13 Programmation modulaire

13.1 Retour sur la compilation

13.1.1 Externaliser une fonction

Jusqu'ici, nous avons essentiellement travaillé sur des travaux pratiques pour lesquels écrire le code dans une seul fichier semblait faire l'affaire. Cependant pour un projet ou quand le code commence à devenir plus conséquent et plus complexe, se limiter à un fichier peut vite devenir douloureux.

Prenons l'exemple du code suivant :

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

void maFonction() {
        printf("Ma Fonction\n");
}

int main() {
  maFonction();
  exit(EXIT_SUCCESS);
}
```

Ce code définit une fonction puis l'appel. Imaginons que nous ne voudrions pas définir cette fonction dans ce fichier, mais dans un autre. Ceci est possible :

- Nous avons besoin de la déclaration de la fonction pour l'appeler dans notre code.
- Sa définition peut être sauvegardée dans un autre fichier.

main.c

```c
#include <stdlib.h>

extern void maFonction();

int main() {
  maFonction();
  exit(EXIT_SUCCESS);
}
```

mes_fonctions.c

```c
#include <stdio.h>

void maFonction() {
  printf("Ma Fonction\n");
}
```

Pour compiler une telle configuration, il faut que les deux fichiers soient compilés et liés vers le même exécutable. Ceci se fait en ajoutant `mes\_fonctions.c` dans la ligne de compilation :

```
gcc -o executable main.c mes_fonctions.c
```

`maFonction` a maintenant été externalisée dans un autre fichier. Le mot-clé `extern` est facultatif pour une fonction. Cependant, il est nécessaire si l'on souhaite utiliser une variable instanciée dans un autre fichier source :

main.c

```c
#include <stdlib.h>

extern int variable;

extern void maFonction(int);

int main() {
  maFonction(variable);
  exit(EXIT_SUCCESS);
}
```

mes_fonctions.c

```c
#include <stdio.h>

int variable = 42;

void maFonction(int v) {
  printf("Ma Fonction %d\n", v);
}
```

13.1.2 Compilation séparée

Revenons sur la compilation. Rappelez vous, nous avions parlé de compilateur et d'éditeur des liens :

- **La compilation** c'est la transformation du code source en langage C en langage utilisable par la machine. Ceci produira des fichiers "Objets" avec pour extension `.o`.

- **L'édition des liens** c'est l'assemblage de ces fichiers `.o` en une application (exécutable) et le raccordement des définitions de chaque élément à sa déclaration et son appel.

S'il reste une déclaration non résolue (absence du fichier contenant la définition d'une fonction par exemple), le compilateur vous l'indiquera à l'édition des liens :

```
# gcc -o executable main.c
/tmp/cc5vxQgu.o : Dans la fonction « main » :
main.c:(.text+0xa) : référence indéfinie vers « maFonction »
collect2: error: ld returned 1 exit status
```

Nous avons vu ici une version simplifiée de la commande de compilation : tout est compilé puis linké. Cependant ceci veut dire que pour la modification d'un fichier, l'ensemble est recompilé. Il est donc possible de compiler les fichiers uns à uns avec l'option `-c`. Celle-ci fabrique un fichier `.o` associé à chaque fichier `.c` fourni. Ensuite on procède à l'édition des liens en fournissant les fichiers `.o` et bibliothèques utilisées.

```
# gcc -c mes_fonctions.c
# gcc -c main.c
# gcc -o executable main.o mes_fonctions.o
```

Si vous souhaitez visualiser la table des symboles d'un fichier `.o`, vous pouvez utiliser la commande `readelf` :

```
# readelf -s main.o

La table de symboles « .symtab » contient 13 entrées :
   Num:    Valeur          Tail Type    Lien   Vis      Ndx Nom
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS main.c
...
     8: 0000000000000000    27 FUNC    GLOBAL DEFAULT    1 main
     9: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND
     ↪  variable
    10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND
    ↪  _GLOBAL_OFFSET_TABLE_
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND
    ↪  maFonction
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND exit
```

Nous utilisons aussi `printf`, une fonction définie ailleurs, sans l'avoir déclarée, non ? Vos fichiers ne sont en réalité pas si vides et vous l'avez fait avec votre

`#include <stdio.h>`. C'est une directive préprocesseur, voyons comment ceci fonctionne.

13.2 Directives préprocesseur

Un programme qui affiche "Hello ESGI!", ça tient en 10 lignes, regardez :

```c
#include <stdio.h>

int main() {
  printf("Hello ESGI !\n");
  return 0;
}
```

Essayez maintenant la commande suivante :

```
gcc -o main.i -E -P main.c
```

Ceci ressemble plutôt à ça :

```c
typedef long unsigned int size_t;
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
... /* environ 200 lignes */

int main() {
 printf("Hello ESGI !\n");
 return 0;
}
```

Cette commande permet de résoudre les directives préprocesseurs. En effet, avant d'être compilé, un code source en langage C est pré-traité pour qu'il ne reste réellement que des instructions en langage C. Ces directives préprocesseur permettent de gagner en lisibilité ou maintenabilité lorsque nous avons besoin que des informations soient inscrites en dur dans du code C. Dans notre cas, ceci nous permet de déclarer toutes les fonctionnalités que nous utiliserons de la bibliothèques `stdio.h` depuis son entête de déclaration. Notez que simple déclarer `printf` fonctionne sans erreur ni warning :

```c
extern int printf(const char *, ...);

int main() {
        printf("Hello ESGI !\n");
        return 0;
}
```

13.2.1 include

La directive préprocesseur `include` permet de recopier le contenu d'un fichier à l'emplacement du fichier C courant où elle est appelée. C'est un outil puissant, mais à utiliser avec parcimonie. En général, nous les utiliserons pour inclure des fichiers d'entête dans notre code :

- `#include <entete.h>` : permet l'inclusion d'une bibliothèque présente ou installée sur la machine.
- `#include "entete.h"` : est réservé à l'inclusion relative de vos propres fichiers d'entête.

Ceci permet par exemple de définir sa propre bibliothèque de de fonctionnalités :

main.c

```c
#include <stdlib.h>
#include "mes_fonctions.h"

int main() {
  maFonction(variable);
  exit(EXIT_SUCCESS);
}
```

mes_fonctions.h

```
extern int variable;

extern void maFonction(int);
```

mes_fonctions.c

```
#include <stdio.h>

int variable = 42;

void maFonction(int v) {
  printf("Ma Fonction %d\n", v);
}
```

Nous verrons dans la suite comment protéger cette bibliothèques contre des problèmes de doublons dans l'inclusion et en faire ainsi un module.

13.2.2 define

Nous avions déjà croisé une utilisation de la directive préprocesseur `define` pour la création de constantes. Cette directive fera que pour un nom donné, celui-ci sera remplacé par le code associé.

Expressions constantes

Ceci permet par exemple de déclarer des expression constantes :

```
#include <stdio.h>
#include <stdlib.h>
#define TAILLE 10

int main() {
  int i;
  for(i = 0; i < TAILLE; ++i) {
```

```
    printf("%d\n", i);
  }
  exit(EXIT_SUCCESS);
}
```

En général, on les introduit plutôt en début de fichier, mais ces directives peuvent être placées où souhaité dans le code. Leur définition peut être annulée avec la directive préprocesseur `undef` et elles peuvent être redéfinies :

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

int main() {
#define NOMBRE 42
  printf("define NOMBRE %d\n", NOMBRE);
#undef NOMBRE
#define NOMBRE 13.37
  printf("define NOMBRE %g\n", NOMBRE);
#undef NOMBRE
#define NOMBRE "1234"
  printf("define NOMBRE %s ?\n", NOMBRE);
  exit(EXIT_SUCCESS);
}
```

La directive `define` peut être utilisée pour remplacer un nom de fonction ou même du code :

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

#define print puts

int main() {
  print("Hello ESGI");
  /* Python de langage C ! */
  exit(EXIT_SUCCESS);
```

```
}
```

Pour revenir à la ligne dans le contenu associé à un `#define`, il faut utiliser un anti-slash :

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

#define INSTRUCTIONS print("Hello ESGI\n"); \
                     exit(EXIT_SUCCESS);

int main() {
  INSTRUCTIONS
}
```

Macros

Il est possible de définir des "Macros" à l'aide du `#define`. Ces macros vont écrire en dur du code en fonction de paramètres données. Ceci peut être utilisé par exemple pour remplacer des fonctions courtes garder une généricité sur les arguments fournis.

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

#define min(a, b) (a < b) ? a : b

int main() {
  int a = 42, b = 1337;
  printf("min(%d, %d) = %d\n", a, b, min(a, b));
  float c = 13.37, d = 42.;
  printf("min(%g, %g) = %g\n", c, d, min(c, d));
  exit(EXIT_SUCCESS);
}
```

Cependant, ceci reste à utiliser avec parcimonie. En effet, la macro va récrire à l'identique la partie de code fournie en argument, là où une fonction travaillerait

avec la valeur de retour fournie. Il est donc conseillé de mettre les arguments entre parenthèses dans le code de la macro et d'être rigoureux avec les arguments dont la duplication de l'expression pourrait être problématique :

```c
int fonction_min(int a, int b) {
  return (a < b) ? a : b;
}

#define macro_min(a, b) ((a) < (b)) ? (a) : (b)

int main() {
  int v, a, b;
  v = fonction_min(a = getchar() - '0', b = getchar() - '0');
  printf("fonction_min(%d, %d) = %d\n", a, b, v);
  v = macro_min(a = getchar() - '0', b = getchar() - '0');
  printf("macro_min(%d, %d) = %d\n", a, b, v);
  exit(EXIT_SUCCESS);
}
```

Ce code produit en réalité le code suivant :

```c
int fonction_min(int a, int b) {
  return (a < b) ? a : b;
}

int main() {
  int v, a, b;
  v = fonction_min(a = getchar() - '0', b = getchar() - '0');
  printf("fonction_min(%d, %d) = %d\n", a, b, v);
  v = ((a = getchar() - '0') < (b = getchar() - '0')) ? (a =
  ↪  getchar() - '0') : (b = getchar() - '0');
  printf("macro_min(%d, %d) = %d\n", a, b, v);
  exit(0);
}
```

D'où une exécution potentiellement erronée :

```
24246
fonction_min(4, 2) = 2
macro_min(6, 4) = 6
```

Il est possible de transformer une expression fournie à une macro en une chaîne de caractère en précédant le nom de l'expression par un dièse **#** dans le code associé à la macro :

```c
#define FAIRE_CALCUL(exp) printf("%s = %d\n", #exp, exp)

int main() {
  int valeur = 4;
  FAIRE_CALCUL(valeur * valeur + 2 * valeur + 1);
  exit(EXIT_SUCCESS);
}
```

Ce qui produit l'exécution suivante :

```
valeur * valeur + 2 * valeur + 1 = 25
```

Pour concaténer l'expression fournie à une macro avec du code : par exemple dans le nom d'une fonction. Il est possible d'appeler d'appeler l'opérateur **##** dans le code associé à la macro :

```c
#define macro_abs(a) (((a) < 0) ? -(a) : (a))
#define macro_carre(a) ((a) * (a))

#define call(f, x) printf("%s(%d) = %d\n", #f, x, macro_##f(x))

int main() {
  call(abs, -5);
  call(carre, -5);
  exit(EXIT_SUCCESS);
}
```

Ce qui produit l'exécution suivante :

```
abs(-5) = 5
carre(-5) = 25
```

13.2.3 conditionnement

Il est possible de conditionner du code sous condition. Par exemple, il peut être utile d'avoir un mode verbeux pour travailler sur votre code et un mode moins verbeux pour le fournir à un utilisateur.

Par exemple dans le code suivant, on peut adapter le code en fonction du fait que certaines `#define` existent ou non :

```c
#define DEBUG

int main() {
#if defined VERBOSE
  fprintf(stderr, "Entrée dans le main\n");
#endif
#if defined VERBOSE
  printf("42, la réponse à la vie !\n");
#elif defined DEBUG
  printf("42, vous savez pourquoi.\n");
#else
  printf("42 !\n");
#endif
#if defined VERBOSE
  fprintf(stderr, "Sortie du main\n");
#endif
  exit(EXIT_SUCCESS);
}
```

Ceci produit le code suivant :

```c
int main() {
  printf("42, vous savez pourquoi.\n");
```

```
  exit(0);
}
```

Il est possible de réaliser un `#define` depuis la commande de compilation avec l'option `-D` :

```
gcc -o main.i -E -P main.c -DVERBOSE
```

Ce qui produit le code suivant :

```
int main() {
  fprintf(stderr, "Entrée dans le main\n");
  printf("42, la réponse à la vie !\n");
  fprintf(stderr, "Sortie du main\n");
  exit(0);
}
```

Il est possible de raccourcir l'écriture des conditions vérifiant la définition d'un `#define` :

```
#if defined NOM
#ifdef NOM

#if !defined NOM
#ifndef NOM
```

13.2.4 Module

Nous avons vu précédemment qu'il peut être intéressant de répartir ses fonctionnalités dans d'autres fichiers. Plus précisément nous découperons notre code en **modules**. Un module est un ensemble de fonctionnalités documentées pour lequel on fourni un fichier d'entête `.h` à l'utilisateur (un autre programmeur ou vous-même) et pour lequel vous avez implémenté les fonctionnalités associées dans un fichier `.c` du même nom.

Exemple d'organisation d'un module :

module.h

```c
#ifndef DEF_HEADER_MODULE
#define DEF_HEADER_MODULE
/* Protection du module */

/**
  Documentation du module et auteurs
*/

/* Macros publiques */
#define abs(x) (((x) < 0) ? -(x) : (x))

/* Types publiques du module */
typedef struct Point Point;
struct Point {
  float x;
  float y;
};

/* Variables publiques du module */
extern Point origine;

/* Fonctionnalités publiques du module */

/* Affiche un point dans la sortie standard */
extern void Point_afficher(const Point * point);

#endif
```

module.c

```c
#include "module.h"
/* Inclusion des déclarations du module */

#include <stdio.h>
/* Autres inclusions */

/* Définition des variables globales relatives au module
 ↪  */
Point origine = {0, 0};

/* Fonctionnalité privée au module par le mot-clé static
 ↪  */
static void Point_print(FILE * flow, const Point * point)
 ↪  {
  if(! point) {
    fprintf(flow, "(nil)");
  }
  fprintf(flow, "(%g, %g)", point->x, point->y);
}

/* Définition des fonctionnalités annoncées par l'entête
 ↪  */
void Point_afficher(const Point * point) {
  Point_print(stdout, point);
}
```

13.3 Makefiles

13.3.1 Concept

Avec l'augmentation du nom de modules, l'ajout de bibliothèques et autres la compilation de votre projet va se complexifier. Pour ceci, il peut être intéressant d'automatiser sa compilation. À noter qu'un avantage offert par la programmation modulaire est que l'on peut travailler sur chaque module indépendamment des interdépendances entre les modules. Et donc avoir une compilation potentiellement plus rapide que recompiler l'intégralité du projet.

Le moyen proposé pour compiler un projet en langage C est un **Makefile**. Un Makefile est un fichier qui donne le schéma à suivre pour compiler le code. Celui-ci demande d'être créé à l'emplacement d'où vous souhaitez que l'utilisateur compile votre projet. Il n'aura ensuite besoin que d'utiliser la commande `make`.

Makefile

```
executable :
        gcc -o executable *.c
```

```
make
```

13.3.2 Possibilités

Un Makefile s'organise comme une liste de cibles, dépendances associées et commandes à exécuter. Chaque commande est précédée d'une **tabulation**.

```
cible: dépendances
        commande
        ...
        commande
```

Les dépendances vont permettre de relancer les commandes pour construire la cible si elles ont été modifiées.

La proposition de compilation est la suivante :

- Compilation séparée des fichiers `.c` en fichiers `.o` pour chaque module.
- Fichier source `.c` du module et fichiers `.h` inclus dans le module en dépendances.
- Linkage de tous les fichiers `.o` pour création de l'exécutable.
- Une commande pour nettoyer le répertoire de la compilation.

Ceci pourrait donner le Makefile suivant :

```
executable : main.o module.o
	gcc -o executable main.o module.o

main.o : main.c module.h
	gcc -c main.c

module.o : module.c module.h
	gcc -c module.c

clean :
	rm -rf *.o
```

La commande de nettoyage du dossier peut s'appeler de la manière suivante :

```
make clean
```

Cependant ce Makefile peut être assez redondant, il est possible d'introduire les symboles suivants :

Symbole	Correspondance
`$@`	Cible
`$^`	Toutes les dépendances
`$<`	Première dépendance

Ceci permet la transformation du Makefile en le suivant :

```
executable : main.o module.o
	gcc -o $@ $^
```

```
main.o : main.c module.h
	gcc -c $<

module.o : module.c module.h
	gcc -c $<
```

Pour aller plus loin, il est aussi possible d'utiliser une règle générique en utilisant un `%`. Celui-ci sera automatiquement remplacé de manière à construire les cibles manquantes correspondant au schéma donné. Cependant, il sera nécessaire d'ajouter les dépendances pour qu'elles soient prises en compte :

```
executable : main.o module.o
	gcc -o $@ $^

main.o : main.c module.h
module.o : module.c module.h

%.o : %.c
	gcc -c $<
```

Il est aussi possible de définir des variables pour votre Makefile. Leur définition prend la syntaxe `VARIABLE=VALEUR` et leur appel `$(VARIABLE)`. Elles sont utiles pour écrire à un endroit :

- le compilateur : `CC=gcc` (ou par exemple `CC=clang`).
- les drapeaux de compilation et optimisations dans `CFLAGS` (les optimisations peuvent être `-O1`, `-O2` et éventuellement `-O3`).
- les bibliothèques dans `CLIBS`.
- le nom de l'exécutable.

```
CC= gcc
CFLAGS= -O2 -Wall -Wextra -Werror -ansi
CLIBS= -lm
EXE= executable

$(EXE) : main.o module.o
	$(CC) $(CFLAGS) -o $@ $^ $(CLIBS)
```

```
main.o : main.c module.h
module.o : module.c module.h

%.o : %.c
        $(CC) $(CFLAGS) -c $<
```

En langage C, on structueras général un projet en séparant les différents fichiers dans des répertoires :

- `include` contient vos fichiers d'entêtes `.h`.
- `src` contient vos fichiers sources `.c`.
- `obj` permet de sauvegarder les fichiers compilés `.o`.

L'exécutable sera ainsi généré à la racine du projet avec les fichiers pertinents : `README.md`, Makefile et autres. Ceci donne le Makefile suivant :

```
CC= gcc
CFLAGS= -O2 -Wall -Wextra -Werror -ansi
CLIBS= -lm
EXE= executable
OBJ= obj/
SRC= src/
INCL= include/

$(EXE) : $(OBJ)main.o $(OBJ)module.o
        $(CC) $(CFLAGS) -o $@ $^ $(CLIBS)

$(OBJ)main.o : $(SRC)main.c $(INCL)module.h
$(OBJ)module.o : $(SRC)module.c $(INCL)module.h

$(OBJ)%.o : $(SRC)%.c
        $(CC) $(CFLAGS) -o $@ -c $<

clean :
        rm -rf $(OBJ)*.o
        rm -rf $(EXE)
```

Dans le cas où un temps de compilation potentiellement un peu plus important ne vous serait pas dérangeant, il est possible de gagner du temps dans la construction du Makefile en utilisant la fonction `wildcard`. Ceci permet d'obtenir une liste d'éléments suivant un schéma donné. En l'occurrence, dans notre cas obtenir les fichiers sources et entêtes. Et pour la génération de la liste des fichiers objets la fonction `patsubst` qui fera la substitution des `.c` en `.o` dans la listes des sources obtenue. Ceci fait que le Makefile suivant ne vous demandera pas de rajouter une dépendance à chaque ajout de module :

```
CC= gcc
CFLAGS= -O2 -Wall -Wextra -Werror -ansi
CLIBS= -lm
EXE= executable
OBJ= obj/
SRC= src/
INCL= include/
FILEC:= $(wildcard $(SRC)*.c)
FILEH:= $(wildcard $(INCL)*.h)
FILEO:= $(patsubst $(SRC)%.c,$(OBJ)%.o,$(FILEC))

$(EXE) : $(FILEO)
        $(CC) $(CFLAGS) -o $@ $^ $(CLIBS)

$(OBJ)main.o : $(SRC)main.c $(FILEH)
        $(CC) $(CFLAGS) -o $@ -c $<

$(OBJ)%.o : $(SRC)%.c $(INCL)%.h
        $(CC) $(CFLAGS) -o $@ -c $<

clean :
        rm -rf $(OBJ)*.o
        rm -rf $(EXE)
```

13.4 Résumé

Les directives préprocesseur sont des instructions traitées avant la compilation. Celles-cis permettent notamment :

```
#include <lib.h> /* copie l'entête d'une bibliothèque lib */
#include "module.h" /* copie l'entête de mon fichier module.h */
#define NOM EXPRESSION /* remplacera NOM dans le code par
↪  EXPRESSION */
#undef NOM /* supprime la définition de NOM dans le code qui suit
↪  */
```

La directive `define` peut prendre des paramètres dont les expression données lors de l'appel remplacera leur occurrence dans l'expression donnée dans la définition.

```
#define MACRO(exp) exp \ /* remplace exp par le code donné en
↪  argument et '\' permet de continuer la macro à la ligne */
 #exp \ /* fabrique une chaîne de caractères de exp */
 ##exp## \ /* concatène exp avec le code adjacent dans la macro
 ↪  */
```

Il est possible d'utiliser des structures conditionnelles avec les directives préprocesseur :

```
#if CONDITION1
/* instructions1 */
#elif CONDITION2
/* instructions2 */
#else
/* instructions3 */
#endif
```

Le mot clé `defined` permet d'indiquer si une `define` a nommé ce nom. Celui-ci peut être abrégé dans les structures conditionnelles :

```
#if defined NOM
#ifdef NOM
```

```
#if ! defined NOM
#ifndef NOM
```

On peut diviser son code en modules : un fichier `.c` (implémentation des fonctionnalités) et `.h` (déclarations à inclure pour l'utiliser dans un autre fichier source) du même nom. Pour compiler un code découpé en modules, on utilise un `Makefile` :

```
cible: dépendances
        commandes
```

```
$@ # Cible
$^ # Toutes les dépendances
$< # Première dépendance
VARIABLE= valeur # création d'une variable
$(VARIABLE) # appel d'une variable
```

13.5 Entraînement

Exercice noté 50 (⋆⋆ Mise en place d'un Makefile).

Découper le code suivant en modules et écrire un Makefile permettant sa compilation modulaire :

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

typedef struct Point Point;
struct Point {
  double x;
  double y;
};

Point Point_creer(double x, double y) {
  Point p = {x, y};
  return p;
}

void Point_afficher(const Point * p) {
  printf("(%g, %g)", p->x, p->y);
}

double Point_distance(const Point * first, const Point * second) {
  return sqrt(pow(second->x - first->x, 2) + pow(second->y -
  ↪  first->y, 2));
}

typedef struct Triangle Triangle;
struct Triangle {
  Point first;
  Point second;
  Point third;
};

Triangle Triangle_creer(Point first, Point second, Point third) {
  Triangle triangle = {first, second, third};
  return triangle;
```

```c
}

void Triangle_afficher(const Triangle * triangle) {
  printf("{Triangle : ");
  Point_afficher(&(triangle->first));
  printf(", ");
  Point_afficher(&(triangle->second));
  printf(", ");
  Point_afficher(&(triangle->third));
  printf("}");
}

double Triangle_perimetre(const Triangle * triangle) {
  return
    Point_distance(&(triangle->first), &(triangle->second))
    + Point_distance(&(triangle->second), &(triangle->third))
    + Point_distance(&(triangle->third), &(triangle->first));
}

int main() {
  Triangle triangle = Triangle_creer(
    Point_creer(0, 0),
    Point_creer(4, 0),
    Point_creer(0, 3)
  );
  Triangle_afficher(&triangle);
  printf("\nPerimetre : %g\n", Triangle_perimetre(&triangle));
  exit(EXIT_SUCCESS);
}
```

Exercice noté 51 (⋆⋆⋆⋆ Profilage d'un code).

Écrire un fichier d'entête `profile.h` dont les macros permettent un profilage d'un code qui les utiliseraient :

- `DO_PROFILE` active l'aspect verbeux du profilage (inactif sinon).
- `START_FUNCTION` prend le type de retour, le nom et les arguments de la fonction.
- `END_FUNCTION` prend le retour de la fonction.

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

#define DO_PROFILE
#include "profile.h"

START_FUNCTION(int, f, int x)
END_FUNCTION(x * x)

START_FUNCTION(int, main)
  int a = 42;
  a = f(a);
  printf("a = %d\n", a);
  exit(EXIT_SUCCESS);
END_FUNCTION(0)
```

Le code ci-dessus se comporte comme le code ci-dessous :

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

int f(int x) {
  return x * x;
}

int main() {
  int a = 42;
  a = f(a);
  printf("a = %d\n", a);
  exit(EXIT_SUCCESS);
  return 0;
}
```

Si la définition préprocesseur `DO_PROFILE` n'est pas définie, le code donne la sortie suivante :

```
a = 1764
```

Si `DO_PROFILE` est définie, le code se montre plus verbeux et affiche les informations additionnelles suivantes dans la sortie standard d'erreur :

```
# Definition of int function main() in file "main.c" at line 10
< Starting function main :
# Definition of int function f(int x) in file "main.c" at line 7
< Starting function f :
> Ending function f with return x * x
a = 1764
> Exit in function main at line 14 with value EXIT_SUCCESS
```

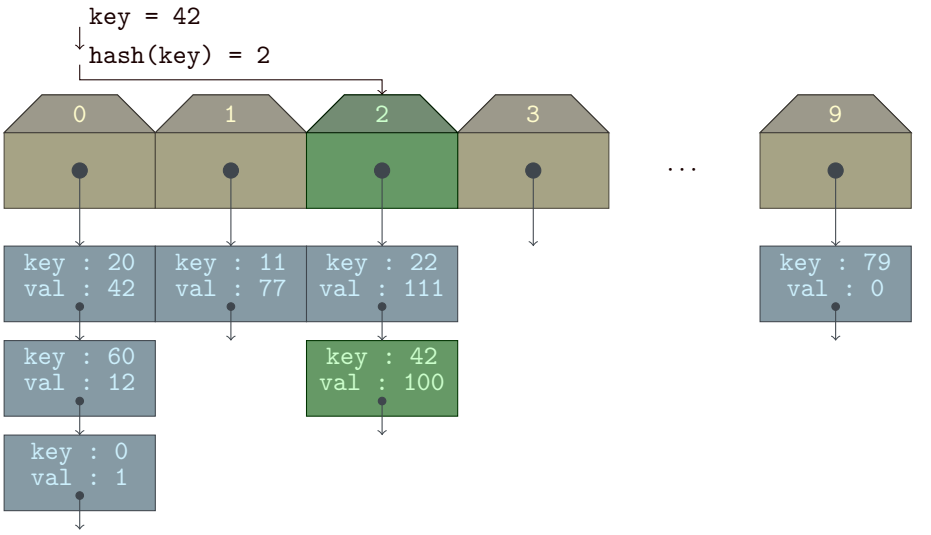
Renseignez vous sur quelques directives préprocesseur standard :

- `__FUNCTION__`
- `__FILE__`
- `__LINE__`
- `__VA_ARGS__`

Exercice noté 52 (⋆ ⋆ ⋆⋆ Implémentation table de hachage).

Nous vous proposons d'implémenter une table de hachage. Une table de hachage est une structure de données qui permet d'associer une **clé** à une **valeur**. Nous proposons ici de rester sur des clés entières et des valeurs entières.

Une table de hachage va réserver un tableau d'éléments d'une capacité donnée. Chaque élément sera accessible via une clé sur laquelle on applique une fonction de hachage. Pour les entiers, nous pouvons garder sa valeur et procéder modulo la capacité. Une fois l'emplacement donné par le hachage de la clé, nous allons à l'indice correspondant dans le tableau d'éléments. Ces éléments peuvent être au choix des tableaux ou des listes chaînées contenant les deux informations que sont la clé et la valeur associée.



La table ci-dessus correspond aux associations suivantes :
$\{0 \mapsto 1, 11 \mapsto 77, 20 \mapsto 42, 22 \mapsto 111, 42 \mapsto 100, 60 \mapsto 12, 77 \mapsto 0\}$.

Nous aimerions comparer les performances de deux structures de données pour le comptage d'éléments et la recherche de ceux-cis. Un fichier `main.c` vous est donné pour effectuer la comparaison entre une table de hachage implémentée dans un module `hashmap` et un tableau dans un module `arraylist`. Les fichiers d'entête des modules vous sont donnés pour implémentation (à ne pas modifier) :

```c
/* fichier hashmap.h */
#ifndef DEF_HEADER_HASHMAP
#define DEF_HEADER_HASHMAP

#include <stdio.h>

/* Table de hachage */
typedef struct HashMap HashMap;
struct HashMap;

/* création d'une table de hachage */
HashMap * HashMap_creer(int capacite);

/* libération d'une table de hachage */
void HashMap_free(HashMap ** hashmap);

/* ajout de 1 à la valeur associée à key, affectation à 1 si non
↪  trouvée */
int HashMap_ajouter(HashMap * hashmap, int key);

/* affiche la table de hachage dans un flux flow */
void HashMap_afficher(FILE * flow, const HashMap * hashmap);

/* renvoie le nombre d'ajouts de la clé key (valeur associée) */
int HashMap_compter(const HashMap * hashmap, int key);

#endif
```

```c
/* fichier arraylist.h */
#ifndef DEF_HEADER_LISTE
#define DEF_HEADER_LISTE

#include <stdio.h>

/* Liste sous forme d'un tableau de valeurs */
typedef struct ArrayList ArrayList;
struct ArrayList;

/* création d'une liste */
ArrayList * ArrayList_creer(int capacite);
```

```c
/* libération d'une liste */
void ArrayList_free(ArrayList ** arraylist);

/* ajout d'un élément dans la liste */
int ArrayList_ajouter(ArrayList * liste, int valeur);

/* affiche la liste dans un flux flow */
void ArrayList_afficher(FILE * flow, const ArrayList * liste);

/* compte le nombre d'occurrences d'une valeur dans la liste */
int ArrayList_compter(const ArrayList * liste, int valeur);

#endif
```

```c
/* fichier main.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include "arraylist.h"
#include "hashmap.h"

#define BENCHMARKS
#undef BENCHMARKS

#ifdef BENCHMARKS
#define CAPACITY   10000
#define ELEMENTS   1000000
#define RESEARCHES 10000
#else
#define CAPACITY 5
#define ELEMENTS 20
#endif

int main() {
  HashMap * map = HashMap_creer(CAPACITY);
  ArrayList * liste = ArrayList_creer(CAPACITY);
  long seed = time(NULL);
  int value;
```

```
  long i;
#ifdef BENCHMARKS
  int count;
  fprintf(stderr, "Filling ArrayList\n");
  clock_t start, stop;
  start = clock();
#endif
  srand(seed);
  for(i = 0; i < ELEMENTS; ++i) {
    value = rand() % ELEMENTS;
    ArrayList_ajouter(liste, value);
  }
#ifdef BENCHMARKS
  stop = clock();
  fprintf(stderr, "Filling completed in %g s\n", (double)(stop -
  ↪  start) / CLOCKS_PER_SEC);
  fprintf(stderr, "Filling HashMap\n");
  start = clock();
#endif
  srand(seed);
  for(i = 0; i < ELEMENTS; ++i) {
    value = rand() % ELEMENTS;
    HashMap_ajouter(map, value);
  }
#ifdef BENCHMARKS
  stop = clock();
  fprintf(stderr, "Filling completed in %g s\n", (double)(stop -
  ↪  start) / CLOCKS_PER_SEC);
#endif
#ifndef BENCHMARKS
  HashMap_afficher(stdout, map);
  ArrayList_afficher(stdout, liste);
#else
  seed *= 42;
  fprintf(stderr, "Research in ArrayList\n");
  start = clock();
  srand(seed);
  count = 0;
  for(i = 0; i < RESEARCHES; ++i) {
    value = rand() % ELEMENTS;
    count += ArrayList_compter(liste, value);
```

```
  }
  stop = clock();
  fprintf(stderr, "Research completed in %g s\n", (double)(stop -
  ↪  start) / CLOCKS_PER_SEC);
  fprintf(stderr, "Counting %d elements\n", count);

  fprintf(stderr, "Research in HashMap\n");
  start = clock();
  srand(seed);
  count = 0;
  for(i = 0; i < RESEARCHES; ++i) {
    value = rand() % ELEMENTS;
    count += HashMap_compter(map, value);
  }
  stop = clock();
  fprintf(stderr, "Research completed in %g s\n", (double)(stop -
  ↪  start) / CLOCKS_PER_SEC);
  fprintf(stderr, "Counting %d elements\n", count);
#endif
  HashMap_free(&map);
  ArrayList_free(&liste);
  exit(EXIT_SUCCESS);
}
```

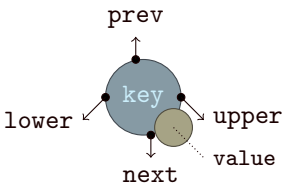
Exercice noté 53 (⋆⋆⋆⋆ Représentation d'un lexique).

Nous vous proposons de représenter un lexique depuis la lecture de mots dans un fichier à l'aide du structure de nœuds chaînés. La structure est la suivante :

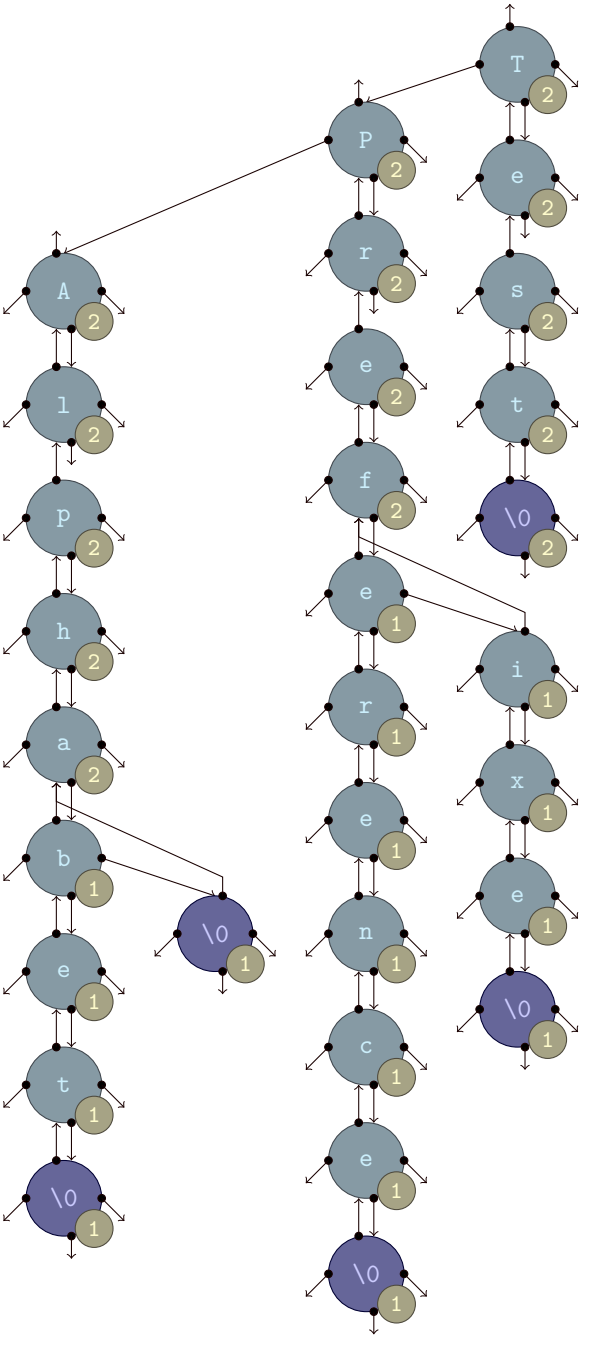
```
typedef struct Node Node;
struct Node {
  char key;
  int value;
  Node * lower;
  Node * upper;
  Node * next;
  Node * prev;
};
```

- `key` correspond à une lettre (la lettre regardée dans un mot).
- `value` correspond au nombre de fois où ce nœud a été visité avec la lettre donnée.
- `lower` correspond à un nœud vers une lettre plus petite mais au même indice dans une chaîne de caractères que le nœud courant.
- `upper` correspond à un nœud de même indice mais dont la lettre est plus grande.
- `next` correspond à un nœud dont la lettre suit la lettre courante dans un mot.
- `prev` correspond à une nœud vers la lettre qui précède celle du nœud courant.

Visuellement, on peut représenter un nœud comme suivant :



Sur les ajouts successifs des mots suivants, ceci donnerait le schéma associé ci-dessous : `Test`, `Preference`, `Prefixe`, `Test`, `Alpha` et `Alphabet`.


T
2
P
2
e
2
r
2
s
2
A
2
e
2
t
2
l
2
f
2
\0
2
p
2
e
1
h
2
i
1
r
1
a
2
x
1
e
1
b
1
e
1
\0
1
n
1
\0
1
c
1
e
1
t
1
e
1
\0
1
\0
1

1. Depuis cette représentation, nous afficherons les mots en ordre lexicographique :

```
Alpha : 1
Alphabet : 1
Preference : 1
Prefixe : 1
Test : 2
```

2. Puis, nous afficherons le nombre d'occurrence de chaque préfixe :

```
2 : A
2 : Al
2 : Alp
2 : Alph
2 : Alpha
1 : Alphab
1 : Alphabe
1 : Alphabet
2 : P
2 : Pr
2 : Pre
2 : Pref
1 : Prefe
1 : Prefer
1 : Prefere
1 : Preferen
1 : Preferenc
1 : Preference
1 : Prefi
1 : Prefix
1 : Prefixe
2 : T
2 : Te
2 : Tes
2 : Test
```