Programmation système III
Programmation Modulaire

DUT 1re année

Université de Marne La vallée

Difficultés des grands projets

Plus les projets sont ambitieux, plus les difficultés sont grandes.

- TP d'étudiant
  (1 personne, 1-3 fichiers)
- Projet d'étudiant
  (1-3 personnes, 5-10 fichiers)
- Projet PME
  (1-5 développeurs, 10-100 fichiers)
- Gros projet (logiciel, bibliothèque, librairie)
  (1-* développeurs, $>$ 100 k lignes de code)

Problèmes : Division du travail, partage du code, ré-utilisation du code d'autrui, duplication de code, cahier des charges des fonctionnalités, documentation, tests, robustesse, correction des bogues...

Idée : diviser un projet

Morceler un projet facilite son développement et les modifications.

- Division du travail : division du projet en morceaux
- partage du code : partage des morceaux (plus petit pour les transferts)
- ré-utilisation du code d'autrui : rajouter un morceau au tout
- duplication de code : multiples utilisations des morceaux
- cahier des charges des fonctionnalités : plusieurs cahiers des charges plus simples (un par morceau)
- documentation : documentation locale sur chaque morceau
- tests : tests indépendants pour chaque morceau
- robustesse : lors d'un bogue, on ne remplace que le morceau incriminé
- correction des bogues : correction locale des bogues (seul le morceau qui a le bogue est corrigé)

Un projet : Réaliser un répertoire

Ce cours sera illustré avec, pour exemple récurrent, l'implantation d'un répertoire en langage C (cf. l'énoncé de votre TP numéro 2).

Un projet : Réaliser un répertoire

Fonctionnalités voulues :

- Charger un répertoire mémorisé dans un fichier.
- Rajouter une fiche dans le répertoire ouvert courant.
- Sauvegarder un répertoire dans un fichier.
- Faire une recherche dans le répertoire
- Trier le répertoire.
- Obtenir diverses informations :
  - nombre de personnes dans un groupe
  - prochain anniversaire à souhaiter dans le répertoire
  - nombre de groupes dans lesquels une personne est rattachée

Module de code : Définition (idée)

On fractionne le projet en différents fichiers qui rassemblent des fonctionnalités ayant un point commun :

- Des fonctions manipulant une même structure
- Des fonctions rassemblant un certain type de fonctionnalités
  (fonctions graphiques, fonctions d'entrées/sorties, fonctions d'opérations mathématiques(math.h), fonctions manipulant des chaînes de caractères(string.h), ...)
- Des fonctions techniques et outils (fonctions de tri, gestion de file d'attente, gestion de listes chaînées, ...)
- ...

Chaque morceau est appelé module. Les modules peuvent avoir des liens de dépendances les uns aux autres ou bien être complètement indépendants.

Règles de découpage

Il n'existe pas de règles/méthodes générales pour découper un projet !!!

Le problème de découpage est un problème de sémantique. Même si des détails de programmation peuvent apparaître, c'est d'abord la compréhension du produit fini désiré qui motive le choix dans la fragmentation.

Démarche chronologique :

- Imagination du produit fini
- fonctionnalités voulues
- division de ces fonctionnalités désirées en plusieurs paquets
- implantation des différents modules
- assemblage et compilation
- produit fini

Exemple récurrent de modules

De nombreux projets incluent des modules de type suivant :

- Interface graphique
- Module d'entrée/sortie
- Module de tests
- Gestion de pile
- Gestion de liste chaînée
- Gestion de table de Hachage
- Module de calcul
- Module misc ou outil (fonctions utiles mais ne se rattachant pas à un module en particulier)
- ...

Un projet : Réaliser un répertoire

Suivant les fonctionnalités voulues et les structures :

- `date` : module gérant les dates et leurs fonctionnalités
- `fiche` : module gérant les fiches et leurs fonctionnalités
- `repertoire` : module gérant les répertoires et leurs fonctionnalités
- `in_out_repertoire` : module gérant les entrées/sorties (chargements/sauvegardes)
- `tri` : module pour trier
- `groupe` : module pour gérer les groupes

Module de code : Définition (programmation)

Un module de code nommé `foo` est une association de deux fichiers :

- Un fichiers d'en-tête : `foo.h`
- Un fichiers de code : `foo.c`

Le fichier d'en-tête est TOUJOURS inclus (`#include "foo.h"`) dans le fichier de code `foo.c`.

Les prototypes des fonctions de `foo.c` apparaissent dans le fichier `foo.h`.

`foo.h` ne contient pas de code exécutable.

Les structures sont toutes définies dans le fichier `foo.h`.

On tente, autant que faire se peut, d'importer les bibliothèques standards dans le fichier `foo.c` sauf quand ce n'est pas possible.

Code d'un module "foo"

Pour le fichier `foo.h`, il apparaît toujours dans l'ordre suivant les déclaration qui suivent :

- Macro de Sécurité pour les inclusions cycliques
- Inclusions des dépendances
- Définition des structures
- Prototypes des fonctions de foo.c
- Fin de sécurisation

Code d'un module "foo"

Pour le fichier `foo.h`

```
1  #ifndef __FOO__
2  #define __FOO__
3
4  #include "autre_module.h"
5  #include "troisieme_module.h"
6
7  typedef struct bla{
8    int bar;
9    float blo;
10 }Bla;
11
12 int une_fonction(Bla a, int b);
13 void autre_fonction(Bla a, Bla b);
14
15 #endif
```

Code d'un module "foo"

Pour le fichier `foo.c`

```
1  #include "foo.h"
2  #include <bliotheque.h>
3
4  int une_fonction(Bla a, int b){
5  /*
6  *   VRAI CODE DE LA FONCTION
7  */
8  }
9
10 void autre_fonction(Bla a, Bla b){
11 /*
12 *   VRAI CODE DE LA FONCTION
13 */
14 }
```

Un projet : Réaliser un répertoire

Pour le fichier `date.h`

```
1  #ifndef __DATE__
2  #define __DATE__
3
4  typedef struct date{
5     int jour;
6     int mois;
7     int annee;
8  }Date;
9
10 void affiche_date(Date *d);
11 void saisir_date(Date *d);
12 int meme_mois(Date *un, Date *deux);
13
14 #endif
```

Un projet : Réaliser un répertoire

Pour le fichier `fiche.h`

```
1  #ifndef __FICHE__
2  #define __FICHE__
3
4  #include "date.h"
5
6  #define MAXCHAINE 50
7  #define MAXGROUPE 10
8
9  typedef struct fiche{
10   char nom[MAXCHAINE];
11   char prenom[MAXCHAINE];
12   char genre;
13   Date naissance;
14   int groupe[MAXGROUPE];
15 }Fiche;
16
17 void affiche_fiche(Fiche *f);
18 void saisir_fiche(Fiche *f);
19 int nombre_groupe(Fiche *f);
20
21 #endif
```

Notion de dépendance

Certains modules dépendent d'autres modules.

Lorsqu'un module A dépend d'un module B, l'information est donnée au module A via la présence dans son fichier d'en-tête A.h de la directive au préprocesseur :

```
#include "B.h"
```

Phénomènes classiques générant des dépendances :

- une fonction est utilisée dans A mais implantée dans B
- une structure de A utilisant une structure définie dans B
- une fonction de A utilise une structure définie dans B
- ...

Notion de dépendance

Toutes les règles de dépendances réunies permettent de définir un graphe qui caractérise le découpage du projet.

Chaque module ainsi que le fichier `main.c` forme un sommet dans le graphe. Pour chaque couple (A, B) de modules, on place une flèche de B vers A lorsque A dépend de B (A.h inclut B.h).

Lorsqu'un fichier source en `.c` inclut un autre fichier, on ne parle pas de dépendance (sauf pour le `main.c`).

Graphe de dépendance

Tout graphe de dépendance est envisageable !!! L'inclure dans la documentation lorsque le projet est supposé être transmis à un autre développeur est une bonne chose.

Une seule règle pour les graphes de dépendances:

UN GRAPHE DE DÉPENDANCE NE DOIT JAMAIS COMPORTER DE CYCLE !!!

Un cycle ne signifie pas forcément que l'exécutable ne fonctionnera pas ou que la compilation ne marchera pas. La présence d'un cycle montre qu'un projet est mal conçu !

Le fichier `main.c`

Chaque projet (ensemble de module) s'assemble avec l'ajout d'un dernier fichier `main.c`.

Ce fichier fait le lien entre toutes les fonctionnalités et réalise la répartition des tâches suivant les paramètres donnés en argument à l'exécutable. C'est donc le SEUL fichier contenant une fonction `main`.

Deux stratégies sont possibles :

- Soit un main à paramètres classique : `int main(int argc, char *argv[])`
- Soit un main sans argument mais qui affiche un menu et attend le choix de l'utilisateur : `int main(void)`

Le fichier `main.c`

Le fichier `main.c` n'est pas un module à proprement parlé (notamment, il ne lui correspond pas de `main.h` normalement). Toutefois, on le rattache dans le graphe de dépendance.

Il se peut que le `main` dépende de tous les modules (le fichier inclut alors les en-têtes de chaque autre module).

Dans un projet bien conçu, la modification d'un module n'impose pas de changement dans le `main`. C'est un but de la programmation modulaire.

Un projet : Réaliser un répertoire

Un menu possible pour le `main.c` :

```
nicolas@lancelot:~$ ./repertoire
Bienvenu dans le projet repertoire :
1 - Charger un répertoire
2 - Ajouter une fiche
3 - Sauvegarder le répertoire courant
4 - Faire une recherche sur le répertoire
Votre choix :
?
```

Compilation séparée

Les projets subdivisés en module se compilent de manière séparée.

On compile chaque module de manière indépendante avec la commande `gcc -c` (qui génère un fichier objet contenant du code machine pas encore exécutable à lui seul).
On compile enfin tous les fichiers objets avec la commande `gcc -o nom_exécutable ...` (phase d'assemblage des morceaux et génération de l'exécutable).

```
gcc -c A.c
gcc -c B.c
gcc -c C.c
gcc -c D.c
gcc -c main.c
gcc -o projet main.o A.o B.o C.o D.o
```

Fichier Makefile

Un fichier Makefile donne une liste d'objets à fabriquer et les règles pour fabriquer chacun de ces objets.

```
1 A.o : A.c A.h B.h D.h
2         gcc -c A.c -Wall -ansi
```

```
objet.o : liste de dependances
        gcc -c fichier_de_code -Wall -ansi
```

Fichier Makefile

Structure d'un Makefile classique :

A dépend de B et C, B dépend de C et main dépend de A.

```
1  exe : main.o A.o B.o C.o
2          gcc -o exe main.o A.o B.o C.o -Wall -ansi
3
4  A.o : A.c A.h B.h C.h
5          gcc -c A.c -Wall -ansi
6
7  B.o : B.c B.h C.h
8          gcc -c B.c -Wall -ansi
9
10 C.o : C.c C.h
11         gcc -c C.c -Wall -ansi
12
13 main.o : main.c A.h
14         gcc -c main.c -Wall -ansi
15
16 clean :
17         rm *.o
18         rm exe
```

Fichier Makefile avec variables

Structure d'un Makefile classique :
A dépend de B et C, B dépend de C et main dépend de A.

```
1  CC=gcc
2  CFLAGS=-Wall -ansi
3  OBJ=main.o A.o B.o C.o
4  exe : $(OBJ)
5          $(CC) -o exe $(OBJ) $(CFLAGS)
6
7  A.o : A.c A.h B.h C.h
8          $(CC) -c A.c $(CFLAGS)
9
10 B.o : B.c B.h C.h
11         $(CC) -c B.c $(CFLAGS)
12
13 C.o : C.c C.h
14         $(CC) -c C.c $(CFLAGS)
15
16 main.o : main.c A.h
17         $(CC) -c main.c $(CFLAGS)
```

Un projet : Réaliser un répertoire

```
1  CC=gcc
2  CFLAGS=-Wall -ansi
3  OBJ=main.o date.o fiche.o repertoire.o in_out.o tri.o groupe.o
4
5  repertoire : $(OBJ)
6        $(CC) -o repertoire $(OBJ) $(CFLAGS)
7  main.o : main.c in_out.h repertoire.h fiche.h
8        $(CC) -c main.c $(CFLAGS)
9  date.o : date.c date.h
10       $(CC) -c date.c $(CFLAGS)
11 fiche.o : fiche.c fiche.h date.h groupe.h
12       $(CC) -c fiche.c $(CFLAGS)
13 repertoire.o : repertoire.c repertoire.h fiche.h tri.h
14       $(CC) -c repertoire.c $(CFLAGS)
15 in_out.o : in_out.c in_out.h repertoire.h
16       $(CC) -c in_out.c $(CFLAGS)
17 tri.o : tri.c tri.h
18       $(CC) -c tri.c $(CFLAGS)
19 groupe.o : groupe.c groupe.h
20       $(CC) -c groupe.c $(CFLAGS)
```