Compilation séparée

Modularité

GCC : $\sim$4 millions de lignes de code
Noyau Linux : $\sim$12 millions de lignes de code

- besoin de partage des tâches entre développeurs

On sépare les différentes composantes d'un logiciel dans des modules

Chaque module possède une interface qui indique quelles sont les fonctionnalités qu'il propose

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

Interface

Contrat qui indique les fonctionnalités fournies par un module :

- définition de types (éventuellement abstraits)
- prototypes de fonctions

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

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, il n'y a pas de mécanisme spécifique pour les interfaces.
Néanmoins, la pratique standard est la suivante :

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

    ```
    typedef struct foo* t;
    int bidule(char, t);
    ```
- 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

    ```
    struct foo { ... };
    int bidule(char c, t bar) { ... }
    ```
- un autre module qui a besoin de `truc` a uniquement besoin d'inclure l'interface (`#include "truc.h"`)

Interfaces en OCaml

Pour un module `truc.ml` l'interface est définie dans un fichier `truc.mli` compilée ensuite en `truc.cmi`. Elle contient :

- des définitions de type (éventuellement abstraits)

  ```
  type s = A | B of int
  type t
  ```

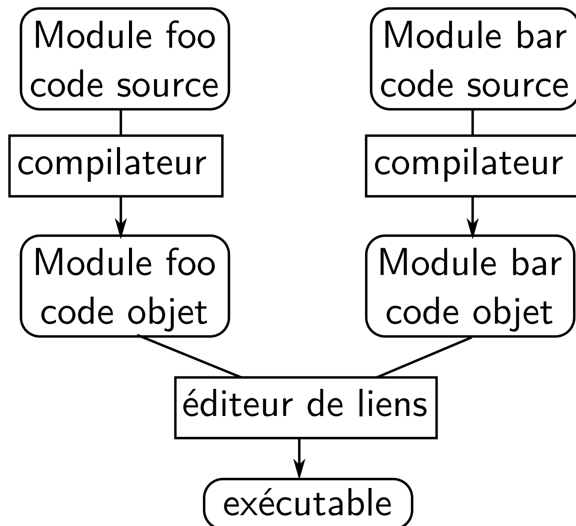
- des prototypes de fonctions

  ```
  val bidule : char -> t -> int
  ```

L'implémentation `truc.ml` doit déclarer des types concrets et des fonctions dont le type est plus général que celui de l'interface.

Un autre module qui a besoin de `truc` peut soit donner explicitement le nom du module (ex : `Truc.bidule`) ou soit ouvrir le module avant utilisation (`open Truc`).

Chaîne de compilation



Chaîne de compilation : C

- source `foo.c` $\rightarrow$ code objet `foo.o` (code natif)

  `gcc -Wall -ansi -c foo.c`

- code objets `foo.o bar.o` $\rightarrow$ exécutable `prog`

  `gcc -Wall -ansi -o prog foo.o bar.o`

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

Chaîne de compilation : OCaml

- source `foo.ml` → code objet `foo.cmo` (bytecode)

  ```
  ocamlc -c foo.ml
  ```

- code objets `foo.cmo bar.cmo` → exécutable `prog`

  ```
  ocamlc -o prog foo.cmo bar.cmo
  ```

Toutes les phrases des modules sont exécutées, l'ordre des modules lors de l'édition de lien est important

Il existe également pour la plupart des architectures une compilation vers du code natif, plus performant :

```
ocamlopt -c foo.ml
ocamlopt -o prog foo.cmx bar.cmx
```

Exemple

Affichage en base 2 et piles

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 ... dependancen
	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 -ansi -c $<
```

Attention, spécifique GNU make, sinon :

```
.SUFFIXES: .c .o

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

Variables

```
CC=gcc -Wall -ansi

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

OBJ=dep1.o dep2.o

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

Pour OCaml

Penser à compiler les interfaces :

```
%.cmo: %.ml
        ocamlc -c $<

%.cmi: %.mli
        ocamlc -c $<

base_2.cmo: pile.cmi
pile.cmo: pile.cmi
```

Outil `ocamldep` pour générer automatiquement les dépendances

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, chaque module crée un nouvel espace de nom :

- On peut donner le même nom de fonction dans des modules différents
  ex. en OCaml : `List.length` `Array.length` `String.length`
- Pour accèder à un fonction, on utilise son nom complet
  ex. en OCaml : `Nom_du_module.nom_de_la_fonction`
- En général, on peut ouvrir les modules pour accèder aux fonctions qu'il contient sans avoir à rappeler le nom du module

Ex. en OCaml :

```
open String
open Array

let comp l a = List.length l < length a
(* val comp : 'a list -> 'b array -> bool = <fun> *)
```

Aparté : surcharge VS polymorphisme

- surcharge : même nom de fonction pour deux algorithmes différents

  Exemple : + en C sur les `int` et les `double`

- polymorphisme : fonction générique qui peut s'appliquer à n'importe quel type de données

  Exemple : `fun x -> x` en OCaml

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
- en OCaml, bibliothèque standard fournie avec le langage (`List`, `Array`, `String`, `Map`, `Set`, `...`)

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 struct contenu_type* type_abstrait`
  dans le `.h`
  la structure `struct contenu_type` n'est pas définie dans l'interface, elle n'est donc pas visible de l'extérieur

  En OCaml : `type type_abstrait` dans le `.mli`

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