Programmation impérative

ENSIIE

Semestre 1 — 2023–24

Compilation séparée et Modularité

Modularité

GCC : $>$ 14,5 millions de lignes de code
Noyau Linux : $>$ 19 millions de lignes de code
Windows Vista : 50 millions de lignes de code

- besoin de partage des tâches entre développeurs
- besoin de maintenance
- besoin de réutilisabilité

Fonction (rappel)

Grain le plus fin de modularité :

- fonction/procédure

Entités indépendantes

Lien avec le reste du code :

- Prototype + contrat (requires/assigns/ensures)

Modules

Grain plus gros

Regroupe

- Structures de données
- Fonctions sur celles-ci

Lien avec le reste du code = interface

- indique les fonctionnalités proposées
- = contrat

En dehors du module, seul ce qui est déclaré dans l'interface peut être utilisé (boîte noire)

Interface

Contient :

- Définitions de types concrets
- Déclarations de types abstraits
- Déclarations de prototypes de fonctions

Les modules extérieurs utilisent ces types et ces fonctions, et uniquement ceux-là, pour pouvoir utiliser le module.

Pour les types abstraits, ils ne peuvent les créer/manipuler qu'à travers les fonctions proposées.

Compilation séparée

Chaque module peut être compilé indépendamment des autres modules

- puisque seule leurs interfaces l'intéresse
- il n'est donc pas nécessaire de tout recompiler à chaque modification
- ni d'attendre qu'un module soit complètement implémenté pour compiler un autre

Une fois chaque module compilé, l'éditeur de liens (*linker*) se charge de « coller » les morceaux les uns avec les autres.

Éventuellement, l'éditeur de lien peut combiner des programmes écrits dans différents langages

Interfaces en C

En C, pas de mécanisme spécifique du langage pour les interfaces, mais pratique standard :

- L'interface est écrite dans un fichier `truc.h`
  - elle contient des prototypes de fonctions
  - des définitions de type (éventuellement abstraite)

```
typedef struct type_concret {
  int x;
  double y;
} type_concret;

typedef void* type_abstrait;

int une_fonction(char, type_abstrait);
void une_autre(type_concret*,int);
```

Implémentation

- Le contenu du module est écrit dans un fichier `truc.c`
  - il inclut l'interface (`#include "truc.h"`)
  - il définit le corps des fonctions et les types restés abstraits dans l'interface
  - éventuellement il définit d'autre fonctions (non accessibles en dehors du module)

```
struct concretisation {  type_concret a;
                         char b;  };

int une_fonction(char c, type_abstrait d) {
  struct concretisation *c = d; ... }
void une_autre(type_concret* tp, int i) { ... }
type_concret fonction_locale(double l) { ... }
```

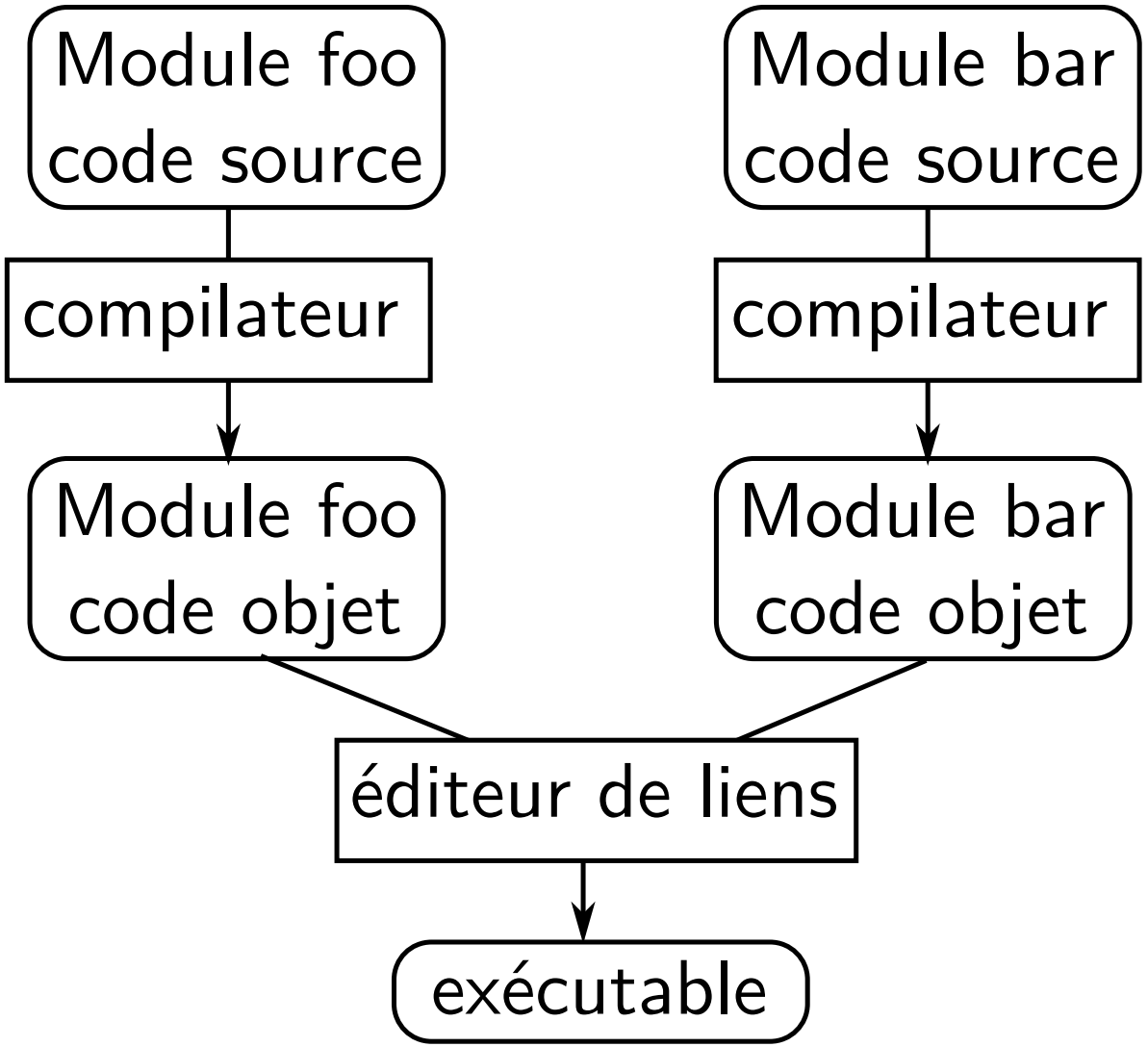
Utilisation

- un autre module qui a besoin de `truc` a uniquement besoin d'inclure l'interface (`#include "truc.h"`)
- il ne peut donc accéder qu'à ce qui est déclaré/défini dans l'interface

```
#include "truc.h"

void machin(type_abstrait x, type_concret* y)
  une_autre(y, une_fonction('z', x));
  /* x->a INTERDIT */
  /* fonction_locale(x) INTERDIT */
}
```

Chaîne de compilation



Chaîne de compilation : C

- source `foo.c` → code objet `foo.o` (code natif)

  `gcc -Wall -c foo.c`

- code objets `foo.o bar.o` → exécutable

  `gcc -Wall foo.o bar.o`

  produit un exécutable `a.out`, si on veut un autre nom : `-o mon_executable`

Il doit y avoir une et une seule fonction `main` dans l'ensemble des modules, qui est appelée au lancement de l'exécutable.

L'ordre des modules n'importe pas.

Exemple

Affichage en base 2 et piles

Modularité

Permet :

- séparation des tâches
- création d'un espace de nom
- réutilisabilité (par exemple, bibliothèques)
- encapsulation
- abstraction (en particulier des types)
- maintenabilité

Séparation des tâches

- Chaque module peut être implémenté par un développeur différent
- Le seul point sur lequel ils doivent s'entendre est l'interface des modules
  (et la sémantique des fonctions...)

Création d'un espace de nom

Dans certains langages (pas en C), chaque module crée un nouvel espace de nom :

- On peut donner le même nom de fonction dans des modules différents
  ex. en Python : `array.fromstring`, `numpy.fromstring`, `...`

Réutilisabilité

Chaque module est indépendant, il peut donc être réutilisé dans un autre projet sans avoir à tout reprendre.
Ex. : Structure de donnée pour stocker des chaînes de caractères
Permet de ne pas réinventer la roue

On peut regrouper des modules pour créer des bibliothèques
En particulier, on dispose en général pour chaque langage d'une bibliothèque standard

- pour C, définie dans le standard C ANSI, implémentée par exemple par la GNU C Library

Encapsulation

- Seules les fonctions déclarées dans l'interface permettent d'accéder aux objets
- Le développeur n'a pas à se soucier d'objets mal formés créés à l'extérieur du module
- Permet de maintenir des invariants (via des constructeurs intelligents par exemple)

Abstraction

Les structures de données utilisées pour implémenter tel ou tel objet peuvent rester abstraites

- définition de type abstrait dans l'interface

  En C : `typedef void *type_abstrait` dans le `.h`
  la structure de donnée pointée dans l'implémentation n'est pas définie dans l'interface, elle n'est donc pas visible de l'extérieur

Pas besoin de connaître l'implémentation concrète depuis l'extérieur

Maintenabilité

Permet de pouvoir changer l'implémentation concrète sans avoir à changer tout le code

Exemple : utilisation de tableaux au lieu de listes pour implémenter des piles

Exemple : changement du tri par bulle (complexité $O(n^2)$) en tri fusion ($O(n \log n)$)

Invisible depuis les autres modules

Extensions

Suivant les langages de programmation, modules plus avancés :

- modules imbriqués
- modules paramétrés par des modules (foncteurs)
  - généricité
- modules de première classe : peuvent être manipulés comme des valeurs du langage

Dépendances

Un module dépend de son implémentation, mais également des interfaces des modules qu'il utilise

- si celles-ci changent, il faut le recompiler

Par contre, il ne dépend pas de l'implémentation des modules qu'il utilise

- même si celle-ci change, il n'est pas nécessaire de le recompiler

Il peut être difficile de savoir quel fichier a besoin d'être recompilé après un changement

- Utilisation d'un `Makefile` pour gérer automatiquement les dépendances

Makefile

Un fichier `Makefile` est une suite de règles

```
cible: dependance1 dependance2 ... dependancer
_____commande_pour_creer_cible
```

(attention, tabulation au début de la deuxième ligne)

- `cible` : fichier à créer
- `dependancei` : fichiers dont `cible` dépend
- `commande_pour_creer_cible;` optionnel
  peut utiliser les raccourcis suivants :
  - `$@` nom de la cible
  - `$^` dépendances
  - `$<` première dépendance

Utilisation d'un Makefile

Quand on appelle `make cible` dans le shell, on vérifie récursivement que les dépendances n'ont pas besoin d'être recompilées, puis on appelle `commande_pour_creer_cible` si une des dépendances est plus récente que `cible` (ou si `cible` n'existe pas).

Si `make` est appelé sans argument, utilise la première règle.

- Convention : première règle

  ```
  all: executable1 executable2
  ```

  sans commande de production (pas de création de `all`, donc règle toujours active)

Règle avec filtrage

```
%.o: %.c
␣␣␣␣␣␣␣␣␣gcc -Wall -Wextra -ansi -c $<
```

Attention, spécifique GNU make, sinon :

```
.SUFFIXES: .c .o
```

```
.c.o:
␣␣␣␣␣␣␣␣␣gcc -Wall -Wextra -ansi -c $<
```

Cette règle (`.c` en `.o`) est ajoutée par défaut.

Variables

```
CC=gcc -Wall -Wextra -ansi

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

OBJ=dep1.o dep2.o

prog: $(OBJ)
	$(CC) -o $@ $^
```

CC définie par défaut à cc

Makefile générique

Au final, entre les règles et les variables définies par défaut :

```
CC=gcc -Wall -Wextra -ansi
OBJ=module1.o module2.o ... modulen.o

all: prog

module1.o: modulei.h modulej.h
module2.o: modulek.h
...

prog: $(OBJ)
	$(CC) -o $@ $^
```

Autres systèmes de construction

Ninja, CMake, Dune (OCaml), Ant (Java), ...

Exemple : Meson, `meson.build`

```
project('my_project', 'c')
common_src = ['file1.c', 'file2.c', 'file3.c']
executable('executable1', ['file4.c'] + common_src)
executable('executable2', ['file5.c', 'file6.c']
                          + common_src)
```

Aussi systèmes de construction inclus dans les environnements de développement intégrés (e.g. Eclipse)