# Guide pour la conception d'une application en C

Ph. Preux*
DESS IMST, ULCO

Novembre 1999

## 1 Principes généraux

Une application informatique, dès qu'elle dépasse une centaine de lignes de code, doit impérativement être structurée sous peine de se révéler difficile à mettre au point, difficile à modifier, donc inutilisable à terme. Pour cela, une application se découpe à deux niveaux : à l'intérieur d'un module en fonctions et en plusieurs modules. Nous donnons ici quelques indications sur la manière de décomposer une application en plusieurs modules et la gestion de cette application quant à sa compilation.

Tout d'abord, des règles de bons sens doivent être respectées :

1. **une fonction ne doit pas être de longueur trop importante ; du fait de nos capacités psychologiques et de la taille de nos écrans, il est raisonnable que le corps d'une fonction ne dépasse pas largement le nombre de lignes qu'affiche un écran, soit quelques dizaines de lignes ;**
2. **une fonction doit, comme son nom l'indique, avoir une fonction bien précise dans le programme. Une fonction qui fait plusieurs choses différentes, ce n'est pas normal ; si l'on doit faire plusieurs choses, c'est qu'il faut définir une fonction pour chacune ;**
3. **un module (fichier `.c`) ne doit pas être de longueur démesurée lui non plus. 1000 lignes constituent déjà un beau module ; il est beaucoup plus facile de gérer 10 modules de 100 lignes chacun qu'un seul module de 1000 lignes ; on gagnera du temps à tous les niveaux : temps de mise au point, temps de compilation, ...**
4. **un module doit lui-aussi, avoir un rôle précis : il doit regrouper des fonctions correspondant à des fonctionnalités proches. Par exemple, on trouvera souvent un module spécialisé dans les entrées-sorties.**

*philippe.preux@univ-lille3.fr

Pour illustrer ces règles, dans le cadre du TP sur la calculatrice, on devrait avoir défini les modules suivants :

- un module spécialisé dans la gestion de la pile ; nommons-le `pile.c`. Celui-ci contiendrait les fonctions primitives de manipulation d'une pile, telles `initPile()`, `empiler()` et `depiler()` ;
- un module spécialisé dans les fonctions arithmétiques (`arith.c`) ;
- un module contenant les opérations de manipulation des éléments de la pile (`manipulation.c`) telles `swap()`, `dup()`, ...
- un module contenant le programme principal (`main.c`).

## 2 Mise en œuvre

Nous montrons sur le TP concernant la calculatrice comment mettre en œuvre ces principes.

### Étape 1 : les structures de données

Une, ou des, structures de données sont à la base de toute application informatique. Une structure de données s'exprime sous la forme d'un enregistrement (`struct` en C). Ces structures de données doivent être définies dans un fichier en-tête particulier.

Ainsi, la définition des types `Pile` et `PPile` doit être accessible dans différents modules ; pour cela, on la place dans un fichier en-tête ayant une extension `.h`. Ainsi, on créera le fichier `type-pile.h` contenant :

```
/*
 * type-pile.h
 */

#define N 100

typedef struct
  int elt [N];
  int sommet;
 Pile, *PPile;
```

Ce fichier sera alors inclus dans chaque module nécessitant la définition de l'un de ces deux types. Pour cela, on indique en début de module :

```
#include "type-pile.h"
```

### Étape 2 : les fonctions primitives

On réalise ensuite le module contenant les primitives de manipulation des structures de données. Une règle à laquelle il ne faut pas déroger est la suivante : **les champs d'une structure de données ne doivent jamais être accédés directement par une fonction qui n'est pas une primitive**. Autrement dit, **seules les primitives doivent accéder directement aux champs de la structure de données dont elles sont les primitives**. Toutes les autres fonctions utilisant la structure de données doivent l'accéder au travers de ces primitives.

Le respect de cette règle présente l'avantage que l'utilisation de la structure de données (en dehors des primitives) est indépendante de son implantation. Ainsi, le changement de l'implantation de la structure de données n'entraînera que des modifications bien localisées et non pas une modification de l'ensemble de l'application. C'est un grand avantage quant au temps nécessaire et surtout, à la limitation des risques de bugs.

D'une manière générale, le jeu de fonctions primitives doit être bien choisi pour qu'il soit aussi réduit que possible, tout en permettant d'effectuer tous les traitements nécessaires sur la structure de données.

Dans le fichier `pile.c`, on définit donc les fonctions :

- `void empiler (PPile p, int x)`
- `int depiler (PPile p)`
- `void initPile (PPile p)`
- `int sommetPile (PPile p)`
- `int estVide (PPile p)`

À ce fichier, on associe un fichier en-tête (`pile.h`) qui contient les prototypes des fonctions définies dans `pile.c` :

```
/*
 * pile.h
 */

void empiler (PPile, int);
int depiler (PPile);
void initPile (PPile);
int sommetPile (PPile);
int estVide (PPile);
```

### Étape 3 : module de test des fonctions primitives

Enfin, on écrit un programme principal (fonction `main()`) dans un nouveau module (`main.c`) qui va permettre de tester les primitives que nous venons de définir. Puisque `main()` va appeler

les fonctions primitives, il faudra définir celles-ci, du moins leur prototype, en incluant `pile.h` qui nécessite lui-même `type-pile.h`. On fera comme suit :

```
/*
 * main.c
 */

#include "type-pile.h"
#include "pile.h"

main ()

  ...
```

en remplaçant les `...` par des instructions permettant de tester les fonctions primitives de manipulation d'une pile.

### Étape 4 : la compilation séparée

Pour compiler, il faut générer un fichier objet (extension `.o`) pour chaque module. Ensuite, il faut éditer les liens entre ces objets pour obtenir un programme exécutable.

Par exemple, on aura les commandes suivantes :

```
gcc -c pile.c
gcc -c main.c
gcc main.o pile.o -o main
```

Les deux premières invocations de `gcc` génèrent le fichier objet de chacun des modules sources. La troisième entraîne l'édition de liens et la création du fichier exécutable `main`.

D'une manière plus générale, la solution la plus simple pour réaliser la compilation d'un ensemble de modules est d'écrire un fichier `Makefile`. Utilisé par la commande `gmake`, un fichier `Makefile` spécifie les dépendances entre les modules d'une application ; de plus, `gmake` prend garde de ne déclencher que les traitements vraiment nécessaires : si un module source n'a pas été modifié depuis que son objet à été généré, il n'est pas de nouveau compilé. Quand une application compte quelques dizaines de fichiers, on peut ainsi gagner beaucoup de temps.

Sous sa forme la plus simple, un fichier `Makefile` pour la calculatrice aura la forme suivante :

```
# commentaire

main: pile.o main.o
      gcc pile.o main.o -o main

pile.o: pile.c type-pile.h
      gcc -c pile.c

main.o: main.c type-pile.h
      gcc -c main.c
```

Chaque règle est ici composée de deux lignes. Prenons la première règle :

```
main: pile.o main.o
      gcc pile.o main.o -o main
```

Elle indique que le fichier `main` dépend des deux fichiers `pile.o` et `main.o`. Si l'un de ces deux fichiers est plus récent que `main`, alors l'action pour le regénérer est déclenchée. Cette action, la ligne suivante, appelle `gcc` pour éditer les liens entre `pile.o` et `main.o` et créer l'exécutable `main`.

Les deux fichiers `pile.o` et `main.o` dépendent eux-aussi d'autres fichiers ; les deux règles qui suivent l'indiquent et on retrouve une action qui permet de les générer si nécessaire.

`gmake` cherche à engendrer le premier fichier associé à la première règle trouvée dans le fichier `Makefile`. Ici, il cherche donc à générer un fichier `main`. L'ordre dans lequel les règles sont données est donc important, au moins pour la première.

`gmake` est un logiciel sophistiqué dont l'utilisation complète dépasse de loin l'objet de cette note. Voir la documentation adéquate.

### Étape 5 : poursuite du projet

Une fois les primitives mises au point, on peut compléter l'application par la définition des opérations arithmétiques. Les fonctions associées seront placées dans le module `arith.c`, leurs prototypes dans le fichier `arith.h`. La fonction `main()` sera modifiée pour tester ces nouvelles fonctions. Désormais, si l'on a bien pris soin de tester les primitives, si le programme ne fonctionne pas bien, cela ne peut pas venir du module `pile.c`. Mis à part des erreurs triviales dans `main()` qui peuvent survenir, les erreurs seront à chercher dans le nouveau module `arith.c`.

On procédera ensuite de même pour `manipulation.c`.

Il est également classique de définir un module `erreur.c` qui affiche les messages d'erreur et provoque éventuellement l'arrêt du programme. Les autres modules font appel à une fonction de `erreur.c` lorsqu'une condition d'erreur est rencontrée. Ainsi, la gestion des erreurs est centralisée.

Il est beaucoup plus facile d'avoir ainsi des messages d'erreur qui ont un aspect uniforme pour toute l'application. Il est également ainsi beaucoup plus facile de modifier tous les messages d'erreur que de devoir parcourir tous les modules source pour trouver et modifier les messages d'erreur.

## 3 Utilisation de répertoires

Les fichiers constituant une application comprenant plus d'une dizaines de modules sources doivent être rangés dans des répertoires distincts pour ne pas mélanger sources, en-têtes, objets et exécutables. Ainsi, on utilise généralement :

- un répertoire nommé `src` qui contient tous les fichiers source ;
- un répertoire nommé `o` qui contient tous les fichiers objet ;
- un répertoire nommé `h` qui contient tous les fichiers en-tête.

Cela impose de spécifier les répertoires dans le `Makefile` et d'ajouter une option `-I...` à `gcc` pour qu'il trouve les fichiers en-têtes.

Ainsi, pour la calculatrice, on pourrait avoir la répartition suivante des fichiers :

- un répertoire `racine` contenant `Makefile` et `main` ;
- un répertoire `racine/src` contenant tous les modules source ;
- un répertoire `racine/o` contenant tous les fichiers objet ;
- un répertoire `racine/h` contenant tous les fichiers en-tête.

Le fichier `Makefile` pourrait être défini comme suit :

```
# fichier Makefile pour la calculatrice
#

SRC=src/
O=o/
H=h/

main: $(O)pile.o $(O)main.o
      gcc $(O)pile.o $(O)main.o -o main

$(O)pile.o: $(SRC)pile.c $(H)type-pile.h
      gcc -c $(SRC)pile.c -I$(H)

$(O)main.o: $(SRC)main.c $(H)type-pile.h
      gcc -c $(SRC)main.c -I$(H)
```