Chapitre 7

La programmation modulaire

Dès que l'on écrit un programme de taille importante ou destiné à être utilisé et maintenu par d'autres personnes, il est indispensable de se fixer un certain nombre de règles d'écriture. En particulier, il est nécessaire de fractionner le programme en plusieurs fichiers sources, que l'on compile séparemment.

Ces règles d'écriture ont pour objectifs de rendre un programme lisible, portable, réutilisable, facile à maintenir et à modifier.

7.1 Principes élémentaires

Trois principes essentiels doivent guider l'écriture d'un programme C.

L'abstraction des constantes littérales L'utilisation explicite de constantes littérales dans le corps d'une fonction rend les modifications et la maintenance difficiles. Des instructions comme :

```
fopen("mon_fichier", "r");
perimetre = 2 * 3.14 * rayon;
```

sont à proscrire.

Sauf cas très particuliers, les constantes doivent être définies comme des constantes symboliques au moyen de la directive `#define`.

La factorisation du code Son but est d'éviter les duplications de code. La présence d'une même portion de code à plusieurs endroits du programme est un obstacle à d'éventuelles modifications. Les fonctions doivent donc être systématiquement utilisées pour éviter la duplication de code. Il ne faut pas craindre de définir une multitude de fonctions de petite taille.

La fragmentation du code Pour des raisons de lisibilité, il est nécessaire de découper un programme en plusieurs fichiers. De plus, cette règle permet de réutiliser facilement une partie du code pour d'autres applications. Une possibilité est de placer une partie du code dans un fichier en-tête (ayant l'extension `.h`) que l'on inclut dans le fichier contenant le programme principal à l'aide de la directive `#include`. Par exemple, pour écrire un programme qui saisit deux entiers au clavier et affiche leur produit, on peut placer la fonction `produit` dans un fichier `produit.h`, et l'inclure dans le fichier `main.c` au moment du traitement par le préprocesseur.

```
/********************************************************************/
/***  fichier: main.c                                             ***/
/***  saisit 2 entiers et affiche leur produit                    ***/
/********************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "produit.h"

int main(void)
{
  int a, b, c;
  scanf("%d",&a);
  scanf("%d",&b);
  c = produit(a,b);
  printf("\nle produit vaut %d\n",c);
  return EXIT_SUCCESS;
}

/********************************************************************/
/***  fichier: produit.h                                          ***/
/***  produit de 2 entiers                                        ***/
/********************************************************************/

int produit(int, int);

int produit(int a, int b)
{
  return(a * b);
}
```

Cette technique permet juste de rendre le code plus lisible, puisque le fichier effectivement compilé (celui produit par le préprocesseur) est unique et contient la totalité du code.

Une méthode beaucoup plus pratique consiste à découper le code en plusieurs fichiers sources que l'on compile séparemment. Cette technique, appelée *compilation séparée*, facilite également le débogage.

7.2 La compilation séparée

Si l'on reprend l'exemple précédent, le programme sera divisé en deux fichiers : `main.c` et `produit.c`. Cette fois-ci, le fichier `produit.c` n'est plus inclus dans le fichier principal. Les deux fichiers seront compilés séparément ; les deux fichiers objets produits par la compilation seront liés lors l'édition de liens. Le détail de la compilation est donc :

```
gcc -c produit.c
```

```
gcc -c main.c
gcc main.o produit.o
```

La succession de ces trois commandes peut également s'écrire

```
gcc produit.c main.c
```

Toutefois, nous avons vu au chapitre 4, page 61, qu'il était risqué d'utiliser une fonction sans l'avoir déclarée. C'est ici le cas, puisque quand il compile le programme `main.c`, le compilateur ne dispose pas de la déclaration de la fonction `produit`. L'option `-Wall` de `gcc` signale

```
main.c:15: warning: implicit declaration of function ‘produit’
```

Il faut donc rajouter cette déclaration dans le corps du programme `main.c`.

7.2.1 Fichier en-tête d'un fichier source

Pour que le programme reste modulaire, on place en fait la déclaration de la fonction `produit` dans un fichier en-tête `produit.h` que l'on inclut dans `main.c` à l'aide de `#include`.

Une règle d'écriture est donc d'associer à chaque fichier source *nom*`.c` un fichier en-tête *nom*`.h` comportant les déclarations des fonctions non locales au fichier *nom*`.c`, (ces fonctions sont appelées *fonctions d'interface*) ainsi que les définitions des constantes symboliques et des macros qui sont partagées par les deux fichiers. Le fichier en-tête *nom*`.h` doit être inclus par la directive `#include` dans tous les fichiers sources qui utilisent une des fonctions définies dans *nom*`.c`, ainsi que dans le fichier *nom*`.c`. Cette dernière inclusion permet au compilateur de vérifier que la définition de la fonction donnée dans *nom*`.c` est compatible avec sa déclaration placée dans *nom*`.h`. C'est exactement la procédure que l'on utilise pour les fonctions de la librairie standard : les fichiers `.h` de la librairie standard sont constitués de déclarations de fonctions et de définitions de constantes symboliques.

Par ailleurs, il faut faire précéder la déclaration de la fonction du mot-clef `extern`, qui signifie que cette fonction est définie dans un autre fichier. Le programme effectuant le produit se décompose donc en trois fichiers de la manière suivante.

```
/**********************************************************************/
/***  fichier: produit.h                                           ***/
/***  en-tete de produit.c                                         ***/
/**********************************************************************/

extern int produit(int, int);
```

```
/**********************************************************************/
/***  fichier: produit.c                                           ***/
/***  produit de 2 entiers                                         ***/
/**********************************************************************/
#include "produit.h"

int produit(int a, int b)
{
  return(a * b);
}
```

```
/*****************************************************************/
/***  fichier: main.c                                          ***/
/***  saisit 2 entiers et affiche leur produit                 ***/
/*****************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "produit.h"

int main(void)
{
  int a, b, c;
  scanf("%d",&a);
  scanf("%d",&b);
  c = produit(a,b);
  printf("\nle produit vaut %d\n",c);
  return EXIT_SUCCESS;
}
```

Une dernière règle consiste à éviter les possibilités de double inclusion de fichiers en-tête. Pour cela, il est recommandé de définir une constante symbolique, habituellement appelée *NOM*_H, au début du fichier *nom*.h dont l'existence est précédemment testée. Si cette constante est définie, c'est que le fichier *nom*.h a déjà été inclus. Dans ce cas, le préprocesseur ne le prend pas en compte. Sinon, on définit la constante et on prend en compte le contenu de *nom*.h. En appliquant cette règle, le fichier `produit.h` de l'exemple précédent devient :

```
/*****************************************************************/
/***  fichier: produit.h                                       ***/
/***  en-tete de produit.c                                     ***/
/*****************************************************************/

#ifndef PRODUIT_H
#define PRODUIT_H

extern int produit(int, int);
#endif /* PRODUIT_H */
```

En résumé, les règles d'écriture sont les suivantes :

- A tout fichier source *nom*.c d'un programme on associe un fichier en-tête *nom*.h qui définit son interface.
- Le fichier *nom*.h se compose :
  - des déclarations des fonctions d'interface (celles qui sont utilisées dans d'autres fichiers sources) ;
  - d'éventuelles définitions de constantes symboliques et de macros ;
  - d'éventuelles directives au préprocesseur (inclusion d'autres fichiers, compilation conditionnelle).

- Le fichier *nom*.c se compose :
  - de variables permanentes, qui ne sont utilisées que dans le fichier *nom*.c ;
  - des fonctions d'interface dont la déclaration se trouve dans *nom*.h ;
  - d'éventuelles fonctions locales à *nom*.c.
- Le fichier *nom*.h est inclus dans le fichier *nom*.c et dans tous les autres fichiers qui font appel à une fonction d'interface définie dans *nom*.c.

Enfin, pour plus de lisibilité, il est recommandé de choisir pour toutes les fonctions d'interface définies dans *nom*.c un identificateur préfixé par le nom du fichier source, du type *nom*_fonction.

7.2.2 Variables partagées

Même si cela doit être évité, il est parfois nécessaire d'utiliser une variable commune à plusieurs fichiers sources. Dans ce cas, il est indispensable que le compilateur comprenne que deux variables portant le même nom mais déclarées dans deux fichiers différents correspondent en fait à un seul objet. Pour cela, la variable doit être déclarée une seule fois de manière classique. Cette déclaration correspond à une définition dans la mesure où le compilateur réserve un espace-mémoire pour cette variable. Dans les autres fichiers qui l'utilisent, il faut faire une référence à cette variable, sous forme d'une déclaration précédée du mot-clef extern. Contrairement aux déclarations classiques, une déclaration précédée de extern ne donne pas lieu à une réservation d'espace mémoire.

Ainsi, pour que les deux fichiers sources main.c et produit.c partagent une variable entière x, on peut définir x dans produit.c sous la forme

```
int x;
```

et y faire référence dans main.c par

```
extern int x;
```

7.3 L'utilitaire make

Losrqu'un programme est fragmenté en plusieurs fichiers sources compilés séparemment, la procédure de compilation peut devenir longue et fastidieuse. Il est alors extrèmement pratique de l'automatiser à l'aide de l'utilitaire make d'Unix. Une bonne utilisation de make permet de réduire le temps de compilation et également de garantir que celle-ci est effectuée correctement.

7.3.1 Principe de base

L'idée principale de make est d'effectuer uniquement les étapes de compilation nécessaires à la création d'un exécutable. Par exemple, si un seul fichier source a été modifié dans un programme composé de plusieurs fichiers, il suffit de recompiler ce fichier et d'effectuer l'édition de liens. Les autres fichiers sources n'ont pas besoin d'être recompilés.

La commande make recherche par défaut dans le répertoire courant un fichier de nom makefile, ou Makefile si elle ne le trouve pas. Ce fichier spécifie les dépendances entre les différents fichiers sources, objets et exécutables. Il est également possible de donner un autre nom au fichier `Makefile`. Dans ce cas, il faut lancer la commande `make` avec l'option `-f` *nom_de_fichier*.

7.3.2 Création d'un `Makefile`

Un fichier `Makefile` est composé d'une liste de règles de dépendance de la forme :

```
cible:  liste de dépendances
<TAB>   commandes UNIX
```

La première ligne spécifie un fichier cible, puis la liste des fichiers dont il dépend (séparés par des espaces). Les lignes suivantes, qui commencent par le caractère `TAB`, indiquent les commandes Unix à exécuter dans le cas où l'un des fichiers de dépendance est plus récent que le fichier cible.

Ainsi, un fichier `Makefile` pour le programme effectuant le produit de deux entiers peut être

```
## Premier exemple de Makefile

prod: produit.c main.c produit.h
	gcc -o prod -O3 produit.c main.c

prod.db: produit.c main.c produit.h
	gcc -o prod.db -g -O3 produit.c main.c
```

L'exécutable `prod` dépend des deux fichiers sources `produit.c` et `main.c`, ainsi que du fichier en-tête `produit.h`. Il résulte de la compilation de ces deux fichiers avec l'option d'optimisation `-O3`. L'exécutable `prod.db` utilisé par le débogueur est, lui, obtenu en compilant ces deux fichiers avec l'option `-g` nécessaire au débogage. Les commentaires sont précédés du caractère `#`.

Pour effectuer la compilation et obtenir un fichier cible, on lance la commande `make` suivie du nom du fichier cible souhaité, ici

```
make prod
```

ou

```
make prod.db
```

Par défaut, si aucun fichier cible n'est spécifié au lancement de `make`, c'est la première cible du fichier `Makefile` qui est prise en compte. Par exemple, si on lance pour la première fois `make`, la commande de compilation est effectuée puisque le fichier exécutable `prod` n'existe pas :

```
% make
gcc -o prod -O3 produit.c main.c
```

Si on lance cette commande une seconde fois sans avoir modifié les fichiers sources, la compilation n'est pas effectuée puisque le fichier `prod` est plus récent que les deux fichiers dont il dépend. On obtient dans ce cas :

```
% make
make: `prod' is up to date.
```

Le `Makefile` précédent n'utilise pas pleinement les fonctionnalités de `make`. En effet, la commande utilisée pour la compilation correspond en fait à trois opérations distinctes : la compilation des fichiers sources `produit.c` et `main.c`, qui produit respectivement les fichiers objets `produit.o` et `main.o`, puis l'édition de liens entre ces deux fichiers objet, qui produit l'exécutable `prod`. Pour utiliser pleinement `make`, il faut distinguer ces trois étapes. Le nouveau fichier `Makefile` devient alors :

```
## Deuxieme exemple de Makefile

prod: produit.o main.o
	gcc -o prod produit.o main.o
main.o: main.c produit.h
	gcc -c -O3 main.c
produit.o: produit.c produit.h
	gcc -c -O3 produit.c
```

Les fichiers objet `main.o` et `produit.o` dépendent respectivement des fichiers sources `main.c` et `produit.c`, et du fichier en-tête `produit.h`. Ils sont obtenus en effectuant la compilation de ces fichiers sources sans édition de liens (option `-c` de `gcc`), et avec l'option d'optimisation `-O3`. Le fichier exécutable `prod` est obtenu en effectuant l'édition de liens des fichiers `produit.o` et `main.o`. Lorsqu'on invoque la commande `make` pour la première fois, les trois étapes de compilation sont effectuées :

```
% make
gcc -c -O3 produit.c
gcc -c -O3 main.c
gcc -o prod produit.o main.o
```

Si l'on modifie le fichier `produit.c`, le fichier `main.o` est encore à jour. Seules deux des trois étapes de compilation sont exécutées :

```
% make
gcc -c -O3 produit.c
gcc -o prod produit.o main.o
```

De la même façon, il convient de détailler les étapes de compilation pour obtenir le fichier exécutable `prod.db` utilisé pour le débogage. Le fichier `Makefile` devient alors :

```
## Deuxieme exemple de Makefile

# Fichier executable prod
prod: produit.o main.o
	gcc -o prod produit.o main.o
main.o: main.c produit.h
	gcc -c -O3 main.c
produit.o: produit.c produit.h
	gcc -c -O3 produit.c

# Fichier executable pour le debuggage prod.db
```

```
prod.db: produit.do main.do
        gcc -o prod.db produit.do main.do
main.do: main.c produit.h
        gcc -o main.do -c -g -O3 main.c
produit.do: produit.c produit.h
        gcc -o produit.do -c -g -O3 produit.c
```

Pour déterminer facilement les dépendances entre les différents fichiers, on peut utiliser l'option `-MM` de `gcc`. Par exemple,

```
% gcc -MM produit.c main.c
produit.o: produit.c produit.h
main.o: main.c produit.h
```

On rajoute habituellement dans un fichier `Makefile` une cible appelée `clean` permettant de détruire tous les fichiers objets et exécutables créés lors de la compilation.

```
clean:
        rm -f prod prod.db *.o *.do
```

La commande `make clean` permet donc de "nettoyer" le répertoire courant. Notons que l'on utilise ici la commande `rm` avec l'option `-f` qui évite l'apparition d'un message d'erreur si le fichier à détruire n'existe pas.

7.3.3 Macros et abbréviations

Pour simplifier l'écriture d'un fichier `Makefile`, on peut utiliser un certain nombre de macros sous la forme

nom_de_macro = corps de la macro

Quand la commande `make` est exécutée, toutes les instances du type `$(`*nom_de_macro*`)` dans le `Makefile` sont remplacées par le corps de la macro. Par exemple, on peut définir une macro `CC` pour spécifier le compilateur utilisé (`cc` ou `gcc`), une macro `PRODUCTFLAGS` pour définir les options de compilation utilisées pour générer un fichier produit, une macro `DEBUGFLAGS` pour les options de compilation utilisées pour générer un fichier produit pour le débogage... Le fichier `Makefile` suivant donne un exemple :

```
## Exemple de Makefile avec macros

# definition du compilateur
CC = gcc
# definition des options de compilation pour obtenir un fichier .o
PRODUCTFLAGS = -c -O3
# definition des options de compilation pour obtenir un fichier .do
DEBUGFLAGS = -c -g -O3

# Fichier executable prod
prod: produit.o main.o
        $(CC) -o prod produit.o main.o
```

```
main.o: main.c produit.h
        $(CC) $(PRODUCTFLAGS) main.c
produit.o: produit.c  produit.h
        $(CC) $(PRODUCTFLAGS)  produit.c

# Fichier executable pour le debuggage prod.db
prod.db: produit.do main.do
        $(CC) -o prod.db produit.do main.do
main.do: main.c produit.h
        $(CC) -o main.do $(DEBUGFLAGS) main.c
produit.do: produit.c produit.h
        $(CC) -o produit.do $(DEBUGFLAGS) produit.c
```

La commande `make` produit alors

```
% make
gcc -c -O3  produit.c
gcc -c -O3 main.c
gcc -o prod produit.o main.o
```

Cette écriture permet de faciliter les modifications du fichier `Makefile` : on peut maintenant aisément changer les options de compilation, le type de compilateur...

Un certain nombre de macros sont prédéfinies. En particulier,

- `$@` désigne le fichier cible courant :
- `$*` désigne le fichier cible courant privé de son suffixe :
- `$<` désigne le fichier qui a provoqué l'action.

Dans le `Makefile` précédent, la partie concernant la production de `main.do` peut s'écrire par exemple

```
main.do: main.c produit.h
        $(CC) -o $@ $(DEBUGFLAGS) $<
```

7.3.4 Règles générales de compilation

Il est également possible de définir dans un `Makefile` des règles générales de compilation correspondant à certains suffixes. On peut spécifier par exemple que tout fichier `.o` est obtenu en compilant le fichier `.c` correspondant avec les options définies par la macro `PRODUCTFLAGS`. Pour cela, il faut tout d'abord définir une liste de suffixes qui spécifient les fichiers cibles construits à partir d'une règle générale. Par exemple, avant de définir des règles de compilation pour obtenir les fichiers `.o` et `.do`, on écrit :

```
.SUFFIXES: .o .do
```

Une règle de compilation est ensuite définie de la façon suivante : on donne le suffixe du fichier que `make` doit chercher, suivi par le suffixe du fichier que `make` doit produire. Ces deux suffixes sont suivis par `:;` puis par une commande Unix (définie de la façon la plus générale possible). Les règles de production des fichiers `.o` et `.do` sont par exemple :

```
# regle de production d'un fichier .o
.c.o:; $(CC) -o $@ $(PRODUCTFLAGS) $<
# regle de production d'un fichier .do
.c.do:; $(CC) -o $@ $(DEBUGFLAGS)  $<
```

Si les fichiers `.o` ou `.do` dépendent également d'autres fichiers, il faut aussi spécifier ces dépendances. Ici, il faut préciser par exemple que ces fichiers dépendent aussi de `produit.h`. Le fichier `Makefile` a donc la forme suivante :

```
## Exemple de Makefile

# definition du compilateur
CC = gcc
# definition des options de compilation pour obtenir un fichier .o
PRODUCTFLAGS = -c -O3
# definition des options de compilation pour obtenir un fichier .do
DEBUGFLAGS = -c -g -O3

# suffixes correspondant a des regles generales
.SUFFIXES: .c .o .do
# regle de production d'un fichier .o
.c.o:; $(CC) -o $@ $(PRODUCTFLAGS) $<
# regle de production d'un fichier .do
.c.do:; $(CC) -o $@ $(DEBUGFLAGS)  $<

# Fichier executable prod
prod: produit.o main.o
        $(CC) -o prod produit.o main.o
produit.o: produit.c produit.h
main.o: main.c produit.h

# Fichier executable pour le debuggage prod.db
prod.db: produit.do main.do
        $(CC) -o prod.db produit.do main.do
produit.do: produit.c produit.h
main.do: main.c produit.h

clean:
        rm -f prod prod.db *.o *.do
```