TP - Programmation modulaire en C

Dès qu'un programme informatique comporte un grand nombre de lignes et de définitions de fonctions, il devient essentiel l'organiser de manière *modulaire*. Ce principe de base du *génie logiciel* consiste à découper un programme en plusieurs parties relativement indépendantes appelées *unités de compilation*. Ce découpage permet à la fois de maîtriser la complexité de logiciels de grandes tailles, de réaliser un développement en équipe et de recompiler rapidement le programme en ne recompilant que ce qui est nécessaire après une modication. Le programme est ainsi découpé en plusieurs fichiers dont certains peuvent être réutilisés par d'autres programmes. On les appelle des *bibliothèques* ou des *modules*.

L'objet de ce TP est la mise en œuvre de cette approche en vue de construire des *modules* associés à l'implantation de structures de données telles que les listes chaînées, les piles et les files. La *compilation séparée* est ensuite l'occasion de se familiariser avec les fichiers `Makefile`.

Modules

Question 1. Dans un fichier `main1.c`, écrire une fonction `power` qui calcule x^n, x et n étant des entiers naturels. Tester votre programme pour qu'il calcule 2^{10}.

Question 2. Dans un fichier `main2.c`, écrire une fonction `gcd` qui calcule le *pgcd* de deux entiers. Tester votre programme pour qu'il calcule le *pgcd* de 128 et 48.

Question 3. On souhaite à présent pouvoir réutiliser les deux fonctions définies dans les fichiers `main1.c` et `main2.c`. Ces fonctions doivent être copiés dans un nouveau *fichier source*, nommé `module.c` et enregistrés dans le répertoire courant où se trouvent déjà les deux fichiers `main1.c` et `main2.c`. Un *fichier d'entête*, nommé `module.h`, est créé avec les *prototypes* des deux fonctions précédentes.

```
// fichier source module.c
int power(int x, int n) {
// votre code
}

int gcd(int a, int b) {
// votre code
}
```

```
// fichier header module.h
int power(int x, int n);
int gcd(int a, int b);
```

□ **3.1.** Créer les fichiers `module.c` et `module.h` puis un nouveau fichier `main3.c` dont le contenu est le suivant.

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

int main() {
  int x = 2;
  int n = 10;
  int a = 48;
  int b = 128;

  printf("power(%d, %d) = %d\n", x, n, power(x, n));
  printf("gcd(%d, %d) = %d\n", a, b, gcd(a, b));
}
```

Noter la présence de guillemets autour de `module.h` qui indiquer que le fichier doit être cherché dans le répertoire courant plutôt que dans la bibliothèque du compilateur.

□ **3.2.** Si notre programme est composé de nombreux fichiers, qui incluent à chaque fois les fichiers d'en-têtes nécessaires, on peut vite se retrouver à inclure plusieurs fois un même entête, par transitivité. Par exemple, si on utilise à la fois les fichiers `module1` et `module2`, et qu'on inclut donc les entêtes `module1.h` et `module2.h`, il se peut que le fichier `module2.h` inclut lui-même l'en-tête `module1.h`, car il a besoin d'un type qui y est défini. Le compilateur se retrouve alors avec le type défini deux fois, ce qui provoque une erreur, avec un message du type suivant.

```
error: redefinition of 'struct ...'
```

Une solution à ce problème consiste à rendre l'inclusion d'un entête indempotente, c'est-à-dire sans effet la seconde fois, en se servant des directives `#ifndef` et `#define` de la manière suivante (ici sur l'exemple de notre fichier `module1.h`).

```
#ifndef MODULE1
#define MODULE1
...
#endif
```

La première fois que le fichier est inclus, la macro `MODULE1` n'est pas définie et tout le contenu du fichier est donc considéré. En particulier, `#define` définit la macro `MODULE1` — avec un contenu vide, en l'occurrence. Si le fichier est inclus de nouveau par la suite, la macro étant maintenant définie, tout le bloc entre `#ifndef` et `#endif` est ignoré, ce qui est l'effet attendu.
Modifier votre fichier d'entête `module.h` pour tenir compte de ces remarques

Retenir / Savoir / Savoir-faire

Un *module* est un ensemble de deux fichiers : un *fichier d'implémentation*[a], d'extension `.c` et un *fichier d'interface*[b], d'extension `.h`. Chaque fichier porte généralement le même nom et on désigne le module par le nom des ces fichiers. Par exemple, un module associé à l'implémentation de la structure de données de listes chaînées peut être identifié par le nom `list`, les deux fichiers de ce module étant `list.c` et `list.h`.

Le *fichier d'implémentation* contient le *code source* du module : instanciations des éventuelles variables globales, définitions des fonctions. Le *fichier d'interface* contient toutes les informations utiles au compilateur pour lui permettre d'utiliser les fonctionnalités d'un module. Il regroupe en particulier les *définitions de types* du module, les *prototypes* des fonctions.

Ces fichiers peuvent eux-mêmes inclure d'autres fichiers d'entête, par l'intermédiaire de l'instruction `#include`.

Dans le cadre de la *programmation modulaire* et, plus largement, du *génie logiciel*, le découpage d'un projet informatique sous la forme de multiples fichiers est une pratique courante.

a. *Source file* en anglais.
b. *Header file* en anglais.

Makefile

Question 4.
□ **4.1.** Compiler votre projet constitué des fichiers sources `main3.c`, `module.c` et du fichier d'entête `module.h` en tapant :

```
gcc module.c main3.c -o main3
```

□ **4.2.** La compilation précédente est équivalente à une *compilation séparée* des fichiers sources puis à une *édition de liens* avec les codes objets `module.o` et `main3.o`. Elle peut se décomposer en trois phases de compilation. Les deux premières phases créent des fichiers objets (extension `.o`). La troisième phase effectue une édition de lien des fichiers objets pour générer un exécutable.

```
gcc -c module.c
gcc -c main3.c
gcc module.o main3.o -o main3
```

Taper ces instructions après avoir nettoyer votre répertoire des fichiers objets `.o` et de l'exécutable `main3`. Se rappeler l'intruction du shell suivante :

```
rm -f *.o main3
```

Vérifier le bon déroulement des opérations en affichant la liste des fichiers du répertoire par l'instruction `ls`.
□ **4.3.** Si plusieurs modules sont créés et utilisés par un programme, les trois lignes de compilation précédentes se transforment en beaucoup plus de lignes et la procédure de compilation plus lourde à gérer en raison des dépendances entre fichiers. Pour automatiser la construction de fichiers exécutables à partir de nombreux fichiers sources, on fait généralement appel à l'utilitaire `make`, programme UNIX qui exécute les commandes définies dans un fichier spécifique `Makefile`[1]. En tapant simplement :

```
make
```

dans le terminal, le programme recherche et lit les instructions contenues dans le fichier `Makefile` du répertoire courant. L'utilitaire n'exécute ces instructions que si cela est nécessaire. Par exemple, en cas de recompilation alors qu'aucune modification n'a été apportée aux fichiers sources, l'utilitaire ne fait rien. L'écriture d'un fichier `Makefile` suit des règles dont certaines sont présentées ci-dessous en vue de répondre à nos besoins spécifiques. La forme générale d'une règle est la suivante.

1. Ou `makefile` avec un m minuscule.

```
cible : dépendance
<tab>   commandes à exécuter
```

En tapant :

```
make cible
```

seule la règle dont la cible est passée en argument de `make` est exécutée.
□ **4.4.** Construire une fichier `Makefile` avec les deux règles indiquées ci-dessous puis exécuter le dans le terminal. Observer l'évolution du contenu de votre répertoire courant.

```
# Construction du fichier main3
# cible : main3
# dépendances : fichiers objets module.o et main3.o
# commande : édition de liens avec gcc des dépendances et construction de l'exé
   cutable main3
main3 : module.o main3.o
        gcc module.o main3.o -o main3

# Suppression des fichiers objets et de l'exécutable du répertoire courant
# cible : clean
# dépendances : aucune
# commande : rm (remove) -f (fichiers) *.o (ceux d'extension .o) et main3
clean :
        rm -f *.o main3
```

□ **4.5.** On peut ajouter des règles pour la construction de chaque fichier objet. Ajouter les lignes suivantes puis tester votre fichier.

```
# Construction des fichiers objets
module.o : module.c
        gcc -c module.c

main3.o : main3.c
        gcc -c main3.c
```

□ **4.6.** Un fichier `Makefile` peut contenir des *macros* qui définissent les substitutions à faire. Une macro est une ligne de la forme :

```
id_macro = <chaine>
```

où `id_macro` est un identifiant de la macro et `<chaine>` est une chaîne de caractères. À l'exécution, une expression de la forme `$(id_macro)` est remplacée par la chaîne associée. Ce mécanisme très utile permet d'adapter un `Makefile` à des usages variés comme, par exemple, choisir entre deux compilateurs C : le compilateur natif `cc` du système ou le compilateur `gcc` de GNU. Il suffit de définir la macro suivante au début du `Makefile` et de remplacer chaque occurrence de `gcc` par `$(CC)`. Si on souhaite utiliser un autre compilateur, seule la chaîne `gcc` de la macro est remplacée.

```
CC = gcc
```

Ajouter cette macro à votre `Makefile` et remplacer tous les autres `gcc` par `$(CC)`.
□ **4.7.** Certaines règles ont un caractère répétitif comme celles définissant les fichiers objets. D'autres *macros prédéfinies* simplifient leur écriture. Les quatre macros suivantes sont associées aux cibles et dépendances d'un règle.

- `$@` : cible d'une règle
- `$<` : première dépendance d'une cible
- `$^` : toutes les dépendances d'une cible
- `$?` : liste des dépendances plus récentes que la cible

En utilisant le symbole `%` pour écrire des *règles génériques*, on peut remplacer les lignes des règles de construction des fichiers objets par la seule ligne suivante.

```
% .o : %.c
        $(CC) $(CLFAGS) $<
```

Si le répertoire courant ne contient que les fichiers utiles pour construire l'exécutable [2], le fichier `Makefile` peut prendre la forme suivante.

2. Éventuellement, les copier dans un répertoire spécifique.

```
CC = gcc
CFLAGS = -c -g -Wall

# recherche de tous les fichiers .c du répertoire
SRC = $(wildcard *.c)
# liste des fichiers .o associés aux .c
OBJ = $(SRC:.c=.o)

main3 : $(OBJ)
        $(CC) $^ -o $@

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

clean :
        rm -f *.o main3
```

Tester ce fichier avec vos fichiers sources.

Retenir / Savoir / Savoir-faire

L'utilitaire `make` est un programme qui permet la construction et la mise à jour automatique de fichiers, souvent des exécutables ou des bibliothèques, à partir de fichiers sources.

Un fichier `Makefile` contient les instructions exécutés par `make`, leur écriture suivant des règles bien précises.

Dans le cadre de la *programmation modulaire*, l'utilisation de cet outil avec un fichier `Makefile` adapté à chaque projet est très largement répandue.

Retenir / Savoir / Savoir-faire

Dans son rapport de l'épreuve orale d'informatique, le CCINP ajoute la note suivante pour la session 2024.

Nous souhaitons encourager de bonnes pratiques de programmation tout au long de la formation. Pour la session 2023, le CCINP proposait aux candidats une ligne de compilation simple, sans avertissements, afin de ne pas pénaliser les candidats en fonction de leur formation, des moyens mis à leur disposition et du type de salle informatique disponible dans leur établissement. À ce titre, nous rappelons que le programme d'informatique de MPI indique explicitement que "le système Linux est le plus propice pour introduire les éléments de ce programme".

À compter de la session 2024, les examinateurs souhaitent que les programmes en C soient compilés en utilisant au moins l'option `-Wall`.

Une ligne de compilation utilisant également les options `-Wextra` et `-fsanitize=address` sera systématiquement proposée et l'examinateur pourra demander aux candidats de l'utiliser lors de leur passage.

Aucune connaissance spécifique sur ces options n'est néanmoins attendue. L'examinateur ne demandera pas aux candidats d'interpréter directement les éventuels messages, mais pourra utiliser ceux-ci comme une base pour une discussions sur les faiblesses éventuelles du programme proposé. Un fichier Makefile type est documenté sur le site du concours (Epreuves orales Filière MPI). Les exercices de type B en langage C pourront commencer par un texte introductif de la forme suivante.

> *Cet énoncé est accompagné d'un code compagnon en C `*.c` fournissant certaines des fonctions mentionnées dans l'énoncé : il est à compléter en y implémentant les fonctions demandées.*
>
> *La ligne de compilation `gcc -o main.exe -Wall *.c -lm` vous permet de créer un exécutable `main.exe` à partir du ou des fichiers C fournis. Vous pouvez également utiliser l'utilitaire `make`. En ligne de commande, il suffit d'écrire `make`. Dans les deux cas, si la compilation réussit, le programme peut être exécuté avec la commande `./main.exe`.*
>
> *Il est possible d'activer davantage d'avertissements et un outil d'analyse de la gestion de la mémoire avec la ligne de compilation `gcc -o main.exe -g -Wall -Wextra -fsanitize=address *.c -lm` ou en écrivant `make safe`. L'examinateur pourra vous demander de compiler avec ces options.*
>
> *Si vous désirez forcer la compilation de tous les fichiers, vous pouvez au préalable nettoyer le répertoire en faisant `make clean` et relancer une compilation.*