts du programme principal : c'est bien sûr toujours une bonne pratique de découper son code en sous-parties, plutôt que de tout écrire dans un seul fichier .c !
Avant cela, on fait quelques rappels concernant la visibilité et la durée de vie des différents types de variables existant en C.

Déclaration et allocation des variables

Il existe en C (comme dans la plupart des autres langages impératifs) différents types de variables, principalement :

- des variables locales à une fonction, qui sont allouées dans la pile d'exécution de la fonction ;
- des variables globales au programme, qui sont allouées dans la zone data du processus.

Variables locales

Prenons l'exemple de la fonction suivante :

```
void fct(void)
{
    int32_t val;
    static uint32_t cpt = 1;
    val = 5;
    cpt++;
}
```

La variable `val` est une variable locale classique : elle est visible uniquement dans la fonction `fct` et sa valeur est réinitialisée à chaque appel à `fct` (cela signifie que la variable `val` n'a pas de valeur au début de chaque appel à `fct` et qu'il faut donc lui donner une valeur dans le code de la fonction).
La variable `cpt` est quant à elle une variable locale statique : cela signifie qu'elle n'est visible que dans fct mais que sa valeur est préservée entre chaque appel à fct. Ce type de variables est typiquement utilisée pour implanter des compteurs d'appel à la fonction.
Dans l'exemple donné, si on appelle 3 fois la fonction `fct`, la variable `cpt` aura donc pour valeur 1 au premier appel, 2 au deuxième et 3 au troisième. Notez bien que l'initialisation de `cpt` à 1 lors de sa déclaration n'est prise en compte que lors du premier appel. A-contratio, si on avait écrit `cpt = 1;` ailleurs que lors de son initialisation, la valeur de `cpt` aurait été remisé à 1 à chaque appel.

Variables globales

Prenons l'exemple du programme `prog.c` suivant :

```
uint32_t val;

void fct1(void)
```

```
{
    val = 1;
}

static int32_t cpt;

void fct2(void)
{
    cpt++;
}

int main(void)
{
    cpt = 0;
}
```

et d'un module `mod.c` faisant partie du même programme :

```
extern uint32_t val;

void proc(void)
{
    val++;
}
```

La variable `val` est une variable globale au programme : elle est alloué dans la zone data du fichier objet `prog.o` : comme les zones data des différents fichiers objets composant le programme sont fusionnées lors de l'édition de liens, sa valeur peut être accédée (lue ou écrite) dynamiquement aussi bien depuis `fct1` et `fct2` que depuis `proc`).
Par contre, cette variable n'est par défaut pas visible dans `mod.c` : comme la compilation en C est séparée, le compilateur va générer un message d'erreur lors de la compilation du fichier `mod.c` car `proc` utilise cette variable. C'est pour cela qu'on doit rajouter la ligne `extern uint32_t val` dans le fichier `mod.c` : le mot clé `extern` permet de dire au compilateur qu'il existe une variable `val` de type entier naturel sur 32 bits mais qu'elle est alloué dans la zone data d'un autre fichier objet (cela permet au compilateur de vérifier que les accès à `val` sont bien cohérents avec son type).
Que se passerait-il si on ne mettait pas le mot clé `extern` devant la déclaration de `val` dans `mod.c` ?

- la compilation de `proc.c` et de `mod.c` se passerait sans problème, vu que le compilateur compile séparément ces deux fichiers ;
- par contre, au moment de la fusion des zones data des deux fichiers objets, l'éditeur de lien génèrerait un message d'erreur, car il trouverait deux variables s'appelant `val` dans le même programme !

Réciproquement, la variable `cpt` est une variable globale statique : cela signifie qu'elle n'est accessible que depuis les fonctions du fichier `prog.c` et pas depuis le module `mod.c` : dans ce contexte, le mot clé `static` se rapproche du mot clé `private` du langage Java par exemple.

En résumé

Les deux tableaux ci-dessous résument les conditions de visibilité et d'accès à une variable déclarée dans un certain module :

Type de variables	Zone d'allocation	Durée de vie
locale à une fonction	pile de la fonction	un appel à la fonction
locale statique à une fonction	zone data du module	vie du programme
globale	zone data du module	vie du programme
globale statique	zone data du module	vie du programme
globale externe	zone data d'un autre module	vie du programme

Type de variables	Visibilité statique	Accès dynamique
locale à une fonction	le corps de la fonction	le corps de la fonction
locale statique à une fonction	le corps de la fonction	le corps de la fonction
globale	toutes les fonctions du module	toutes les fonctions du programme
globale statique	toutes les fonctions du module	toutes les fonctions du module
globale externe	toutes les fonctions du module	toutes les fonctions du programme

Ecriture de modules

Le langage C offre un support assez simpliste de la notion de module. Supposons que l'on veuille implanter un module de gestion d'une table de hachage contenant des entiers naturels sur 8 bits (`uint8_t`). On va devoir créer deux fichiers :

— un fichier `tablehash.c` contenant les différentes fonctions de notre module ;
— un fichier d'en-tête (*header*) `tablehash.h` contenant l'interface de notre module.

Implantation du corps du module

Le fichier .c aura une forme assez classique :

```
#include "liste.h"

#define TAILLE_TABLE 4

struct tablehash_t {
    uint64_t nbr_elem;
    struct liste_t *table[TAILLE_TABLE];
};

static int64_t hash(uint8_t x)
{
// code de la fonction
}

struct tablehash_t *nouv_tablehash(void)
{
// code de la fonction
}

bool est_vide_tablehash(struct tablehash_t *th)
{
// code de la fonction
}

void inserer_val_tablehash(uint8_t val, struct tablehash_t *th)
{
// code de la fonction
}

void supprimer_val_tablehash(uint8_t val, struct tablehash_t *th)
{
// code de la fonction
}

void afficher_tablehash(struct tablehash_t *th)
{
// code de la fonction
```

```
}

void detruire_tablehash(struct tablehash_t **th)
{
// code de la fonction
}
```

Dans ce fichier, on commence par inclure l'interface d'un autre module, `liste.h`, qui implante vraisemblablement le type des listes chainées qu'on va utiliser.
On définit ensuite le type de données qui seront manipulés : ici, une table de hachage est composée d'un entier contenant le nombre d'éléments de la table, et d'un tableau de listes chainées contenants les valeurs (le type `struct liste_t` est certainement défini dans le module de gestion des listes qu'on a importé).
Le fait que ce type `tablehash_t` soit défini dans le fichier .c le rend privé : cela veut dire qu'un autre module, ou le programme principal, ne pourront par accéder directement par exemple au champ `nbr_elem` d'une table de hachage. C'est une bonne pratique de ne rendre visible que ce qui a besoin de l'être, pour éviter des erreurs.
Ensuite, on implante une fonction `static uint64_t hash(uint8_t x)` : le mot clé `static` rend cette fonction privée, ce qui signifie qu'elle ne pourra pas être appelée par un autre module ou le programme principal.
Enfin, on implante d'autres fonctions de gestion d'une table de hachage : l'absence du mot clé `static` implique que ces fonctions sont toutes publiques (c'est le comportement par défaut en C), ce qui veut dire qu'elles pourront être appelées par un autre module ou le programme principal.

Implantation du fichier d'en-tête (interface) du module

Le fichier d'en-tête correspondant pourra prendre la forme suivante :

```
#ifndef CR_TABLEHASH_H
#define CR_TABLEHASH_H

#include <stdbool.h>
#include <inttypes.h>

struct tablehash_t *nouv_tablehash(void);

bool est_vide_tablehash(struct tablehash_t *);

void afficher_tablehash(struct tablehash_t *);

void inserer_val_tablehash(uint8_t, struct tablehash_t *);

void supprimer_val_tablehash(uint8_t, struct tablehash_t *);

void detruire_tablehash(struct tablehash_t **);

#endif
```

Ce fichier commence par des directives gérées par le pré-processeur. On a déjà parlé de la directive `#include` qui inclut des fichiers d'en-tête dans le fichier contenant la macro. On rappelle que ce remplacement est récursif : si le fichier inclut contient lui-même des `#include`, les fichiers qu'il inclut seront aussi inclut dans le fichier initial. En pratique, cela peut poser des problèmes : si on inclut dans un fichier `prog.c` les fichiers d'en-tête `abc.h` et `xyz.h`, mais que le fichier `xyz.h` inclut lui-même le fichier `abc.h`, on peut se retrouver avec des doubles définitions de constantes ou de variables globales, ce qui est interdit en C.
Pour comprendre ce problème, on peut regarder l'exemple simpliste ci-dessous, qui comprend deux fichiers d'en-tête `un.c` et `deux.h`, et un programme principal dans le fichier `trois.c` :

```
// Fichier un.h
```

```
#include <inttypes.h>

// on definit une variable globale
uint64_t x = 5;

// on definit une constante
#define Y 10

// Fichier deux.h

// on inclut un.h parce qu'on a besoin de la constante Y
#include "un.h"
#include <inttypes.h>

// on definit une autre variable globale
uint32_t y = Y;

// Fichier trois.c

// on inclut les deux fichiers d'en-tete pour pouvoir acceder a x et y
#include "un.h"
#include "deux.h"
#include <stdio.h>

int main(void)
{
    printf("x = %" PRIu64 ", y = %" PRIu32 "\n", x, y);
    return 0;
}
```

A la compilation de `trois.c`, le compilateur affiche un message d'erreur expliquant que `x` est définie deux fois. Pour comprendre pourquoi, il suffit de jouer au pré-processeur en regardant ce que devient le fichier `trois.c` une fois qu'on a inclut les deux fichiers d'en-tête :

```
// on inclut les deux fichiers d'en-tete pour pouvoir acceder a x et y
// #include "un.h" :
#include <inttypes.h>

// on definit une variable globale
uint64_t x = 5;

// on definit une constante
#define Y 10

// #include "deux.h" :
// on inclut un.h parce qu'on a besoin de la constante Y
// #include "un.h" :
#include <inttypes.h>

// on definit une variable globale
uint64_t x = 5;

// on definit une constante
#define Y 10

#include <inttypes.h>
```

```
// on definit une autre variable globale
uint32_t y = Y;

#include <stdio.h>

int main(void)
{
    printf("x = %" PRIu64 ", y = %" PRIu32 "\n", x, y);
    return 0;
}
```

On voit bien la double définition de `x`.

Pour éviter cela, on utilise la directive `#ifndef CONSTANTE` : cette directive est un `if` pour le pré-processeur, sa sémantique est « si la constante `CONSTANTE` n'est pas déjà définie lors de cette passe du pré-processeur, alors inclut le code qui suit jusqu'au prochain `#endif` ».

Dans l'exemple de la table de hachage, on comprend donc que la première fois que le fichier `tablehash.h` sera inclut dans un autre fichier, le texte qu'il contient sera inclut puisque la constante `CR_TABLEHASH_H` ne sera (normalement) pas déjà définie. Mais comme on commence le texte du fichier par définir cette directive, si le même fichier `tablehash.h` est inclut une deuxième fois, alors la deuxième fois aucun texte ne sera copié-collé puisque la constante existera déjà.

Reste le problème du choix du nom de la constante : en effet, il ne faudrait pas tomber par hasard sur un nom de constante existant déjà dans le programme (ou les autres fichiers inclus). En règle générale, on utilise le nom du fichier d'en-tête comme nom de constante, en le préfixant avec ses initiales pour être certains qu'il n'y aura pas de conflit.

Dans le fichier `tablehash.h`, on voit qu'on défini les prototypes des fonctions implantées dans le fichier `tablehash.c`, de façon à rendre ces fonctions visibles et appelables dans tous les fichiers qui vont inclure le fichier `tablehash.h`.

Il est important de comprendre la différence entre cette notion de visibilité et le fait que les fonctions sont par défaut publiques si on n'utilise pas le mot-clé `static` :

- lors de la phase de compilation, le compilateur doit avoir accès aux prototypes des fonctions appelées, pour pouvoir vérifier si les types des paramètres sont corrects : c'est à ça que sert le .h, mais comme on fait de la compilation séparée, le compilateur n'a pas besoin d'avoir accès à l'implantation des fonctions (dans le .c) pour pouvoir compiler le fichier qui ne fait qu'appeler ces fonctions ;
- par contre, lors de la phase d'édition de liens, l'éditeur fusionne tous les fichiers .o et résout les appels de fonctions inter-modules : à ce moment-là, il va vérifier que la fonction appelée est bien publique et affichera un message d'erreur si ce n'est pas le cas.

Notez bien que c'est le même mot-clé `static` que l'on a déjà vu concernant les déclarations de variables, mais il n'a en fait pas le même sens dans le cas de fonctions : on peut le rapprocher ici de la sémantique du mot-clé `private` en Java.

Compilation avec make

Lorsque l'on découpe un programme en sous-modules, il devient assez vite fastidieux de devoir recompiler chaque module (et sur un gros projet contenant des centaines de modules, cela peut prendre beaucoup de temps).

Un outil existe pour nous simplifier la vie : make, disponible sur tout système Unix digne de ce nom.

make est un outil qui ne recompile que les fichiers qui ont besoin d'être recompilés, c'est à dire ceux qui ont été modifiés depuis la dernière fois qu'on l'a exécuté.

Il se base sur un fichier appelé `Makefile` (qui doit porter précisément ce nom-là et se trouver dans le répertoire dans lequel on lance make) pour connaitre les dépendances entre les différents modules et savoir quoi (et comment) compiler.

Principe général

Imaginons que l'on veuille compiler un programme principal `prog.c` utilisant des tables de hachages (définies dans le module `tablehash.c`) et qui utilisent elles-mêmes des listes chainées (définies dans le module `liste.c`).
On peut alors écrire un `Makefile` de cette forme :

```
prog: prog.o tablehash.o liste.o
    gcc -m64 -o prog prog.o tablehash.o liste.o

prog.o: prog.c
    gcc -c -m64 -std=gnu99 -g -Wall -Wextra -Werror -o prog.o prog.c

tablehash.o: tablehash.c
    gcc -c -m64 -std=gnu99 -g -Wall -Wextra -Werror -o tablehash.o tablehash.c

liste.o: liste.c
    gcc -c -m64 -std=gnu99 -g -Wall -Wextra -Werror -o liste.o liste.c
```

On a définit dans ce fichier `Makefile` 4 règles :

- la première est une règle d'édition de liens : elle dit à make comment produire le binaire `prog` à partir des fichiers objets correspondant ;
- les trois suivantes sont des règles de compilation : elles disent à make comment produire chaque fichier objet correspondant à chacun des fichiers sources.

Une règle a toujours le format suivant :

```
cible: dependances
<TAB>commande
```

- la cible est le fichier à produire, par exemple le binaire ou les fichiers objets ;
- les dépendances sont les fichiers dont dépend la construction du fichier cible ;
- <TAB> représente une tabulation (*hard-tab*) : attention, make ne fonctionnera pas si vous mettez des espaces à la place (les éditeurs intelligents génèrent normalement des tabulations lorsque le fichier s'appelle `Makefile`), mais attention : si vous copiez-collez les exemples de `Makefile` donnés, il faut remplacer les espaces par des tabulations ;
- la commande est celle à exécuter pour construire la cible : il peut y avoir plusieurs lignes de commandes, toutes sous le même format, qui seront exécutées l'une après l'autre.

make se base sur les dates de modification des fichiers pour savoir ce qu'il a besoin de recompiler. Dans notre exemple, si on tape `make prog`, il va :

1. chercher une règle lui expliquant comment générer `prog` : c'est la première du `Makefile` ;
2. pour chaque fichier indiqué dans la liste de dépendance, chercher une règle lui indiquant comment le générer : ce sont les trois lignes suivantes du `Makefile`, et ainsi de suite récursivement ;
3. lorsqu'il arrive sur une dépendance feuille (par exemple, le fichier `prog.c` pour lequel il n'y a pas de règle de construction), make compare simplement la date de dernière modification de cette dépendance et du fichier cible correspondant : si le fichier cible est plus ancien que la dépendance, make lance la commande de génération, sinon il ne fait rien (bien sûr, si un fichier cible n'existe pas encore, la commande de génération est systématiquement lancée).

Par exemple, la première fois que l'on lance `make prog`, aucun fichier n'existe : make va donc exécuter les 4 commandes suivantes, dans cet ordre :

```
gcc -c -m64 -std=gnu99 -g -Wall -Wextra -Werror -o prog.o prog.c
gcc -c -m64 -std=gnu99 -g -Wall -Wextra -Werror -o tablehash.o tablehash.c
gcc -c -m64 -std=gnu99 -g -Wall -Wextra -Werror -o liste.o liste.c
gcc -m64 -o prog prog.o tablehash.o liste.o
```

Si ensuite on modifie le fichier `tablehash.c`, mais aucun autre, et qu'on relance `make prog`, make exécute les 2 commandes suivantes :

```
gcc -c -m64 -std=gnu99 -g -Wall -Wextra -Werror -o tablehash.o tablehash.c
gcc -m64 -o prog prog.o tablehash.o liste.o
```

En effet, make a détecté que les fichiers objets `prog.c` et `liste.c` n'ont pas été modifiés, et que donc les fichiers objets correspondant n'ont pas besoin d'être recompilés.

Utilisation de constantes

Quand on regarde le `Makefile` qu'on a écrit, on remarque tout de suite qu'on a recopié plusieurs fois la même chose, ce qui n'est jamais une bonne idée...
Heureusement, make permet de définir des constantes, ce qui nous permet de factoriser déjà pas mal notre `Makefile` :

```
CC = gcc
CFLAGS = -m64 -std=gnu99 -g -Wall -Wextra -Werror
LD = gcc
LDFLAGS = -m64

prog: prog.o tablehash.o liste.o
    $(LD) $(LDFLAGS) -o prog prog.o tablehash.o liste.o

prog.o: prog.c
    $(CC) -c $(CFLAGS) -o prog.o prog.c

tablehash.o: tablehash.c
    $(CC) -c $(CFLAGS) -o tablehash.o tablehash.c

liste.o: liste.c
    $(CC) -c $(CFLAGS) -o liste.o liste.c
```

Un intérêt évident de l'utilisation de constantes est bien sûr que si on veut changer une paramètre passé à GCC par exemple, on n'aura qu'à le faire une seule fois au début du fichier, et pas dans chaque règle.

Règles implicites

On voit tout de même qu'il nous reste encore pas mal de recopie, notamment en ce qui concerne les règles de compilation des fichiers objets, qui sont sensiblement les mêmes quelques soient les fichiers sources.
make ayant été conçu initialement pour compiler du C, il sait générer des fichiers objets directement, sans qu'on ait besoin de lui expliquer comment faire.
On peut donc écrire plus simplement :

```
CC = gcc
CFLAGS = -m64 -std=gnu99 -g -Wall -Wextra -Werror
LD = gcc
LDFLAGS = -m64

prog: prog.o tablehash.o liste.o

prog.o: prog.c

tablehash.o: tablehash.c

liste.o: liste.c
```

Attention : les noms des constantes choisies ne sont pas anodins : make s'attend à ce que la constante indiquant le nom du compilateur C à utiliser s'appelle `CC`, à ce que celle qui indique les options à passer au compilateur s'appelle `CFLAGS` et ainsi de suite : si vous choisissez d'autres noms, make prendra des valeurs par défaut qui ne vous conviendront surement pas.

On remarque encore une répétition : en effet, les trois fichiers objets sont tous dépendants du fichier source portant le même nom (`prog.o` dépend de `prog.c`, `liste.o` dépend de `liste.c`, etc). make permet d'aller encore plus loin dans l'automatisation, et d'écrire :

```
CC = gcc
CFLAGS = -m64 -std=gnu99 -g -Wall -Wextra -Werror
LD = gcc
LDFLAGS = -m64

prog: prog.o tablehash.o liste.o
```

En effet, make sait qu'un fichier .o se produit en général simplement en compilant le fichier .c correspondant, il n'a donc pas besoin qu'on lui dise.

Utilisation courante de make

Il est souvent pratique d'ajouter aux `Makefile` une règle de nettoyage, qui efface tous les fichiers binaires produits et ne conserve que les sources (par exemple, pour préparer son archive à rendre sur Teide !). On introduit donc fréquemment une règle (traditionnellement appelée `clean`) comme suis :

```
BIN = prog
OBJS = prog.o liste.o tablehash.o

CC = gcc
CFLAGS = -m64 -std=gnu99 -g -Wall -Wextra -Werror
LD = gcc
LDFLAGS = -m64

$(BIN): $(OBJS)

.PHONY: clean
clean:
    $(RM) $(BIN) $(OBJS)
```

Notez qu'on a rajouté des constantes `BIN` et `OBJS` pour factoriser en début du `Makefile` les noms des binaires et des modules, ce qui permettra de les changer plus facilement si nécessaire. La constante `RM` est prédéfinie par make : elle est remplacée par `rm -f` (attention à ne pas écrire n'importe-quoi derrière, les fichiers seront effacés sans préavis !).
La directive `.PHONY` indique à make que `clean` est une cible virtuelle, qui n'est pas un fichier : c'est important car si par hasard votre répertoire contenait un fichier appelé `clean`, make se baserait sur sa date de dernière modification pour savoir s'il doit exécuter ou pas la commande d'effaçage : comme on veut qu'un appel à `make clean` nettoie systématiquement le répertoire, la directive `.PHONY` désactive ce comportement.
Enfin, il est traditionnel d'introduire une cible virtuelle `all` avant toutes les autres règles, afin qu'un appel à `make all` (ou simplement `make` sans paramètre) recompile tout le programme. On finit donc par obtenir le `Makefile` suivant :

```
BIN = prog
OBJS = prog.o liste.o tablehash.o

CC = gcc
CFLAGS = -m64 -std=gnu99 -g -Wall -Wextra -Werror
LD = gcc
LDFLAGS = -m64

.PHONY: all
all: $(BIN)
```

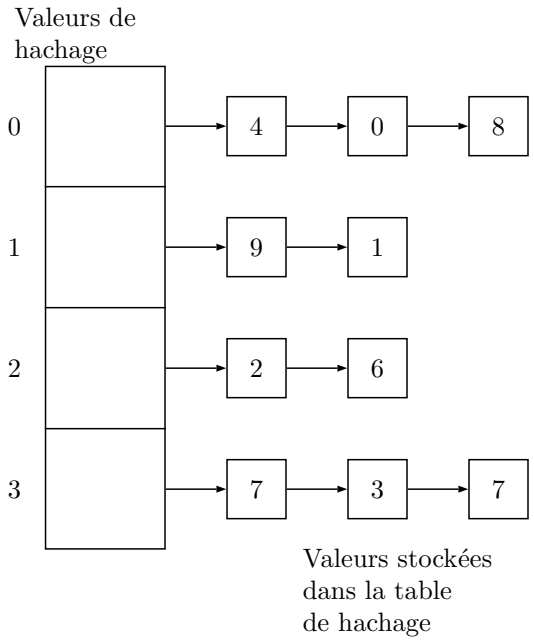
```
$(BIN): $(OBJS)

.PHONY: clean
clean:
	$(RM) $(BIN) $(OBJS)
```

Pour aller plus loin...

La documentation en ligne de make détaille la myriade de directives et de fonctions que cette commande supporte.
Il n'est pas question de toutes les détailler, mais vous pouvez par exemple lire le paragraphe sur les variables automatiques, qui sont souvent utiles dans des `Makefile` compliqués.

Exercice

Récupérez l'archive contenant les sources.
On va travailler sur l'implantation d'un module gérant des tables de hachage. On rappelle qu'une table de hachage est une structure de données permettant de combiner un accès rapide aux éléments qu'elle contient (pas aussi rapidement qu'un tableau, mais mieux qu'une simple liste chainée en moyenne) avec une capacité non-bornée (à part par la quantité de mémoire disponible pour le programme bien sûr).
Une table de hachage se présente typiquement sous cette forme :



C'est à dire plus précisément un tableau de taille fixe, dont chaque entrée contient une liste chainée contenant des valeurs entrées dans la table de hachage.
Les valeurs sont réparties entre les différentes listes chainées en fonction de leur valeur de hachage : une valeur de hachage est la valeur obtenue en application une fonction de transformation (appelée fonction de hachage) à la valeur au moment où on l'insère dans la table.
Dans l'exemple fourni, la fonction de hachage est visiblement simplement la fonction module 4 : on répartit donc les chiffres entrés en fonction du reste de leur division entière par 4 : pour les valeurs 4, ou et 0, le reste vaut 0 donc ces valeurs sont insérées dans la première liste ; pour les valeurs 9 et 1, le reste vaut 1 donc elles sont insérées dans la deuxième liste, et ainsi de suite.
La qualité de la fonction de hachage a une influence déterminante sur les performances des accès à la table : on cherche en général à équilibrer les longueurs des différents listes (pour garantir que les accès aux différents éléments prendront des temps similaires) : une mauvaise fonction de hachage qui mettrait beaucoup de valeurs dans une liste et très peu dans les autres ramènerait le temps d'accès aux éléments

de la grande liste à ce qu'on obtiendrait si tous les éléments étaient stockés dans une simple liste chainée. La fonction modulo 4 utilisée ici est un exemple typique de mauvaise fonction de hachage : en effet, si (par hasard) toutes les valeurs insérées sont des multiples de 4, elles se retrouveront toutes dans la première liste.

Implantation d'un module de listes simplement chainées

On va donc commencer par implanter un module capable de gérer des listes simplement chainées de valeurs entières naturelles sur 8 bits. On vous fourni le fichier d'en-tête `liste.h` définissant la liste des fonctions à implanter (vous avez bien sûr le droit d'en rajouter d'autres si nécessaire) et vous devez compléter le fichier `liste.c` fourni en implantant le corps de fonctions. On impose la structure suivante pour les types des cellules et des listes :

```
// Une cellule contient simplement :
//   - la valeur a stocker
//   - un pointeur vers la cellule suivante
struct cell_t {
    uint8_t val;
    struct cell_t *suiv;
};

// Une liste est simplement composée d'une cellule
//   sentinelle en tete (attention, ce n'est pas un pointeur !)
// On rappelle qu'une sentinelle est une cellule sans valeur,
//   qui sert a simplifier le code de certaines fonctions de
//   manipulation : les valeurs significatives de la liste
//   commencent donc a partir de la cellule suivant la
//   sentinelle de tete
struct liste_t {
    struct cell_t tete;
};
```

Les fonctions à implanter (au minimum) sont :

- `struct liste_t *nouv_liste(void)` : cette fonction crée une nouvelle liste et renvoie un pointeur sur cette liste à la fonction appelante (plus clairement, la fonction doit allouer dynamiquement un objet de type `liste_t` et initialiser son champ `tete` avec des valeurs appropriées) ;
- `bool est_vide_liste(struct liste_t *liste)` : cette fonction prend en paramètre un pointeur vers une liste et renvoie vrai ssi cette liste ne contient aucun élément significatif (c'est à dire autre que la sentinelle de tête) ;
- `void inserer_tete_liste(uint8_t val, struct liste_t *liste)` : cette fonction prend en paramètre une valeur à insérer dans une liste sur laquelle on passe un pointeur : l'insertion doit se faire en tête de liste comme l'indique le nom de la fonction ;
- `bool supprimer_val_liste(uint8_t val, struct liste_t *liste)` : cette fonction prend en paramètre une valeur qu'on veut supprimer de la liste vers laquelle on passe un pointeur : si la valeur est absente, la fonction ne fait rien (pas d'erreur) mais renvoie `false`, si la valeur est présente plusieurs fois dans la liste, on supprime la première occurence, dès qu'une suppression a eu lieu, on renvoie `true` ;
- `void afficher_liste(struct liste_t *liste)` : cette fonction affiche le contenu (c'est à dire les valeurs significatives) de la liste vers laquelle on passe un pointeur en paramètre ;
- `void detruire_liste(struct liste_t **liste)` : cette fonction détruit la liste passée en paramètre (c'est à dire qu'elle désalloue toutes les cellules contenues dans la liste, ainsi que l'objet de type `liste_t` lui-même) : notez qu'on passe un pointeur vers un pointeur vers une liste, car on doit forcer à la fin de la fonction le pointeur passé à `NULL` (par sécurité).

Vous remarquerez que le code fourni ne contient absolument aucun commentaire, ce qui est Très Mal ! Vous ajouterez donc tous les commentaires pertinents (notamment, les préconditions des fonctions implantées). N'hésitez pas à utiliser des assertions pour vérifier ces pré-conditions ainsi que les retours de la fonction `malloc` (utile pour détecter un problème de pointeur nul). On ne fournit pas non plus de programme de

test de ce module, à vous de l'implanter pour vérifier que vos fonctions sont correctes (n'oubliez pas que Valgrind est votre ami dès que vous faites de l'allocation dynamique de mémoire...).
Question couramment posée : « pourquoi passe-t'on des pointeurs vers la liste, plutôt que la liste elle-même » (c'est à dire, pourquoi nos fonctions prennent en paramètres des `struct liste_t *` et pas des `struct liste_t`). Réponse : lorsque vous passez une structure en paramètre d'une fonction, le compilateur passe chaque champ de la structure dans un paramètre différent, alors qu'un pointeur est une simple valeur sur 64 bits. Dans le cas des listes implantées ici, ça ne changerait rien vu que la structure liste n'a qu'un champ, mais dans le cas général, il est plus efficace de passer un pointeur vers une structure compliquée plutôt que la structure elle-même.

Implantation du module table de hachage

On implante ensuite le module gérant les tables de hachage. Là encore, on fournit le .h correspondant, et on vous impose le type des données manipulées :

```
// taille du tableau correspondant aux valeurs de hachage
// cette taille doit bien sur etre coherente avec la fonction de hachage !
#define TAILLE_TABLE 4

// Une table de hachage est composée :
//   - d'un entier naturel qui contient le nombre total d'éléments dans la table
//   - d'un tableau de pointeurs vers les listes chainees contenant les elements
struct tablehash_t {
    uint64_t nbr_elem;
    struct liste_t *table[TAILLE_TABLE];
};
```

On vous fournit aussi la fonction de hachage (très naïve) qu'on utilisera dans cet exercice :

```
static int64_t hash(uint8_t x)
{
    return x % TAILLE_TABLE;
}
```

Vous devez implanter au moins les fonctions suivantes :

- `struct tablehash_t *nouv_tablehash(void)` : cette fonction alloue dynamiquement une nouvelle table de hachage et renvoie un pointeur dessus (là-encore, pensez bien à initialiser les champs de la table avec des valeurs pertinentes) ;
- `bool est_vide_tablehash(struct tablehash_t *th)` : cette fonction renvoie `true` ssi la table de hachage est vide (c'est à dire ne contient aucune valeur insérée) ;
- `void afficher_tablehash(struct tablehash_t *th)` : cette fonction affiche le contenu de la table de hachage dont un pointeur est passé en paramètre ;
- `void inserer_val_tablehash(uint8_t val, struct tablehash_t *th)` : cette fonction insère la valeur `val` à sa place dans la table de hachage ;
- `void supprimer_val_tablehash(uint8_t val, struct tablehash_t *th)` : cette fonction détruit la cellule contenant la première occurence de la valeur `val` dans la table de hachage (si la valeur est absente de la table, la fonction ne fait rien et n'affiche pas de message d'erreur) ;
- `void detruire_tablehash(struct tablehash_t **th)` : cette fonction désalloue complètement la table de hachage (y compris bien sûr toutes ses sous-listes !).

Vous devez de nouveau ajouter tous les commentaires pertinents, et implanter vous-même votre programme de test pour vérifier que vos fonctions sont correctes. Est-il besoin de préciser que vous devez bien sûr utiliser autant que possible les fonctions de votre module de gestion de listes, sans ré-implanter ce que vous avez déjà écrit ?

Pour aller plus loin

Si vous avez fini en avance, on vous propose une extension assez courante : rendre votre table de hachage redimensionnable.

En effet, au fur et à mesure que l'on va insérer des valeurs dans la table, les listes chainées vont grandir, ce qui va rendre l'accès aux éléments insérés en fin de liste de plus en plus couteux.
Pour palier à cet inconvénient, vous pouvez implanter une fonction qui, lorsqu'au moins une des sous-listes dépasse une certaine longueur palier (disons par exemple 5) :

- change la taille du tableau (par exemple, pour passer d'un tableau de taille 4 à un tableau de taille 8, puis 16, puis 32, etc) ;
- change la fonction de hachage (il faudra donc sûrement modifier la constante `TAILLE_TABLE` pour la remplacer par une variable globale) ;
- re-répartit les cellules déjà présentes dans la table pour que leur placement correspondante à la nouvelle fonction de hachage.

Pour changer la taille du tableau, la fonction `realloc` peut être utile, ou bien vous pouvez simplement créer une nouvelle table et y chainer les cellules existantes (par contre, vous ne devez pas détruire et re-créer les cellules elles-mêmes, il faut procéder par modification de chainage).