Décomposition modulaire

Objectifs: maîtriser un projet

- Présenter les autres grands principes
- Définir la notion d'architecture logicielle
- Identifier et minimiser les dépendances entre modules

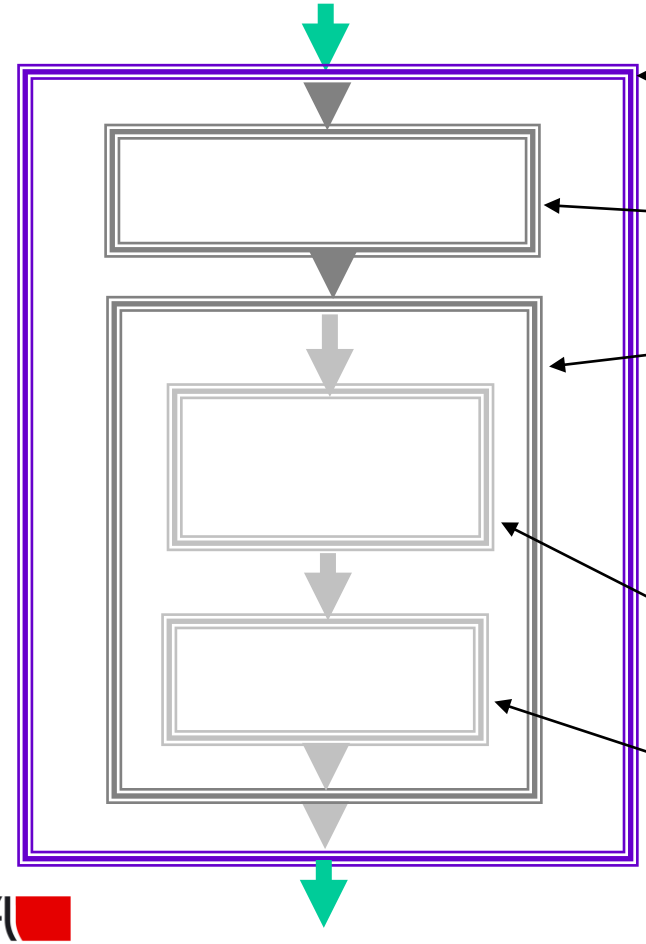
Plan:

- Rappels et nouveaux principes
- un module = 2 fichiers = interface + implementation
- du graphe des appels à l'architecture logicielle
- de l'architecture au graphe des dépendances
- la commande **make** et le fichier **Makefile**

Rappel sur les fonctions: Principe d'Abstraction

Approche top-down

1) ***Principe d'Abstraction***: présenter l'idée générale de la solution (aux niveaux supérieurs) sans se perdre dans les détails



```
main()
{
   tache_A();
   tache_B();
}
```

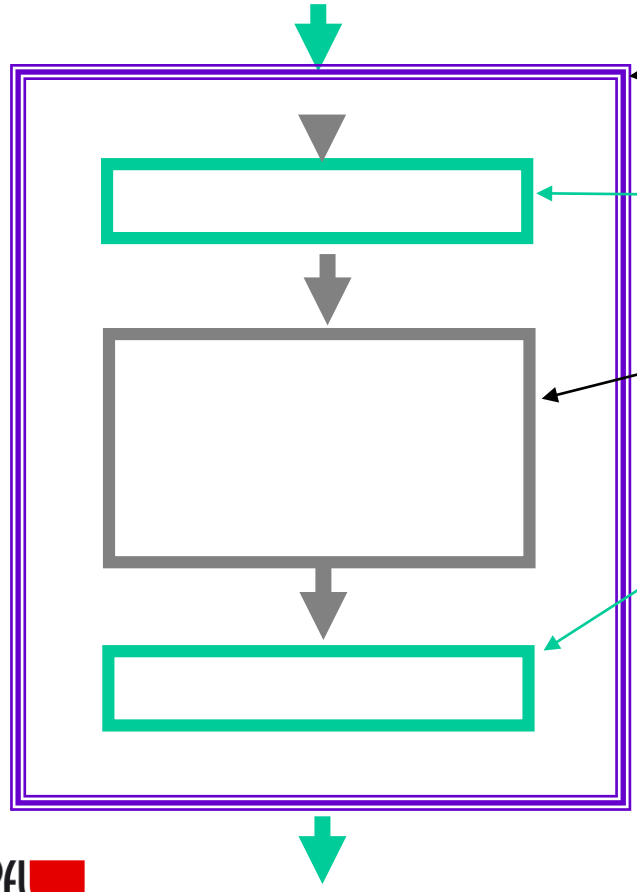
```
tache_A(){...}
```

```
tache_B()
{
   action_1();
   action_2();
}
```

Rappel sur les fonctions : Principe de Ré-utilisation

Approche bottom-up: ne pas ré-inventer la roue

*2) **Principe de Ré-utilisation** pour réduire l'effort de mise au point et la taille du code en **ré-utilisant du code***



```
main()
{
   tache_C(...);
   tache_D();
   tache_C(...);
}
```

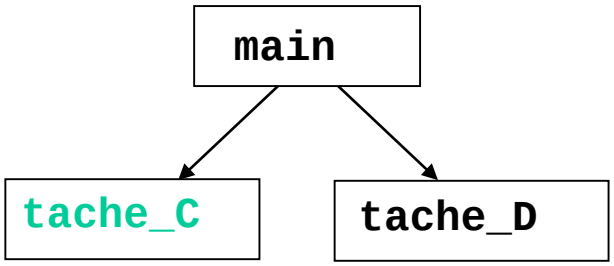
```
tache_C(paramètres)
{
   le traitement dépend
   des paramètres
}
```

Le graphe des appels de fonctions

Offre une vue synthétique des dépendances entre fonctions:

- chaque fonction n'apparait qu'une seule fois
- les fonction standards ne sont pas indiquées (lisibilité)
- les fonctions sont généralement organisées en couches:
  - la fonction appelante est au dessus de la fonction appelée
  - structure de graphe orienté :
    - une fonction = un nœud, une flèche = une dépendance appelant/appelé

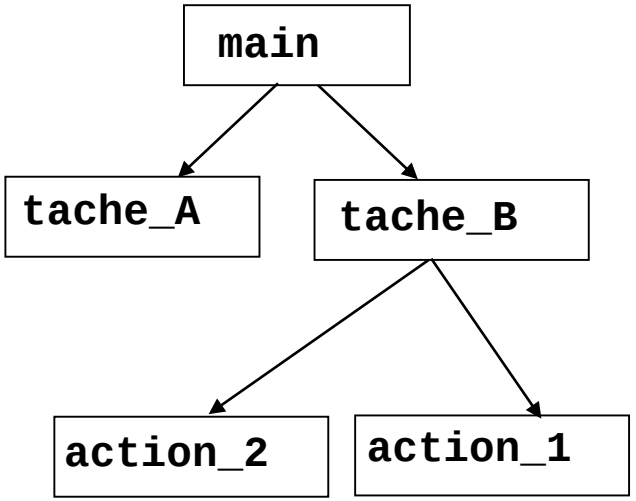
```
main()
{
   tache_C(...);
   tache_D();
   tache_C(...);
}
```




```
main()
{
   tache_A();
   tache_B();
}
```

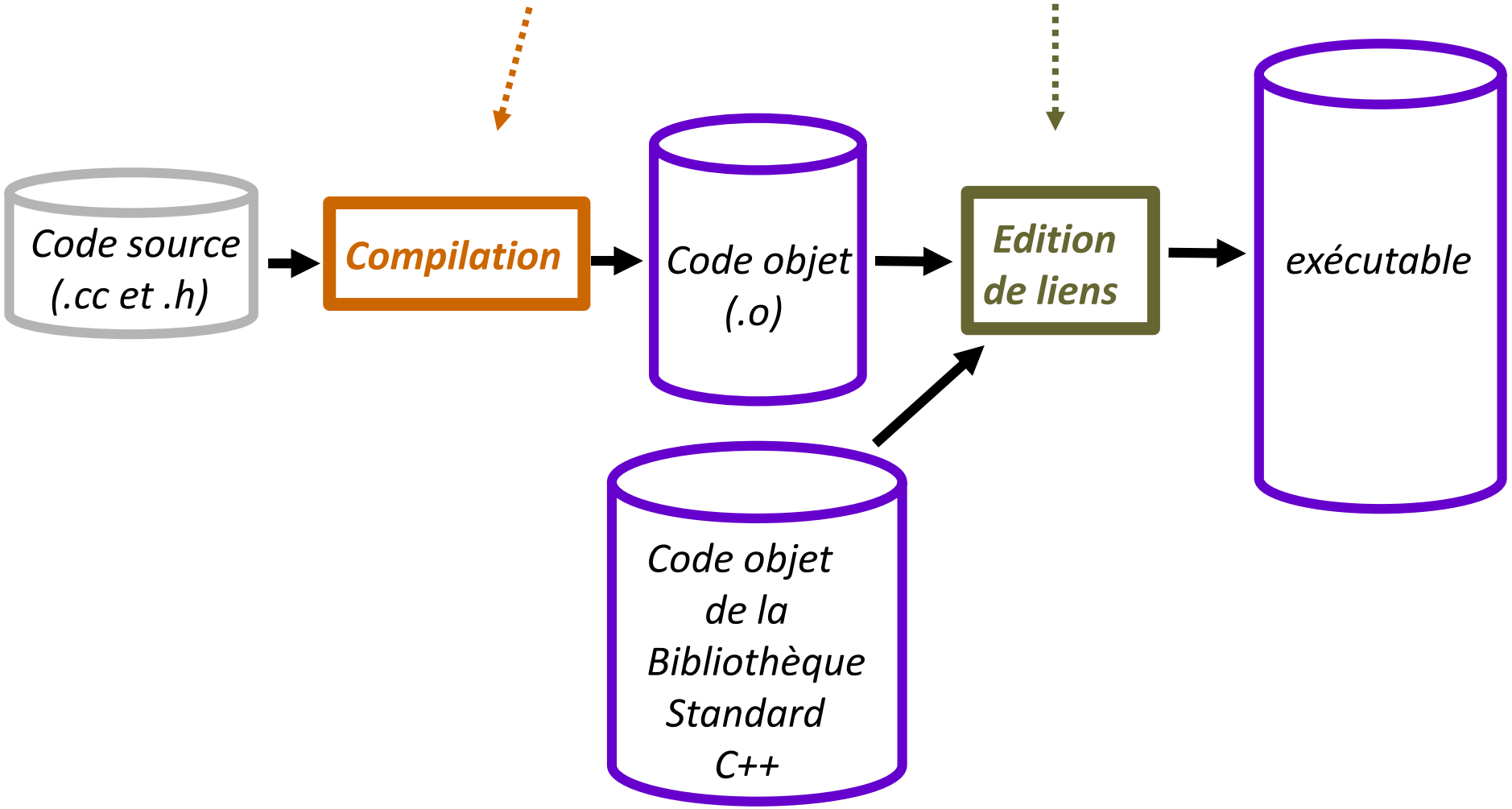
```
tache_A(){...}
```

```
tache_B()
{
   action_1();
   action_2();
}
```

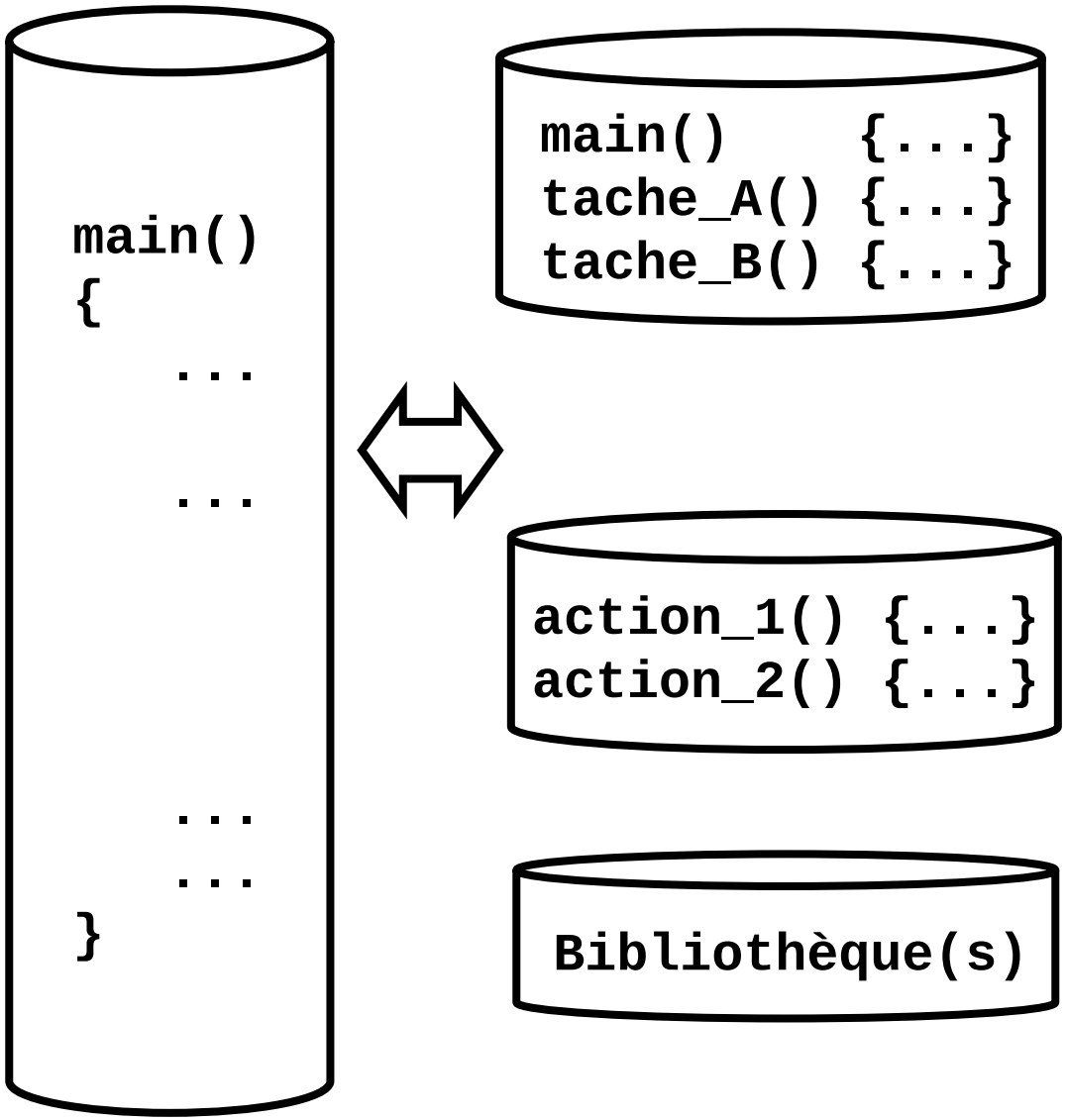


RAPPEL sur la production d'un exécutabe

*2 Etapes: la **compilation** et **l'édition de liens***



Cours d'aujourd'hui: programmation modulaire
*=> décomposition en fonctions **regroupées en modules***



principes pratique

Abstraction

Réutilisation

Séparation des fonctionalités

Encapsulation

Concentration des dépendances

écriture et tests séparés

Décomposition modulaire d'un projet

Pourquoi créer un module ?

Principes:

Abstraction : offrir une vue générale claire, déléguer les sous-problèmes

Ré-utilisation dans d'autre programmes

- fonctions utilitaires (ex: math) ou associées à une structure de données

Séparation des fonctionalités (*Separation of Concerns*)

- offrir des unités logicielles cohérentes (module ou groupe de modules) dédiées à une tâche ou un **type de données**

-> Les responsabilités sont clairement partagées entre modules.

Encapsulation de type « Boîte Noire » (*Information Hiding*)

- minimiser les dépendances entre modules (*type opaque*)

Concentration des dépendances vis-à-vis de bibliothèques externes

- exemple projet: vis à vis de la bliothèque graphique

C'est quoi au juste un module ?

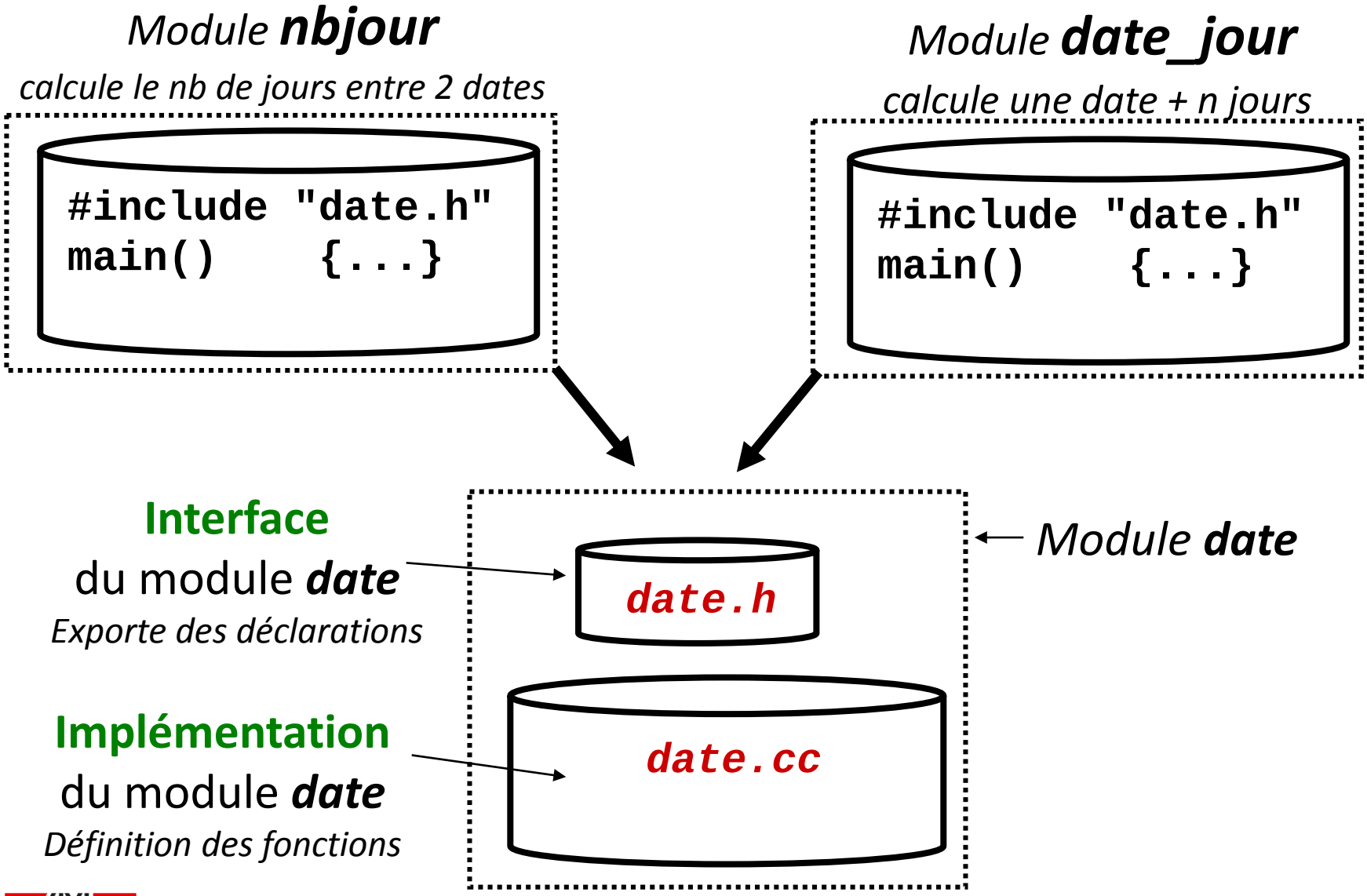
un module = une interface + une implémentation

Définitions: un **module** est composé de deux fichiers sources :

- Son fichier d' **interface** décrit son BUT ; il contient essentiellement les prototypes des fonctions *exportées* et documentées dans le fichier en-tête **(date.h)**.
  - Ces fonctions peuvent être appelées dans d'autres modules, il faut et il suffit d'une directive **include** pour inclure cette interface.

- Son fichier d' **implémentation** est le code source définissant COMMENT les fonctions du module sont mises en œuvre (**date.cc**).
  - Une même interface **(.h)** peut avoir des implémentations **(.cc)** très différentes.

Exemple1: deux applications ré-utilisent un module **date**



Du graphe des appels de fonctions à l'Architecture Logicielle d'un projet

passage à une représentation avec une granularité plus grosse

But: l'Architecture logicielle d'un projet décrit les dépendances entre les blocs le constituant (modules, groupe de modules, bibliothèques)

Structure de graphe orienté *(similaire au graphe des appels de fonctions)*:

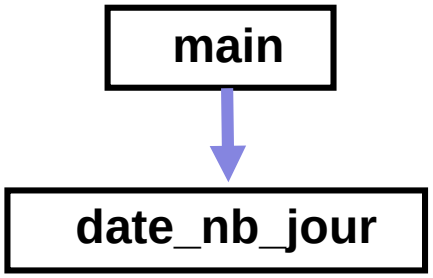
- **un module** **= un nœud**
- **une flèche** **= une dépendance "appelant / appelé" entre 2 noeuds**

Remarque: si 2 modules sont mutuellement dépendants (= possèdent des fonctions qui appellent des fonctions de l'autre module), ils sont regroupés dans un même bloc (notion de *package* en Orienté Objet)

structure de graphe acyclique

Exemple1: application nbjour *avec 2 modules*

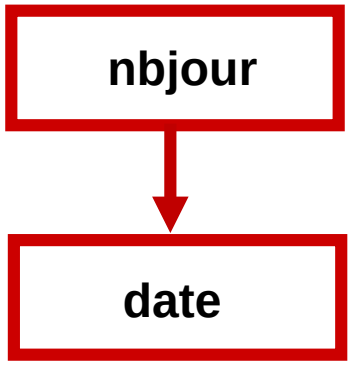
Graphe des appels de fonctions



But du programme :
indiquer le nombre de jours entre deux dates fournies par l'utilisateur

main() est responsable du dialogue utilisateur ; elle est dans le module **nbjour**. Elle appelle la fonction **date_nb_jour()** fournie par le module **date**

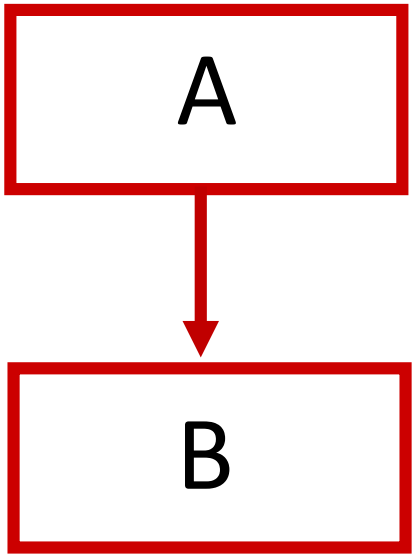
Architecture logicielle



pour pouvoir appeler correctement la fonction **date_nb_jour()**, le module **nbjour** inclut le fichier **date.h** du module **date**

l'**architecture logicielle** montre seulement les liens (dépendances) entre les modules

Nature et conséquences d'une dépendance entre deux modules (1)



Définition: un module **A dépend** d'un module **B** si **A** inclut l'**interface** du module **B** (= **B.h**)

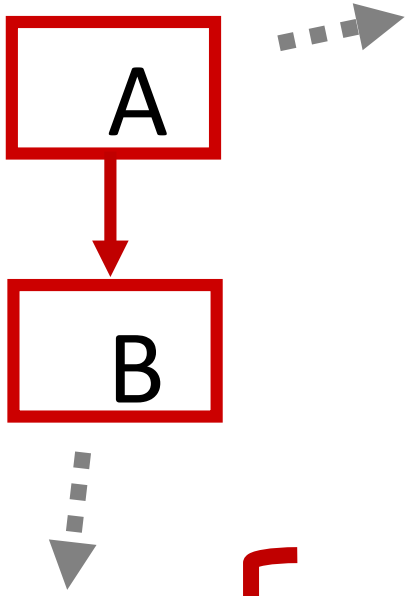
Justification:

le module **A** veut utiliser une ressource du module **B** qui est présente dans **B.h**

Observation: inclure l'**interface** du module **B** réduit la dépendance au minimum car le fichier(**B.h**) est très petit comparé à son **implémentation** (**B.cc**).

On y met seulement ce type d'éléments: ***déclaration*** *de fonctions, définition de type, modèle de structure, symboles...*

Exemple:



module **calcul**

prog.cc

```
#include <iostream>
#include "calcul.h"
using namespace std;
int main(void)
{
  int a(0), b(0);
  cin  >> a >> b ;
  cout << div(a,b) << end;
}
```

calcul.h

```
int div(int num, int denom);
```

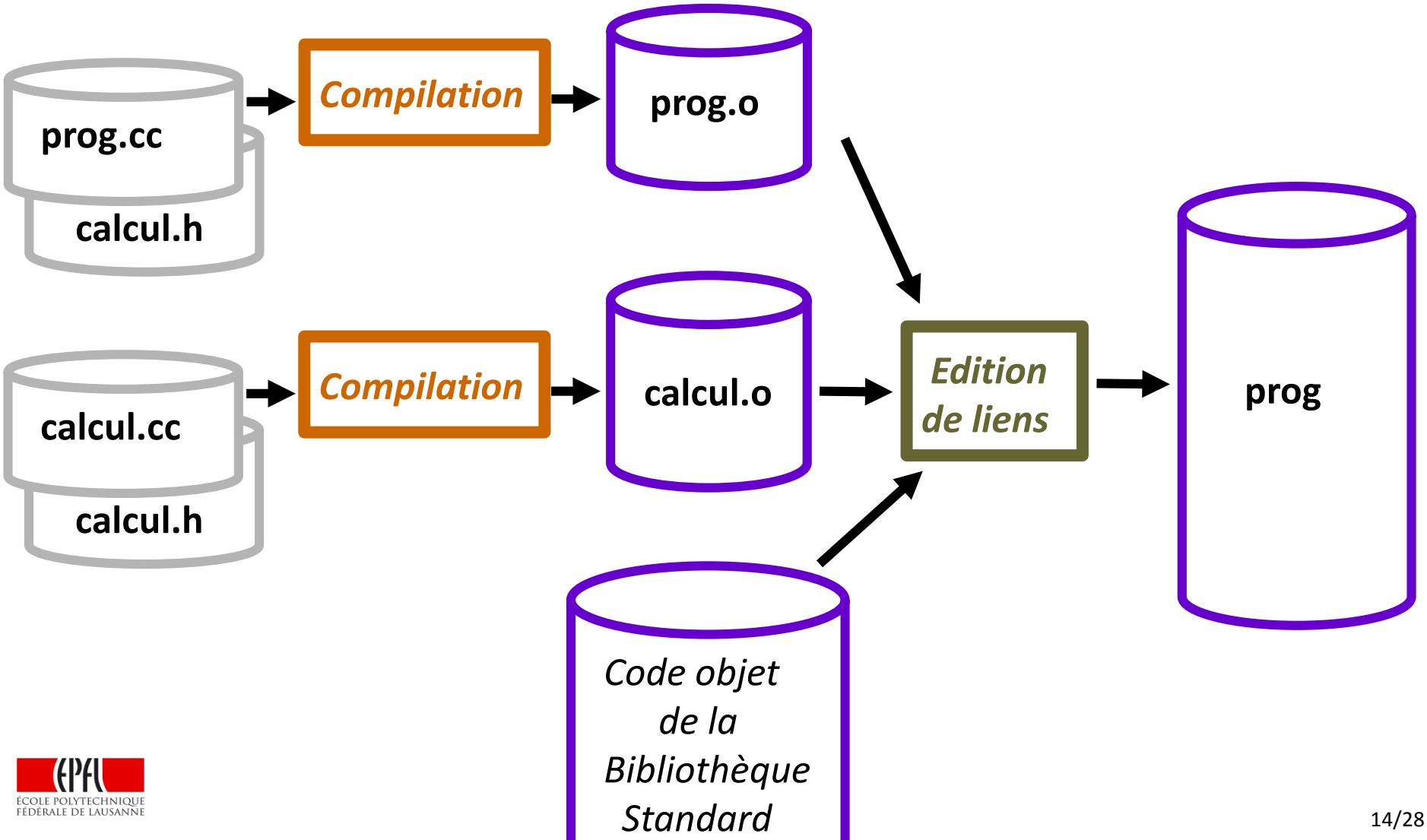
calcul.cc

```
#include "calcul.h"

int div(int num, int denom)
{
  if(denom != 0)
     return num/denom ;
  return 0;
}
```

Exemple (suite):

Comment l'exécutable de ce programme modulaire est-il produit ?



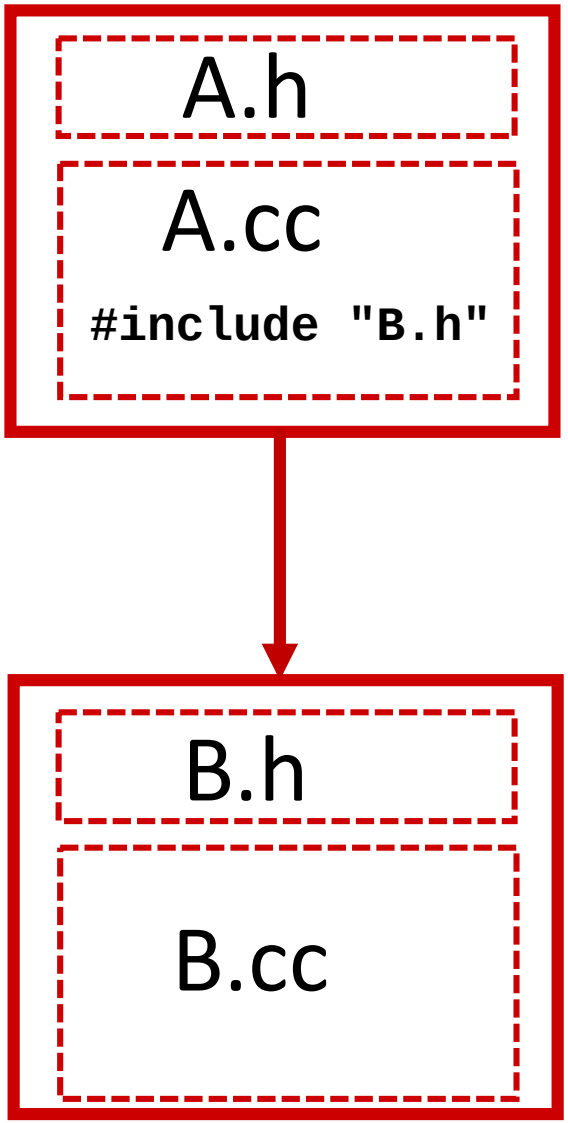
QCM Speakup:

La compilation de **prog.cc** (qui inclut calcul.h) produit le code objet **prog.o** contenant les instructions en langage machine de la fonction **main()** …

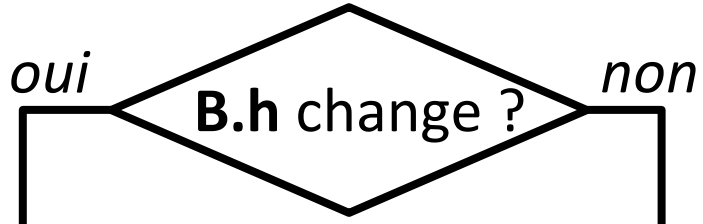
A) … ainsi que l'ensemble des instructions en langage machine de la fonction **div()**

B) … dont une seule instruction en langage machine sert pour traduire l'appel de la fonction **div()**

Nature et conséquences d'une dépendance entre deux modules (2)



Que se passe-t-il si :



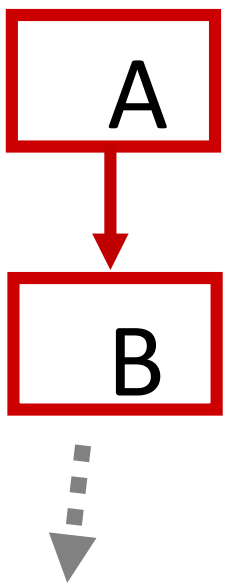
il faut au moins recompiler le module A !

Souvent il faut aussi modifier le module A: peut prendre du temps

Rien à faire au niveau du module A !

Et cela même si l'implémentation **B.cc** a été modifiée: gain de temps

Exemple:



module **calcul**

prog.cc

```
#include <iostream>
#include "calcul.h"
using namespace std;
int main(void)
{
  int a(0), b(0);
  cin  >> a >> b ;
  cout << div(a,b) << end;
}
```

calcul.h

```
int div(int num, int denom);
```

calcul.cc

```
#include "calcul.h"
#define VERYBIG 2147483647
int div(int num, int denom)
{
  if(denom != 0)
     return num/denom ;
  return VERYBIG;
}
```

QCM Speakup:

il faut recompiler **prog.cc** …

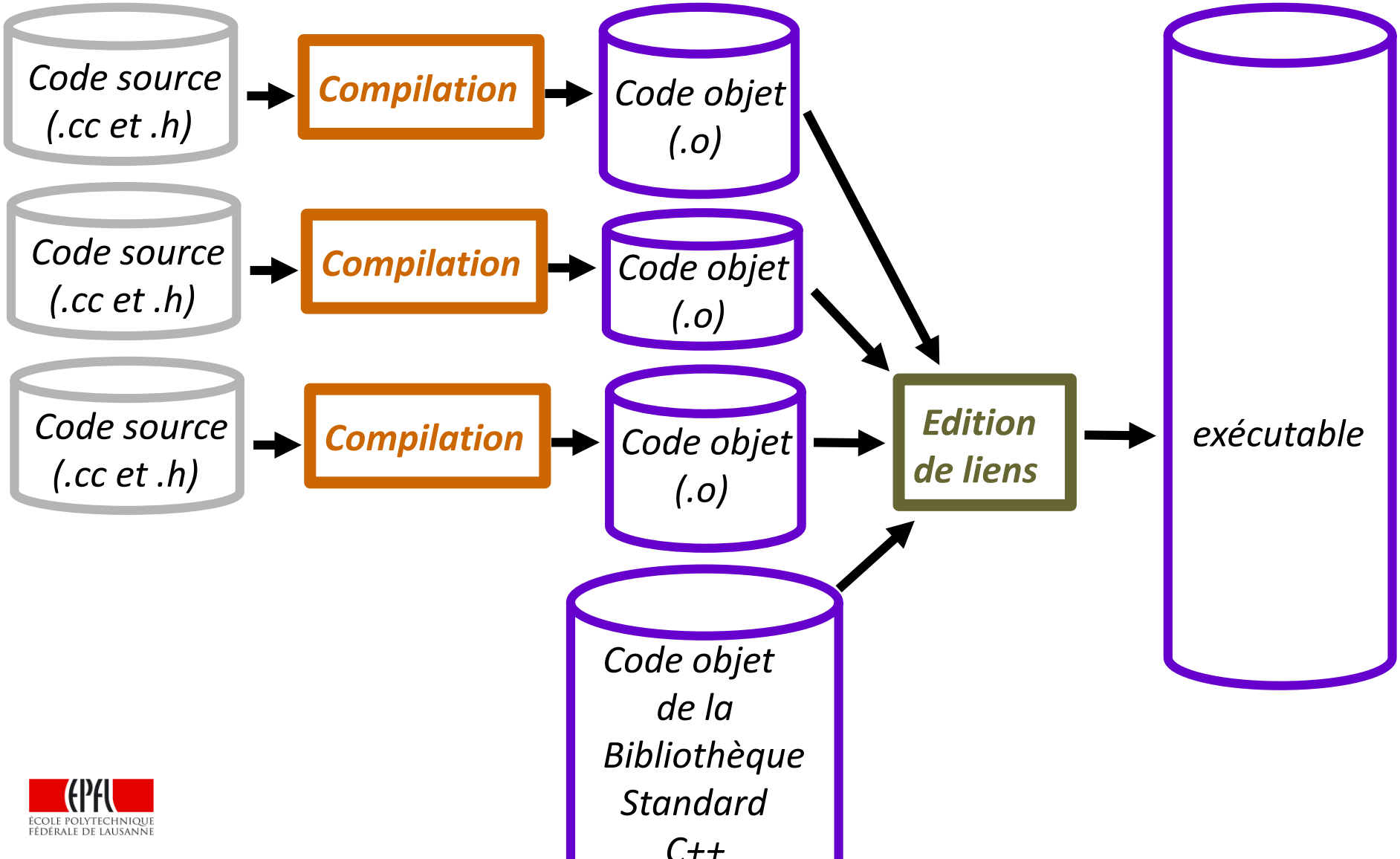
A) … dès que calcul**.h** est modifié

B) … dès que calcul**.cc** est modifié

C) … seulement si calcul**.h** ET calcul**.cc** sont modifiés

Programmation modulaire => Compilation séparée

Cas général



Programmation modulaire => Compilation séparée

Attention: risques importants d'incohérence

Compilation simultanée des fichiers :

- Avantage: garantie de cohérence entre sources, objets, exécutable
- Inconvénient: durée de compilation parfois non négligeable

```
$ g++ prog.cc calcul.cc -o prog
$ ./prog
8 2
4
```

Compilation séparée des fichiers:

- Avantage: tests et mises à jours indépendants
- Inconvénient: risques d'incohérence entre les fichiers objets si le code source est modifié *sans recompiler les fichiers dépendants*

```
$ g++ -c prog.cc
$ geany calcul.cc
$ geany calcul.h
$ g++ -c calcul.cc
$ g++ prog.o calcul.o -o prog
```

De l'Architecture Logicielle au Graphe des dépendances

=> Identifier les dépendances entre fichiers

*Lorsqu'on veut produire un **exécutable*** il faut considérer explicitement tous les fichiers utilisés pour produire **l'exécutable**: ***.h, .cc, .o*** + ***bibliothèques***

Problème: gros risque d'incohérence des versions de tous ces fichiers si on réalise cette gestion « à la main ».

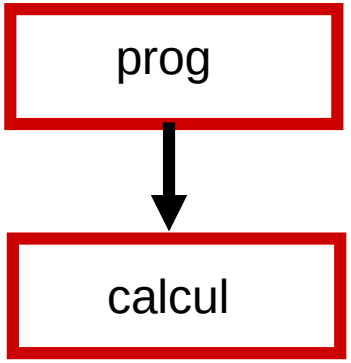
Solution: automatiser les décisions de recompilation avec la commande **make** du système LINUX.

Un Graphe des dépendances de tous les fichiers sources et objets est mémorisé dans un fichier **`Makefile`**. (Série 1)

Différence entre l'architecture logicielle et le graphe des dépendances

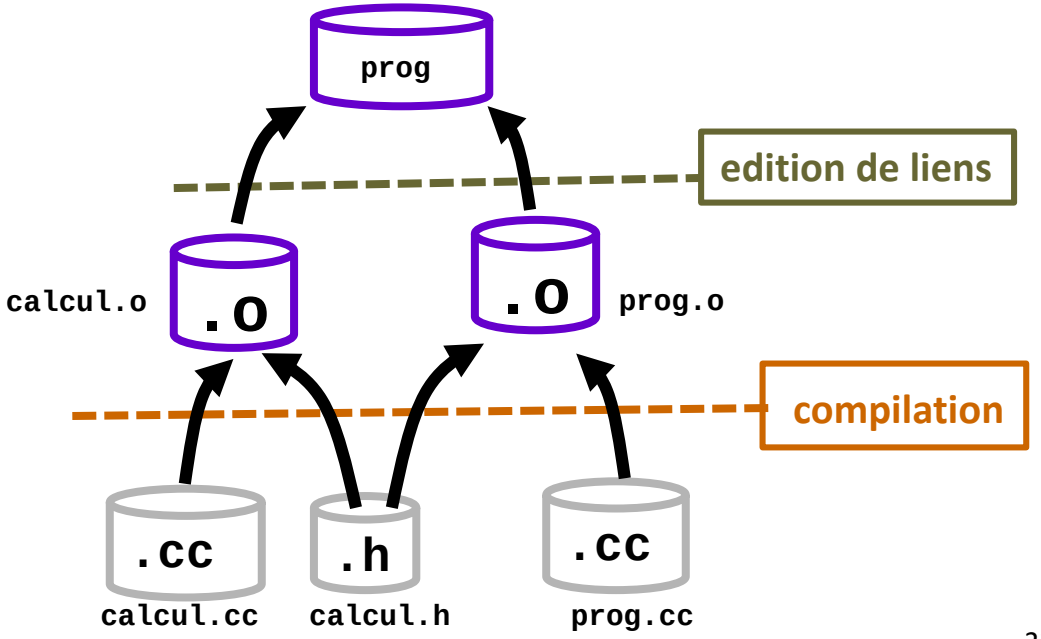
L'architecture logicielle est un outil d'analyse et de conception au moment de la phase *d'analyse* du projet.

L'élément de base est le **module**



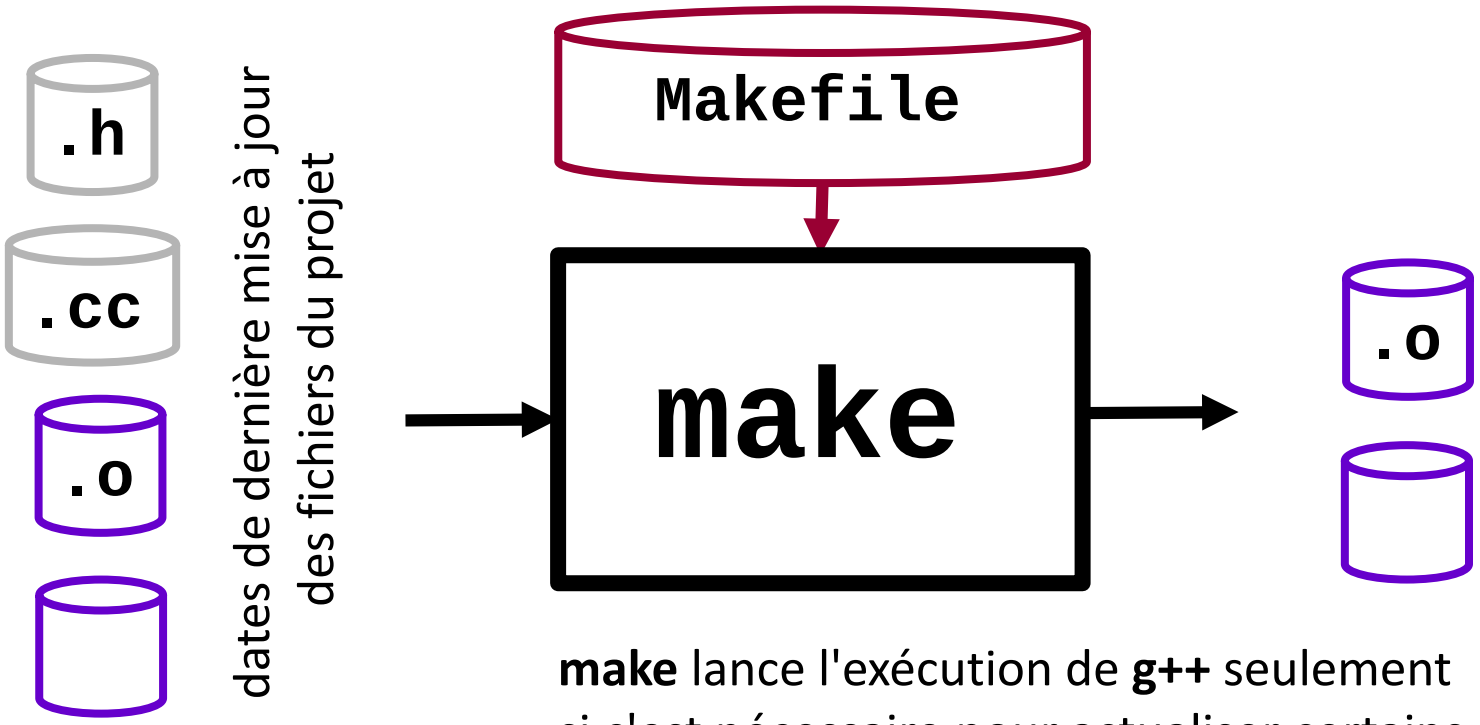
On peut construire le graphe des dépendances de la commande **`make`** à partir de l'architecture logicielle. Ce graphe est un outil de la phase de *codage et de test* du projet.

L'élément de base est le **fichier**



La commande **`make`** *et le fichier* **`Makefile`**

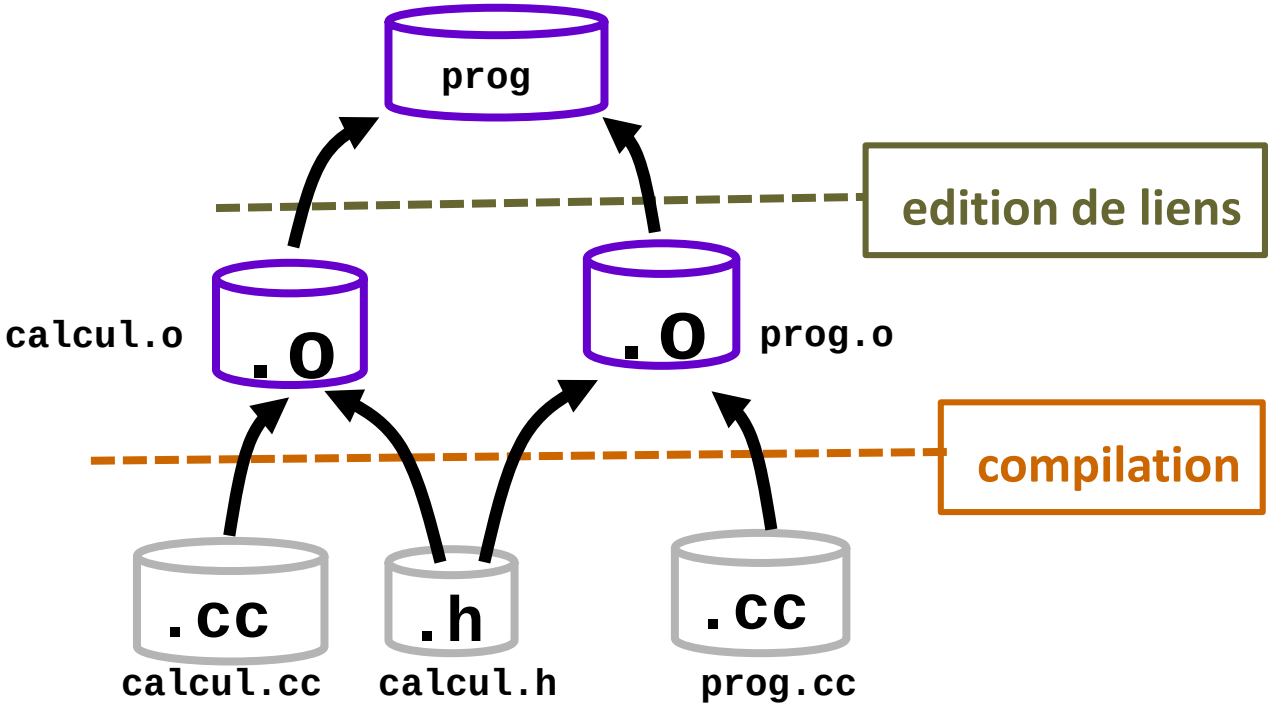
Makefile est un fichier texte contenant des **règles** qui décrivent les dépendances entre fichiers et les commandes à exécuter si **make** détecte une incohérence dans les dates de dernière mise à jour



make lance l'exécution de **g++** seulement si c'est nécessaire pour actualiser certains fichiers objet (.o) et/ou l'exécutable

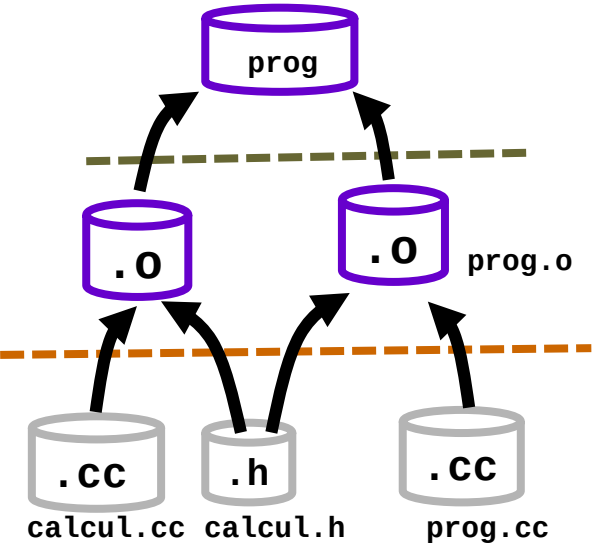
Exemple de Graphe des dépendances d'un projet

Construction: faire apparaitre sur 3 couches les fichiers .c, .h, .o , bibliothêque(s) supplémentaire(s), exécutable. La direction des flèches reflète le flot des input / output des opérations concernées (compilation, édition de liens).



Remarque: l'orientation de ce graphe n'est pas standardisée (cf Série 1)

Que contient le fichier **`Makefile`** *?*



Graphe des dépendances

Essentiellement des **variables** et des **règles**.

Une dépendance et sa commande associée sont décrites avec une **règle** :

```
cible: dépendance(s)
    commande(s)
```

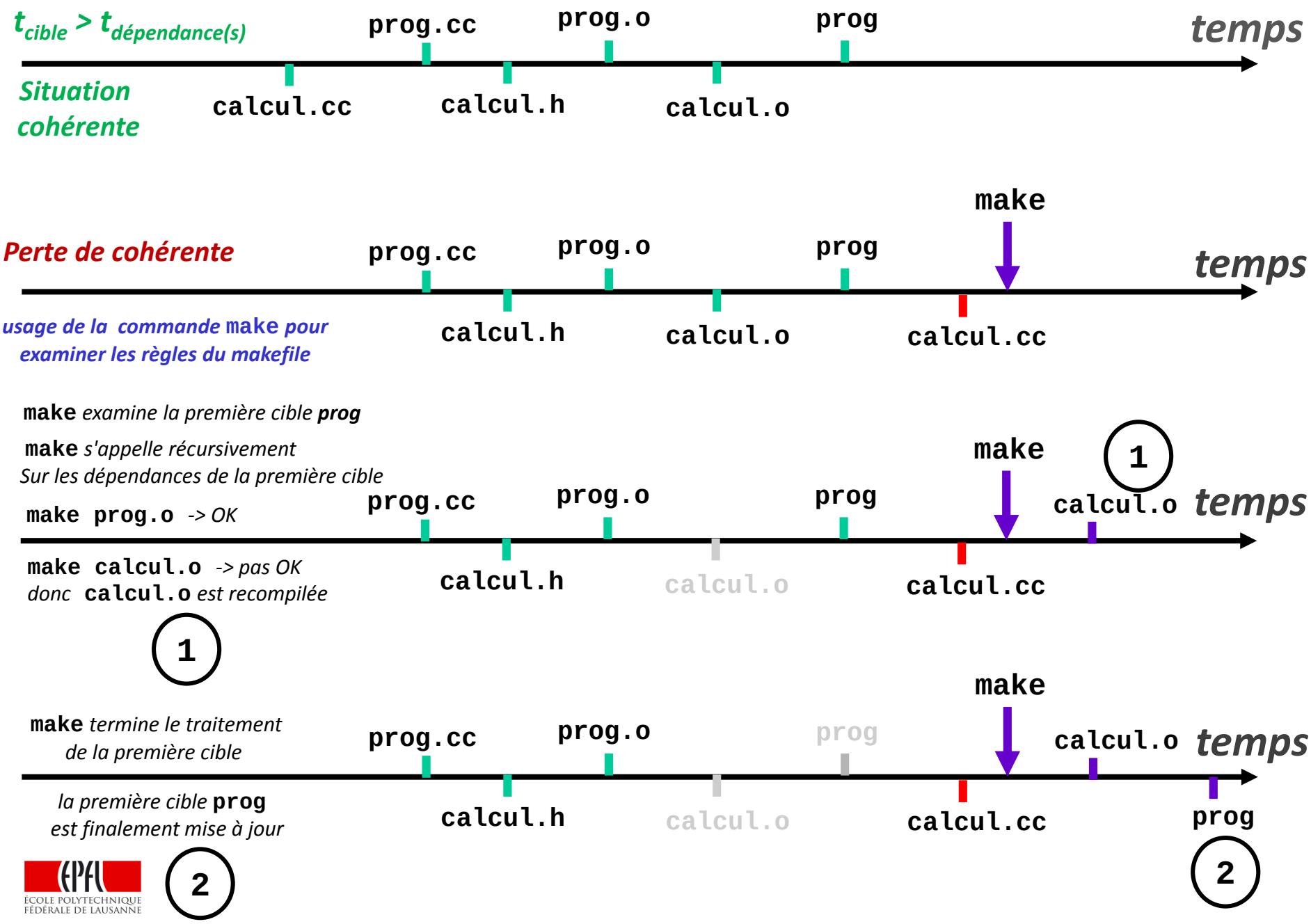
```
prog: prog.o calcul.o
    gcc prog.o calcul.o  -o prog

calcul.o: calcul.cc calcul.h
    gcc -c calcul.cc

prog.o: prog.cc calcul.h
    gcc -c prog.cc
```

la commande **make**
examine la **1ière règle**
ou celle qui est indiquée sur la ligne de commande (ex: **make calcul.o**) .
si une **dépendance** est plus récente que la **cible**
alors la **commande** est exécutée

Illustration d'un exemple sur l'axe temporel montrant la dernière mise à jour de tous les fichiers



Exemple d'exécution de **`make`** *dans le cas du slide précédent*

```
prog: prog.o calcul.o
      gcc prog.o calcul.o  -o prog

prog.o: prog.cc calcul.h
      gcc -c prog.cc

calcul.o: calcul.cc calcul.h
      gcc -c calcul.cc
```

L'ordre des **règles** n'est pas important car **make** examine récursivement les cibles impliquées dans une règle.

Exemple du slide précédent: on modifie seulement **calcul.cc** et on lance **make** :

- ***la première*** cible **`prog`** possède 2 dépendances: **`prog.o calcul.o`**

  - la première dépendance est **`prog.o`**; comme c'est aussi la cible **`prog.o`**, **make** examine d'abord sa règle: elle possède 2 dépendances: **`prog.c`** et **`calcul.h`** qui ne sont pas des cibles (pas de règle). De plus la cible **`prog.o`** est plus récente que ses dépendances donc rien n'est fait.

  - la 2ième dépendance est **`calcul.o`** ; elle est aussi une cible **`calcul.o`**, donc **make** examine ensuite sa règle: elle possède 2 dépendances **`calcul.c`** et **`calcul.h`** qui ne sont pas des cibles (pas de règle). MAIS cette fois la cible **`calcul.o`** est plus ancienne qu'une de ses dépendances,

    DONC **`gcc -c calcul.cc`** permet de mettre à jour **`calcul.o`**

- maintenant **make** constate que **`calcul.o`** est plus récente que **`prog`**

  DONC **`gcc prog.o calcul.o -o prog`** est exécutée

Résumé

- **Principes justifiant un module: séparation des tâches, abstraction, ré-utilisation, et rassembler des dépendances.**
- **Un module est constitué d'une interface (.h) et d'une implémentation (.cc).**
- **L'interface (.h) documente les prototypes des fonctions pouvant être appelées dans d'autres modules. Elle décrit seulement le but de ces fonctions (*what they do*) mais pas le comment (*how they do it*) car c'est la responsabilité de l'implémentation (.cc)**
- **Un module doit inclure l'interface d'un autre module s'il veut appeler des fonctions de cet autre module.**
- **l'architecture logicielle résume les dépendances**
- **la commande make permet de maîtriser les dépendances**