TP C avancé n^o 3
G. Subrenat, Y. Bertrand, N. Courilleau, D. Meneveaux

Licence Informatique L3 - 2021-2022
UFR SFA - Université de Poitiers

Compilation

Table des matières

Résumé et cadre de travail

Dans ce TP il s'agit d'étudier en détail le mécanisme de compilation séparée (on parle de *module* : couple *.h/.c*) ainsi que de concepts liés (mots-clés *static* et *extern*).

Les exercices sont généralement précédés par une partie explicative.

Même si les IDE gèrent les phases de compilation, il est primordial de comprendre les mécanismes de la compilation séparée.

1 Définir ou déclarer ?

1.1 Explications

Point de cours

On a souvent tendance à utiliser indifféremment les verbes “définir” et “déclarer” pour une variable ou une fonction (ou une classe).

Voici les significations correctes :

- *définir* signifie “créer”
- *déclarer* signifie “être défini ailleurs”

On ne peut donc définir qu'une seule fois une entité et on peut la déclarer plusieurs fois.

Pour une fonction, c'est non ambigu :

- pour la *déclarer* on met uniquement le prototype (généralement dans un *.h*)
- pour la *définir* on indique le prototype et le corps de la fonction (généralement dans un *.c*)

Pour une variable locale (à un bloc d'instructions), il y a encore moins d'ambiguïté : une variable locale ne peut qu'être *définie*. Il est préférable, pour des raisons de lisibilité, de définir les variables locales en début de bloc (juste après l'accolade ouvrante).

Pour les variables globales c'est plus délicat car la même instruction peut être une définition ou une déclaration selon le contexte.
S'il y a une initialisation alors c'est obligatoirement une définition :

```
int nb = 3;
```

S'il n'y a pas d'initialisation alors ce peut être une déclaration ou une définition [a] :

```
int nb;
```

Comme il est “hors de question” de laisser un code aussi peu lisible, on imposera l'utilisation du mot-clé *extern* pour les déclarations (cf. ci-dessous).

a. Une variable globale (plus exactement une variable à durée de vie statique) définie sans être explicitement initialisée est initialisée à zéro ; une variable locale non explicitement initialisée contient une valeur a priori aléatoire.

1.2 Exercice

Le but est d'écrire une fonction, nommée *afficheN*, qui affiche plusieurs fois un messages (fonction à deux paramètres donc).

Écrivez un fichier *.c* qui contient dans cet ordre :

- une déclaration de *afficheN*
- une déclaration de *afficheN*
- la définition de *afficheN*
- une déclaration de *afficheN*
- la fonction *main* (sa définition) qui, dans le corps ;
  - déclare la fonction *afficheN*
  - appelle la fonction *afficheN*

L'exercice a pour seul intérêt de montrer qu'on peut déclarer plusieurs fois une entité (une fonction en l'occurrence).

1.3 Exercice

Le but est de faire de la récursivité croisée avec deux fonctions *moins1* et *moins2*.

Chacune prend un entier en paramètre, soustrait un nombre (1 ou 2 donc) et appelle l'autre fonction pour continuer les calculs. On arrête la récursivité [1] lorsqu'on arrive à un nombre inférieur ou égal à 1.

Voici le code [2] :

moins1

```
1 int moins1(int n)   // définition
2 {
3     if (n <= 1)
4         return n;
5     n -= 1;
6     return moins2(n);
7 }
```

moins2

```
1 int moins2(int n)   // définition
2 {
3     if (n <= 1)
4         return n;
5     n -= 2;
6     return moins1(n);
7 }
```

Dans un seul fichier *.c* (sans *.h*) écrivez ces deux méthodes et un *main* qui en appelle une.
Le fichier doit compiler avec les options habituelles et sans warning.

2 Notion de module

2.1 Définitions

Point de cours

Un couple de fichiers *.h/.c* est en règle générale un module (on fera souvent l'abus de dire que c'est une classe au sens objet du terme).
C'est donc une entité :

- spécialisée : elle ne gère qu'un seul concept de manière complète, par exemple un vecteur géométrique, une image, une voiture, ...
- autonomme (si possible) : on entend par ce terme qu'elle peut être réutilisée dans un autre logiciel sans changement.

Dans notre contexte il y a deux types de personnes [a] :

- le concepteur du module : c'est un développeur considéré comme chevronné. C'est lui qui écrit le couple *.h/.c*.
- l'utilisateur du module (ou bibliothèque) : c'est aussi un développeur [b], client du module (pour simplifier celui qui écrit le fichier *main.c*).

Un concepteur partira toujours du principe que l'utilisateur est un mauvais programmeur, et donc il protégera son module au maximum contre une utilisation erronée.

Le fichier *.h* est l'interface publique qui contient uniquement les informations utiles pour l'utilisateur du module. C'est le "quoi", i.e. la description de ce que peut faire le module.

Le fichier *.c* est l'implémentation du module ; l'utilisateur du module n'a pas utilité de regarder ce fichier, et très souvent d'ailleurs il ne le peut pas physiquement (lorsque le code est fourni sous forme de bibliothèque pré-compilée). C'est le "comment", i.e. les solutions choisies et leurs implémentations.

Théoriquement le *.h* devrait être écrit avant le *.c*. C'est en fait assez logique, il est préférable de savoir ce qu'on veut faire avant de se demander comment on le fera.

Le *.c* doit pouvoir être modifié à tout moment sans impact pour les utilisateurs (si ce n'est la performance).
En revanche la modification d'un *.h* impose souvent à l'utilisateur de modifier une partie importante de son code : il faut donc particulièrement soigner l'écriture de ce fichier.

a. de manière schizophrénique, nous jouerons les deux rôles.
b. l'utilisateur final (i.e. la personne utilisant le logiciel), qui n'est a priori pas un informaticien, nous intéresse très peu dans ce cours.

1. plancher de la récursivité
2. ou pourquoi faire simple quand on peut faire compliqué ?

2.2 Exercice

Créez un module *ImageMono* qui représente une image mono-couleur (i.e. tous les pixels ont la même couleur).

Une telle image contient :
- la largeur et la hauteur (des entiers)
- la couleur (une chaîne de caractères)

Note : on ne gérera pas la mémoire ; la chaîne aura une taille en dur et on supposera que le nom d'une couleur ne sera jamais trop grand [3].

Les opérations (i.e. les méthodes/fonctions du module) possibles sont (et la liste est exhaustive) :
- initialiser les données (largeur, hauteur, couleur) ; c'est un "constructeur"
- récupérer la largeur
- récupérer la hauteur
- récupérer la couleur
- changer la couleur

Notamment ce module interdit la modification de la largeur ou de la hauteur après l'initialisation.

Le fichier *main.c* :
- créera une image bleue d'une largeur de 20 pixels et d'une hauteur de 30 pixels
- changera la couleur en vert
- affichera les 3 informations

Écrivez, dans cet ordre :
- *ImageMono.h*
- *main.c* que vous compilerez (option *-c* de *gcc* ou *"make main.o"*) avant d'écrire *ImageMono.c*
- *ImageMono.c*

Testez votre programme avec *valgrind*.

Comparez votre solution avec celle fournie et regardez si vous êtes en accord avec les remarques.

3 Mot-clé *static*

Une variable ou une fonction peut être *static*.

3.1 Fonctions *static*

3.1.1 Définition

Point de cours

Une fonction *static* est une fonction qui n'est connue et appelable que dans le fichier *.c* où elle est définie/déclarée. Autrement dit, une fonction *static* est locale au fichier de définition, et elle n'est pas utilisable dans un autre fichier *.c*.

Cela n'a donc pas de sens (il est même incorrect) de déclarer/définir une fonction *static* dans un *.h* (car les fonctions déclarées dans un *.h* sont par essence utilisables dans d'autres *.c*).

Il y a deux intérêts aux fonctions *static* :
- Le principal est donc qu'elles sont locales à un seul fichier ; elles sont donc privées au sens objet du terme.
- On peut avoir plusieurs fonctions qui ont le même nom à condition qu'elles soient *static* et dans des fichiers *.c* différents.

3. Bien entendu ce n'est pas correct, mais on veut juste se focaliser sur la notion de module.

Il est fortememnt déconseillé d'avoir une fonction *static* et une fonction *non-static* qui ont le même nom. En général cela conduit à une erreur de compilation.
Le seul cas où cela ne pose pas de problème, c'est lorsque le module est autonome, car par définition il n'appelle pas de fonctions extérieures (hormis les fonctions propres au langage).

En fait il y a une manière d'appeler une fonction *static* hors de son fichier de description, mais ce n'est pas abordé dans ce cours [a].

Syntaxe :

```
static <type> <nom_fonction>([paramètres])
```

Exemple :

```
static float moyenne(const int t[], int n)
```

a. Pour les curieux, il faut récupérer un pointeur sur cette fonction, ce qui n'est possible que si le module le permet explicitement.

3.1.2 Exercice

Écrivez un fichier *fonctions.c* qui contient une fonction *f* non-*static* et une fonction *g static*.

Écrivez un fichier *main.c* qui appelle ces deux fonctions.

Il n'y a pas de *.h* dans cet exercice.

Compilez en ignorant les warnings et en ne regardant que les erreurs. Quel appel dans *main.c* est impossible ?

3.2 Variable *static* définie hors des fonctions

3.2.1 Définition

Point de cours

Rappelons que sans *static*, une variable définie hors des fonctions est persistante/permanente et globale ; elle est connue de tous les *.c*.

Une telle variable *static* a le même comportement qu'une fonction *static* : elle est toujours persistante, mais locale au fichier de définition (i.e. utilisable uniquement dans les fonctions du module).
Elle n'est ni connue, ni accessible dans un autre *.c*.

Corrolaire, si l'instruction :

```
static int v = 3;
```

est écrite dans deux *.c* différents, alors chaque fichier *.c* a sa propre variable *v* qui n'interfère pas avec l'autre fichier.
Deux variables *static* de même nom (dans des fichiers différents) peuvent avoir des types différents.

3.2.2 Exercice

Écrivez un fichier *variables.c* qui définit une variable *g* non *static* et une variable *s static*.

Écrivez un fichier *main.c* qui :

- déclare (avec le mot-clé *extern* que nous verrons par la suite) les deux variables comme si elles étaient toutes les deux non *static*, par exemple pour la variable *s* :

```
extern int s;
```

- les affiche

Il n'y a pas de *.h* dans cet exercice.

Compilez en ignorant les warnings et en ne regardant que les erreurs. Quel accès dans *main.c* échoue ?

3.3 Variable *static* locale à une fonction

3.3.1 Définition

Point de cours

Pour une variable locale, la notion de *static* est très différente.

Appuyons-nous sur un exemple :

f_correct

```
1 void f_correct()
2 {
3     static int n = 3;
4     n ++;
5     printf("%d\n", n);
6 }
```

f_faux

```
1 void f_faux()
2 {
3     static int n;
4     n = 3;
5     n ++;
6     printf("%d\n", n);
7 }
```

Une telle variable est locale à la fonction (comme une variable normale) et n'est donc pas utilisable hors de la fonction (et a fortiori hors du module).

En revanche elle est persistante : sa valeur est conservée entre deux appels.

On en déduit que la variable n n'est initialisée qu'une seule fois lors du premier appel. Autrement dit, la ligne 3 est exécutée lors du premier appel et ignorée lors des appels suivants.
C'est pour cela que la version de droite est erronée : la ligne 3 est exécutée lors du premier appel (et n est initialisée à 0) ; en revanche la ligne 4 est exécutée chaque fois ce qui fait perdre tout l'intérêt d'avoir une variable *static*.

Étudions la fonction *main* suivante :

main

```
1 int main()
2 {
3     f_correct();
4     f_correct();
5     f_correct();
6
```

main (suite)

```
7     f_faux();
8     f_faux();
9     f_faux();
10
11    return EXIT_SUCCESS;
12 }
```

Les lignes 3 à 5 vont afficher successivement 4, 5 et 6 :
- lors du premier appel, n est initialisée à 3 et passe à 4
- Lors du deuxième appel, la première ligne de la fonction est ignorée et n passe à 5
- Lors du troisième appel, la première ligne de la fonction est ignorée et n passe à 6

Les lignes 7 à 9 vont afficher trois fois la valeur 4 :
- lors du 1^{er} appel, n est initialisée à 0, passe à 3 et passe à 4
- Lors du 2^{me} appel, la première ligne de la fonction est ignorée, n passe à 3 et passe à 4
- Lors du 3^{me} appel, la première ligne de la fonction est ignorée, n passe à 3 et passe à 4

Ce n'est vraisemblablement pas ce qui était voulu.

Ligne 11 : c'est bien *"return EXIT_SUCCESS"* qu'il faut, et non pas *"return 0"*.

3.3.2 Exercice

Testez le code ci-dessus.

4 Mot-clé *extern*

Point de cours

Normalement, le mot *extern* devrait être placé devant toute déclaration (et non définition) d'une variable globale ou d'une fonction globale.

Cependant il est systématiquement omis pour les fonctions. En effet il n'y a pas d'ambiguïté pour une fonction :

- soit il n'y a pas de corps et il s'agit d'une déclaration
- soit il y a un corps et il s'agit d'une définition

On n'utilise pratiquement jamais le mot-clé *extern* pour les fonctions.

Pour une variable c'est plus délicat car ambigu. Le problème ne se pose que pour les variables globales (i.e. non locales à une fonction ou un fichier).

Considérons un premier exemple :

```
int v = 0;
```

Cet exemple ne pose pas de problème : c'est une définition. Le fait d'initialiser une variable globale ne peut se faire que lors de sa définition.

Considérons un second exemple :

```
int v;
```

Est-ce une déclaration ? Ou est-ce une définition avec initialisation par défaut [a] à 0 ?
Le problème est que ce peut être les deux. En général le compilateur le devine seul, mais il me semble que parfois ce n'est pas le cas. En revanche si on compile avec le compilateur du C++ (`g++`) alors c'est systématiquement une définition.

Aussi prend-on l'habitude de faire précéder toutes les déclarations de variables globales par le mot-clé *extern*, ne serait-ce que pour que l'ensemble soit lisible. Les déclarations de variables globables sont quasi-systématiquement écrites dans les *.h* des modules.

Voici le code d'une déclaration :

```
extern int v;
```

L'exemple ci-dessus n'est pas ambigu : il ne peut s'agir que d'une déclaration.

Attention, *extern* ne signifie absolument pas qu'une variable est globale ; il signifie simplement que la variable est définie ailleurs. Ceci dit, comme nous l'avons vu, le mot *extern* n'a de sens que pour les variables globales ; mais il ne faut pas confondre la cause et l'effet.

Rappelons que les variables globales sont à éviter au maximum ; aussi est-on rarement confronté au problème.

a. (cf. note précédente) Les variables globales et les variables *static* sont initialisées par défaut à 0, au contraire des variables locales qui n'ont pas d'initialisation par défaut. Ceci dit il est bien plus lisible d'initialiser explicitement une variable globale/*static* à 0.

5 Fichiers d'entête *.h*

5.1 Définition

Point de cours

Un fichier *.h* (nommé également fichier d'entête) est la partie publique d'un module. Il contient généralement :

- des déclarations [a] de fonctions (très fréquent)

- des définitions de types
- des commentaires (quasi-systématiquement)
- des macros
- des déclarations de variables globales (très rare)

a. et non pas définition

5.2 Problématique des inclusions multiples

Point de cours

Un même fichier *.h* peut être inclus dans plusieurs fichiers *.c*, mais également dans d'autres fichiers *.h*. De fait il est possible d'inclure, dans un fichier *.c*, plusieurs fois le même *.h* sans s'en apercevoir.

Voyons cela avec un exemple simple et un *.h* système : *stdbool.h*. On suppose qu'on a un module de gestion de calendrier (*calendrier.h* et *calendrier.c*) avec notamment une fonction qui indique si une année est bissextile.

calendrier.h

```
1 #include <stdbool.h>
2
3 bool estBissextile(int annee);
```

calendrier.c

```
1 #include "calendrier.h"
2
3 bool estBissextile(int annee)
4 { ... }
```

Enfin nous considérons le programme principal, *main.c*, qui utilise le module calendrier, mais également des booléens par ailleurs, donc :

main.c

```
1 #include <stdbool.h>
2 #include "calendrier.h"
3
4 ...
```

Et on voit que *stdbool.h* est inclus deux fois dans *main.c*, ce qui est potentiellement une catastrophe si *stdbool.h* définit des types (ce qui est bien entendu le cas).

Ce problème doit absolument avoir une solution, automatique si possible.

5.3 Solution non retenue

Point de cours

Nous continuons sur l'exemple du calendrier.

Puisque *stdbool.h* est déjà inclus dans *calendrier.h*, on n'inclut pas *stdbool.h* dans *main.c*.

Un premier problème est que si *main.c* n'a plus besoin du module des calendriers, il supprimera le *include* correspondant, et il faudra penser à remettre le *include* de *stdbool.h*.

Et de toute façon ce n'est pas satisfaisant car un programmeur n'a pas à connaître la liste des *.h* qu'il inclut indirectement. Par exemple, quand on inclut *stdio.h*, on inclut une quinzaine d'autres fichiers sans le savoir.
Cf. commandes pour avoir la liste des *.h* inclus :

tous les .h

```
gcc -M main.c
```

uniquement les .h non système

```
gcc -MM main.c
```

Donc cette solution n'est pas acceptable et n'est pas retenue.

5.4 Solution retenue

Point de cours

Lorsqu'on inclut plusieurs fois un *.h* dans un *.c*, le *.h* visé se débrouille tout seul pour gérer et empêcher les inclusions multiples. C'est la solution retenue.

Le principe est d'englober chaque *.h* dans une macro conditionnelle qui gère notre problème. Voici le principe à appliquer pour tout *.h* :

toto.h

```
1 #ifndef TOTO_H
2 #define TOTO_H
3
4 // contenu du .h
5
6 #endif
```

La seule difficulté est de trouver un *define* différent pour chaque *.h*, c'est pourquoi on prend généralement le nom du fichier, par convention en majuscules.

Expliquons, sur un cas d'école avec le fichier *main.c*, pourquoi cela marche :

main.c

```
1 #include "toto.h"
2 #include "toto.h"
3
4 ...
```

Ce qui est équivalent à :

main.c

```
1  #ifndef TOTO_H
2  #define TOTO_H
3
4  // contenu du .h
5
6  #endif
7  #ifndef TOTO_H
8  #define TOTO_H
9
10 // contenu du .h
11
12 #endif
13
14 ...
```

- Ligne 1 : "si TOTO_H n'est pas défini alors". C'est bien le cas puisqu'on n'a encore rien fait, donc on va exécuter les lignes 2 à 5.
- Ligne 2 : on définit *TOTO_H*.
- Lignes 3 à 5 : toutes les lignes sont prises en compte.
- Ligne 7 : "si TOTO_H n'est pas défini alors". Mais cette fois-ci *TOTO_H* est défini (par la ligne 2) donc le test échoue et on passe directement à la ligne 12, et par conséquent on ne prend pas en compte une deuxième fois le fichier *toto.h*.

Note : tout ce mécanisme est exécuté par le préprocesseur et non par le compilateur. Autrement dit le compilateur ne voit pas les lignes 7 à 12.

Ce mécanisme peut être remplacé (a priori avantageusement) par la directive *"#pragma once"*. Mais il n'est pas exclu que cette dernière ne soit pas standard.

5.5 Exercice

Dans un premier temps n'implémentez pas le mécanisme de protection des inclusions multiples.

Écrivez un fichier *Vecteur.h* qui définit une structure contenant les coordonnées *(x,y,z)*.

Écrivez un fichier *Repere.h* qui définit une structure contenant un tableau de trois *Vecteur*.

Écrivez le fichier *main.c* qui inclut les deux fichiers précédents et uniquement ceux-ci (i.e. pas les *.h* système). La fonction *main* fait uniquement un *"return 0 ;"*[4].

Compilez le programme et notez l'erreur de redéfinition. Lancez uniquement le préprocesseur sur *main.c* ("*gcc -E main.c*") pour voir l'erreur.

Activez la protection des inclusions multiples et relancez la compilation complète et le préprocesseur.

5.6 Inclusions croisées

Point de cours

Voici un problème peu fréquent et pas toujours simple à résoudre : les inclusions croisées. On a une inclusion croisée lorsqu'un fichier *toto.h* inclut un fichier *titi.h* et réciproquement.

toto.h

```c
#ifndef TOTO_H
#define TOTO_H

#include "titi.h" // car on utilise un type défini dans titi.h : myInt (ligne 7)

typedef float myFloat; // création d'un alias pour float
void g(myInt i);

#endif
```

titi.h

```c
#ifndef TITI_H
#define TITI_H

#include "toto.h" // car on utilise un type défini dans toto.h : myFloat (ligne 7)

typedef int myInt; // création d'un alias pour int
void f(myFloat f);

#endif
```

main.c

```c
#include "toto.h"

...
```

Le problème est ici extrêmement articiel et ne sert qu'à des fins pédagogiques pour des explications sur un code simplifié.
Un exemple concret et classique est le suivant :

- *Vecteur.h* : a besoin de *Point.h* pour construire un vecteur à partir de deux points
- *Point.h* : a besoin de *Vector.h* pour écrire la fonction de translation qui transforme un point via un vecteur.

5.7 Exercice

Compilez le programme de la section précédente (le code est fourni) :

```
$ gcc -Wall -Wextra -pedantic -std=c99 -c main.c
```

Quel est le problème ? Y a-t-il une solution ?

4. Pourquoi exceptionnellement ne fait-on pas un *"return EXIT_SUCCESS ;"* ?

6 Initialisation vs. affectation

6.1 Explications

Point de cours

Initialiser une variable signifie : “mettre une valeur lors de la définition”.
Affecter une variable signifie : “remplacer la valeur courante par une autre”.
Il y a souvent des abus de langage où l'on emploie un mot pour l'autre [a].

Note : cette différence entre ”initialisation” et ”affectation” prendra une grande importance en programmation objet (peut-être plus particulièrement en C++) avec les constructeurs et les opérateurs d'affectation.

Voici un premier code :

toto.h

```
1 int a = 3;
2 a = 7;
```

- Ligne 1 : a est définie et initialisée à 3.
- Ligne 2 : a est affectée (la valeur 3 est remplacée par 7).

Voici un second code :

toto.h

```
1 int a;
2 a = 7;
```

- Ligne 1 : a est définie et soit initialisée à 0 si c'est une variable globale, soit ”initialisée” à une valeur ”aléatoire” pour une variable locale.
- Ligne 2 : a est affectée (sa valeur est remplacée par 7).
- Si a est une variable locale, par abus on dit qu'elle est initialisée à 7 lors de la ligne 2. C'est d'ailleurs conceptuellement vrai (i.e. c'est bien le sens du programme) mais c'est techniquement faux.

Et pour les constantes ? Par définition une constante doit être initialisée et ne peut pas être affectée.

a. ce qui généralement ne porte pas à conséquence

6.2 Exercice

Définissez, sans les initialiser, 5 variables entières :
- une variable globale
- une variable *static* hors des fonctions
- une variable *static* dans une fonction
- une variable non *static* dans une fonction autre que le *main*
- une variable non *static* dans la fonction *main*

Affichez toutes ces variables.

Quelles variables sont concernées par des warnings du compilateur et pourquoi ?

Quelles variables ne sont pas concernées par des warnings du compilateur et pourquoi ?

L'affichage des cinq variables est-il celui attendu ?

6.3 Exercice

Définissez, sans les initialiser, 4 constantes (mot-clé *const*) entières :
- une constante globale

- une constante *static* hors des fonctions
- une constante *static* dans une fonction autre que le *main*
- une constante non *static* dans une fonction autre que le *main*
- une constante non *static* dans la fonction *main*

Notez quelles constantes sont concernées par un warning, et expliquez.

Au fait, pourquoi définir une constante *static* dans une fonction n'a pas d'intérêt ?

Essayez d'affecter chacune des 5 constantes et regardez les messages du compilateur.

7 Exercice complet

Afin de se concentrer sur la compilation séparée et uniquement sur celle-ci, le sujet de l'exercice est volontairement "simpliste".

7.1 Module "mathématiques financières"

On a besoin de faire des opérations sur les comptes bancaires, en l'occurrence des additions et des soustractions (!).
Nous n'allons pas utiliser les opérateurs + et -, mais des fonctions dédiées.

Il faut créer le module *operation* avec les fichiers *operation.h* et *operation.c*.

Deux fonctions principales sont à définir :

- *int plus(int a, int b)* qui renvoie la somme de a et b ;
- *int moins(int a, int b)* qui renvoie la différence de a et b.

En outre, il s'agit de définir la fonction :

- *int getNbOperations()* qui renvoie le nombre d'appels à *plus* et *moins*, c'est-à-dire qui compte le nombre d'appels effectués aux opérations *plus* et *moins*.

De manière gratuite, ce module contiendra une variable globale *dummy* entière qui ne sert à rien. Elle sera initialisée à 14 à sa création. Les autres modules peuvent la manipuler à leur guise.

Créez les fichiers *operation.h* et *operation.c*.

7.2 Module "gestion de comptes"

Il faut créer le module *comptes* avec les fichiers *comptes.h* et *comptes.c*.
Ce module gère les comptes bancaires d'une agence.

Il se trouve que le nombre de comptes est fixe (et immuable), par exemple 10, mais ce nombre doit être modifiable facilement. Pour l'énoncé, *N* sera le nombre de comptes. Les comptes sont numérotés de 0 à N-1. Les comptes ne sont pas initialisés à 0 euro, mais à un montant fixé dans le programme (par exemple 15 euros).

Les soldes des comptes sont dans un tableau d'entiers (de taille N). L'initialisation du tableau doit se faire dans une fonction et non pas directement lors de la définition du tableau ; il faut donc se débrouiller pour que le code d'initialisation de cette fonction :

- soit exécuté une seule fois
- et soit exécuté avant le code de toute autre opération (consultation ou modification de solde)

Les fonctions accessibles à l'utilisateur du module sont :

- *int getNbComptes()*
- *int getSoldeCompte(int numCompte)*
- *void creditCompte(int numCompte, int montant)*
- *void debitCompte(int numCompte, int montant)*

Ce module ne donne pas accès à d'autres fonctions et les prototypes sont imposés.

Les fonctions *creditCompte* et *debitCompte* n'ont pas le droit d'utiliser + et -, mais doivent utiliser les fonctions *plus* et *moins* du module *operation*.

La fonction *creditCompte* est particulière. En règle générale elle ajoute au compte désigné le montant fourni. Mais tous les 7 appels (encore fois c'est une constante qui doit être facilement paramétrable) elle ajoute 1 en plus du montant ; c'est une sorte de publicité que fait la banque. Notez que ce compteur est commun à tous les comptes. On pourrait imaginer avoir un compteur par compte bancaire (ce qui serait plus logique), mais ce n'est pas demandé.

En revanche les N comptes bancaires se sont pas accessibles directement hors du fichier *comptes.c* ; il faut passer par les quatre fonctions décrites ci-dessus.

Il existe une fonction :

verifNumCompte

```
1 void verifNumCompte(int numCompte)
2 {
3     if ((numCompte < 0) || (numCompte >= getNbComptes()))
4     {
5         fprintf(stderr, "numéro de compte incorrect.\n");
6         exit(EXIT_FAILURE);
7     }
8 }
```

Cette fonction est interne au module *comptes* et est appelée dans chaque fonction pour vérifier de manière brutale que le numéro de compte est valide. Interne au module signifie qu'elle ne peut pas être appelée en dehors du module.

Créez les fichiers *comptes.h* et *comptes.c*.

7.3 Module du programme principal : *main*

Il n'y a aucune raison de placer la fonction *main* dans le module *operation* ou *comptes*. En effet ces modules peuvent être utilisés par d'autres programmeurs qui auront leur propre *main*.

Il faut donc un module spécifique cette fonction. On note qu'il n'y a pas de *.h* correspondant. Le fichier *main.c* est fourni à l'exception des *include* des deux modules à ajouter.

7.4 Compilation

Solution 1 : (à éviter en général)

```
$ gcc -Wall -Wextra -pedantic -std=c99 -g *.c -o main
```

Solution 2 : (mieux)

```
$ gcc -Wall -Wextra -pedantic -std=c99 -g operation.c comptes.c main.c -o main
```

Solution 3 : (idéale mais ingérable sans Makefile)

```
$ gcc -Wall -Wextra -pedantic -std=c99 -g -c operation.c
$ gcc -Wall -Wextra -pedantic -std=c99 -g -c comptes.c
$ gcc -Wall -Wextra -pedantic -std=c99 -g -c main.c
$ gcc -Wall -Wextra -pedantic -std=c99 -g operation.o comptes.o main.o -o main
```

Solution 4 : avec un Makfile (qui est fourni !)

```
$ make
```

La solution 4 semble la plus adéquate.

Notez que normalement on utilise un IDE qui gère automatiquement les Makefiles.