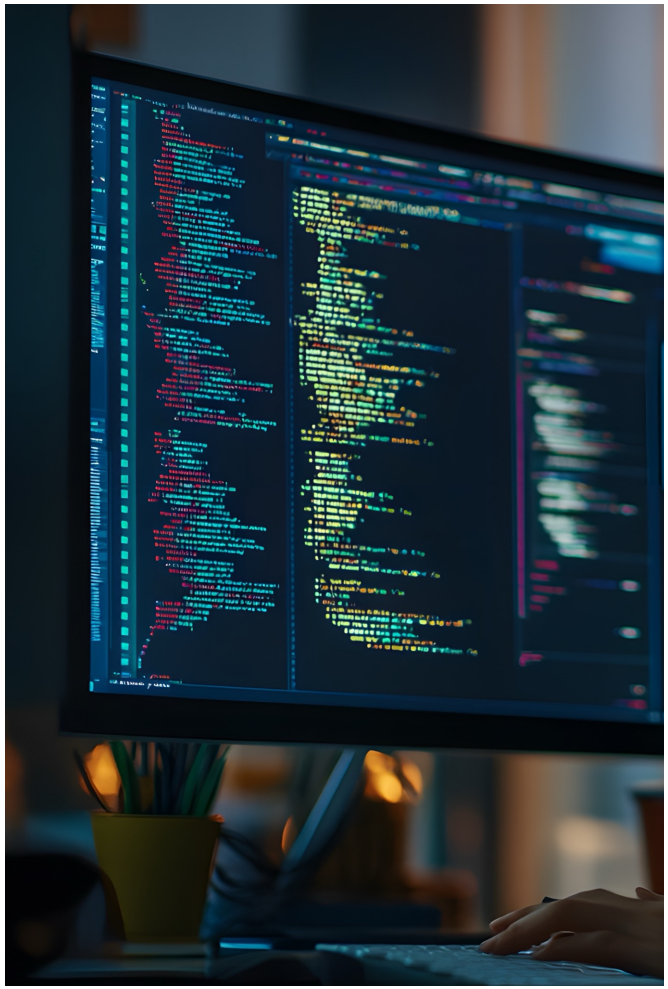
Programmation Orientée Projet
COM-112(a)

Semaine 1

Rafael Pires
rafael.pires@epfl.ch

Lausanne, 20.02.2026

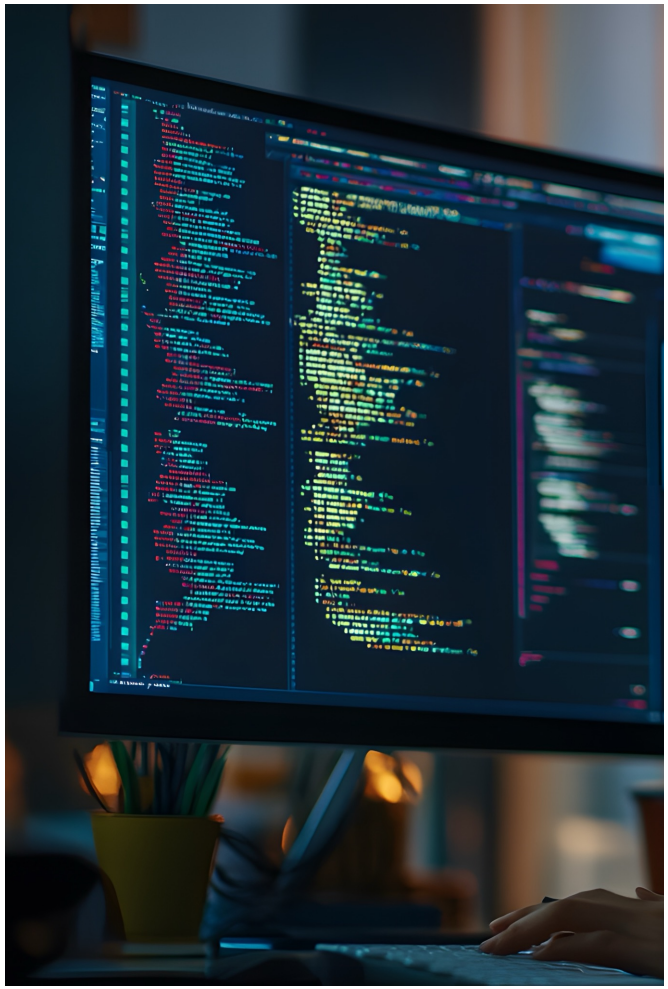
Présentation



- **Rafael Pires**
- Licence et Master en Informatique, Master en Mécatronique (Brésil)
- Doctorat en Informatique à l'Université de Neuchâtel, 2020
- EPFL :
  - Postdoc au laboratoire

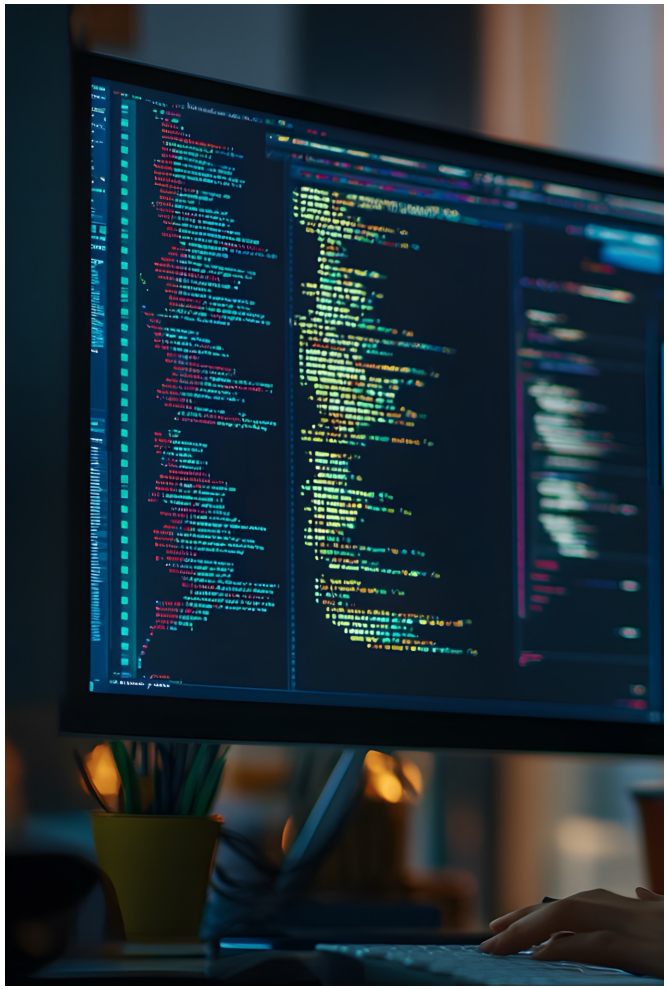
    Scalable Computing Systems (SaCS)
  - Enseignant depuis l'Automne 2024

Objectifs du Cours C++ POP



- Maîtrises des bases du C++11 et programmation orientée objet
- Introduction au développement de projets logiciels
  - *Architecture modulaire*
- Mettre en œuvre une interface graphique (GUI)
  - GTKmm 4

Organisation



- **2h de cours sur 7 semaines**
  - 1h MOOC (massive open online course) obligatoire
  - 1h ex-cathedra (classe inversée + complément pour le projet)
- **2h TP/projet sur 11 semaines**

- **Postes de travail virtuels** :
  - Directement en salles **CO020**, **CO021**, **CO023**, **CO4** et **CO5**
  - Sur votre machine via **https://vdi.epfl.ch**
  - Machine virtuelle : **IC-CO-IN-SC-INJ-2026-Spring** (Linux)

- **Autre possibilité** : **installation personnelle** sur votre propre machine. IDEs : Geany, VS Code, Zed etc.

MOOC
Introduction à la programmation orientée objet en C++

- S'inscrire **uniquement** via le **lien fourni**

  (afin d'éviter les frais Coursera)
- Format de 7 semaines :

  usage partiel sur 6 semaines
- Quizz et problèmes avec autograder
- Exercices avec leur corrigé
- Gratuit, ainsi que la transcription écrite :

  BOOC

Remerciements :

Jamila Sam et Jean-Cédric Chappelier

Documents de cours

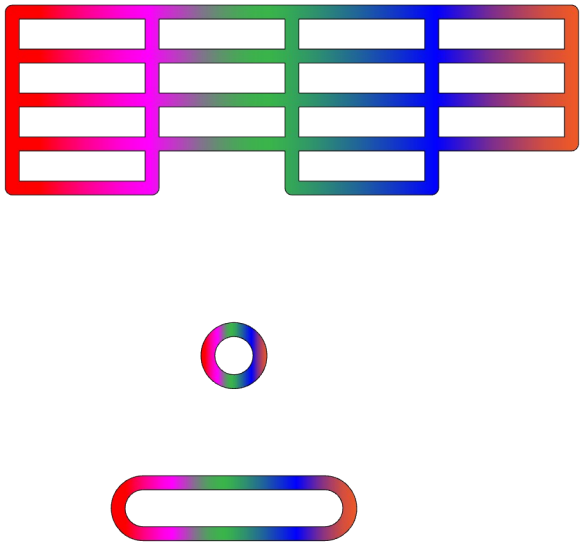
- Toutes les informations et liens vers les documents de cours sont sur **Moodle** cours COM-112(a)

  https://moodle.epfl.ch/course/view.php?id=15698

- Tous les cours sont **enregistrés**
  - **Enregistrement** à retrouver sur Moodle après
  - Mais… venez au cours autant que possible !
- Pour vos questions : **Ed Discussion**
  - Posez vos **questions** sur le cours, les exercices, le miniprojet
  - **Catégorisez** votre question correctement

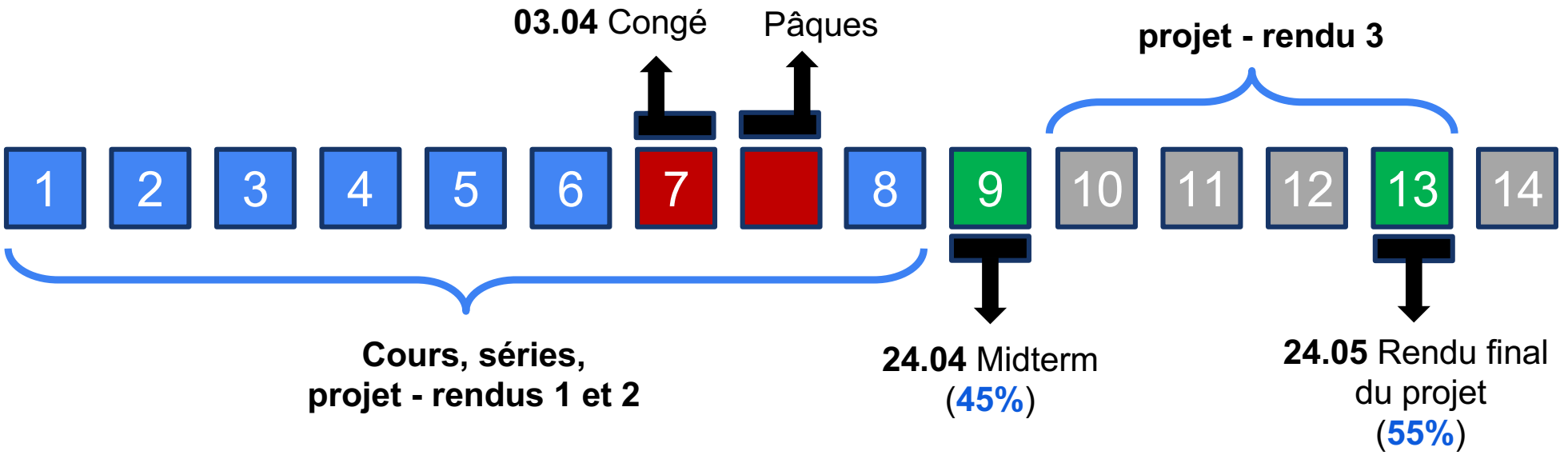
Remerciements : Ronan Boulic

Projet – Brick Breaker



- Constituer un groupe sur Moodle
  - 2 personnes (ni une, ni trois)
- Chaque groupe sera automatiquement affecté à un·e assistant·e / coach
  - Des changements sont possibles avec l'accord des assistant·e·s
  - Un·e assistant·e ne peut coacher qu'au maximum 11 groupes
  - À partir de la semaine 2, un groupe se rend toujours au même endroit lors des séances de TP, où se trouvera son coach
  - Les coachs notent les rendus, mais pas ceux de leurs supervisé·e·s
- Pour toute question : **EdStem**

Programme du cours



Oral aléatoire sur le code du projet (sem 10-14) : convocation par email une semaine à l'avance.
Il s'agira d'expliquer comment fonctionne votre code pour réaliser une partie du projet (sur votre laptop)

Planning

Sem.	Date	MOOC : Cours h1	11-12h : Cours h2	12-13h : TP h1	13-14h : TP h2	Projet
1	20 fév	Inscription MOOC	Prog. Modulaire	PoP_s0: make et makefile	Développement projet	Intro méthodes
2	27 fév	Sem1: Intro POO	Vue générale projet, architecture	PoP_s1: Donnée projet	Exercices MOOC sem 1	Donnée disponible
3	6 mars	Sem2: Constr./Destr.	Lecture fichier, préprocesseur	PoP_s2: Préprocesseur et stubs	Exercices MOOC sem 2	
4	13 mars	Sem3: Surcharge	Vector, type paramétré	PoP_s3: Static, type paramétré	Exercices MOOC sem 3	
5	20 mars	Sem4: Héritage	MVC / GTKmm / dessin	PoP_s4: MVC et dessin	Exercices MOOC sem 4	
6	27 mars	Sem5: Polymorphisme	Modèle indépendant GTKmm	PoP_s5: GTKmm / GUI	Exercices MOOC sem 5	**RENDU 1 (29 mars)**
7	3 avril	**Good Friday - Férié**				
	10 avril	**VACANCES DE PÂQUES**				
8	17 avril	Sem6: Héritage mult.	GTK idle vs timer / Révisions	PoP_s6: GTKmm / événements	Exercices MOOC sem 6	
9	24 avril	Pas de cours	Pas de cours	**TEST ÉCRIT (45%) dans les salles CE 11 et CO 1**		**RENDU 2 (26 avril)**
10	1 mai	Pas de cours	Oral individuel possible	Temps projet	Temps projet	
11	8 mai	Pas de cours	Oral individuel possible	Temps projet	Temps projet	
12	15 mai	Pas de cours	Oral individuel possible	Temps projet	Temps projet	
13	22 mai	Pas de cours	Oral individuel possible	Temps projet	Temps projet	**RENDU FINAL (24 mai)**
14	29 mai	Pas de cours	Oral individuel possible	Fin du semestre		

Equipe

Assistant.e.s doctorant.e.s :

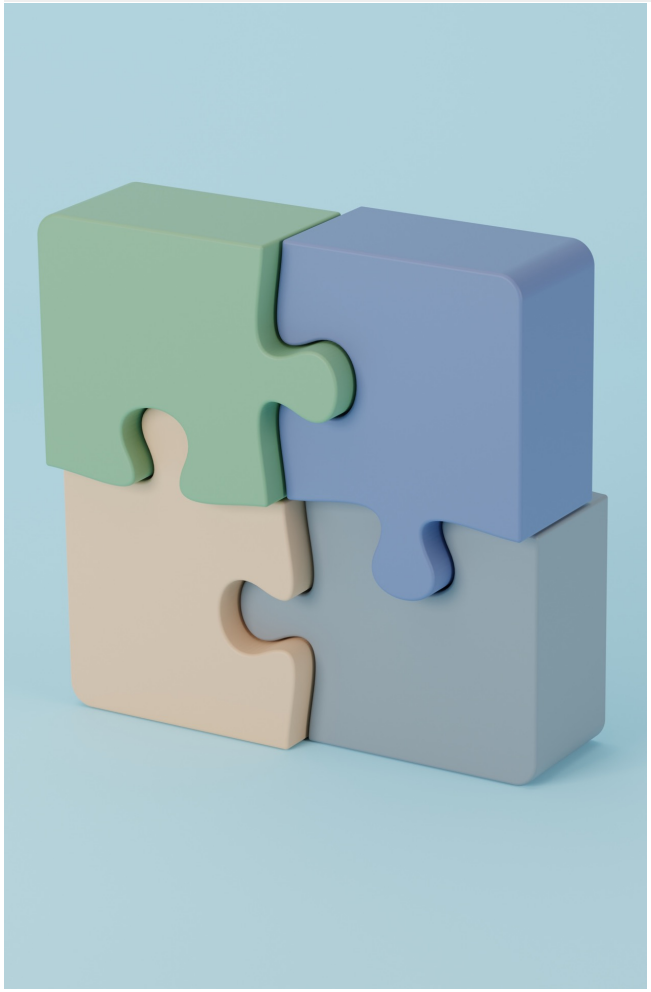
Ramya
Prabhu

Maxime
Jacovella

Assistant.e.s étudiant.e.s :

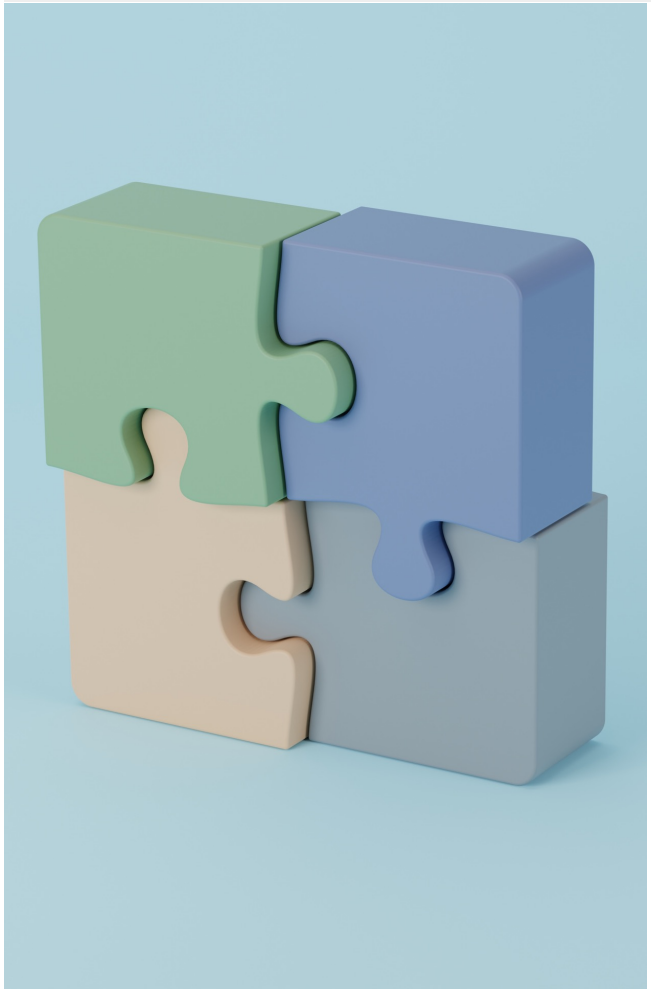
Adam Ben Hamdene
Idriss Benjelloun
Maximo Castellari
Alessandro De Zen
Sébastien Dévaud
Blanche Elisa Marie Doussaud
Flavien Jaquerod
Stanislas Johann Krainik-Saul
Jan Amitai Livny
Jonas Baptiste David Verbois
Océane Voland

Décomposition modulaire

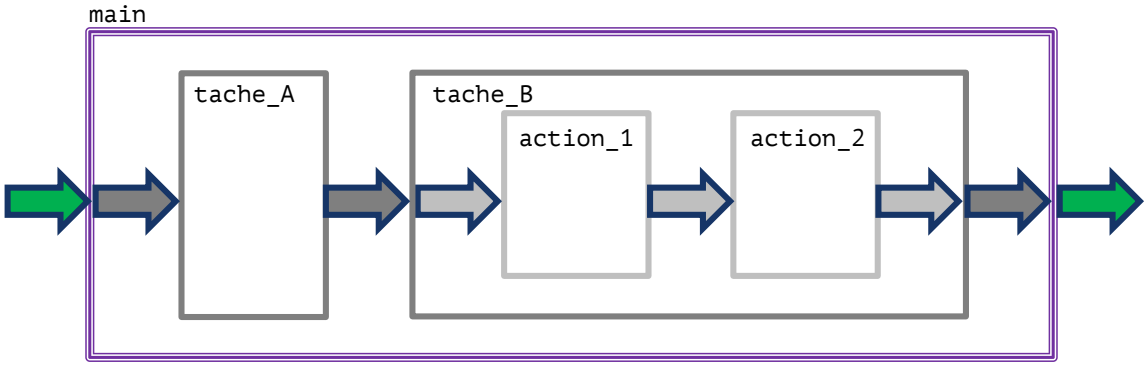


- Objectifs : maîtriser un projet
  - Définir la notion d'architecture logicielle
  - Identifier et minimiser les dépendances entre modules
- Plan :
  - Un module : 2 fichiers = interface + implémentation
  - Graphe des appels
  - Graphe des dépendances
  - La commande `make` et le fichier `Makefile`

Décomposition modulaire : principe d'abstraction



- Présenter l'idée générale de la solution sans se perdre dans les détails

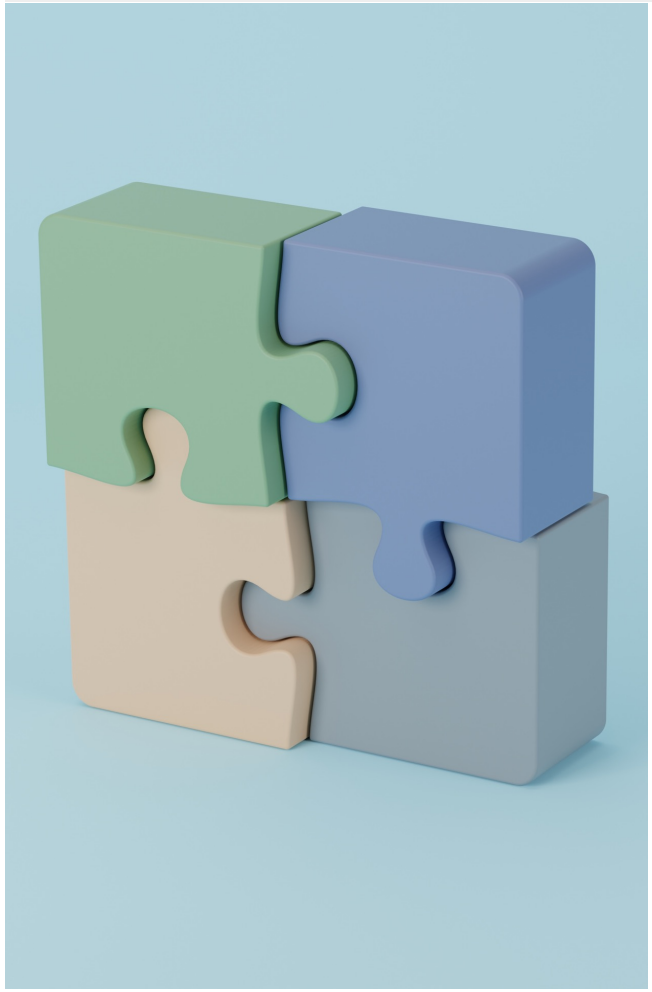



```
main()
{
  tache_A();
  tache_B();
}
```

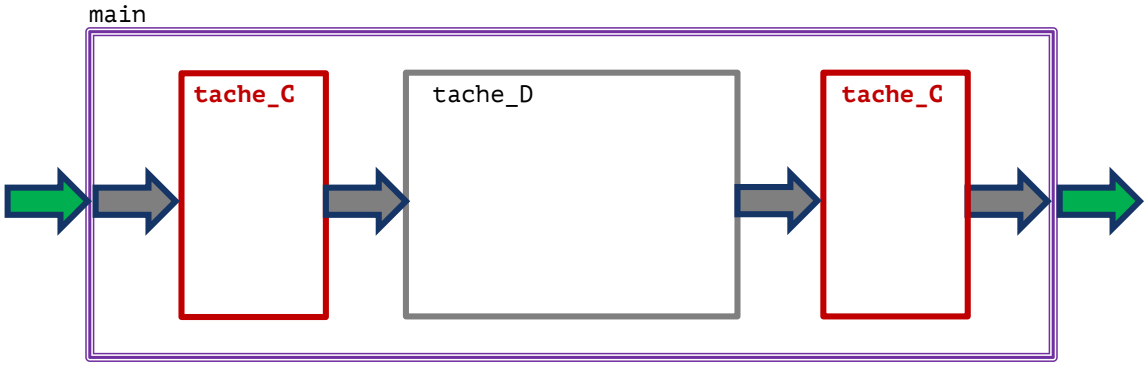
```
tache_A()
{
  …
}
```

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

Décomposition modulaire : principe de ré-utilisation

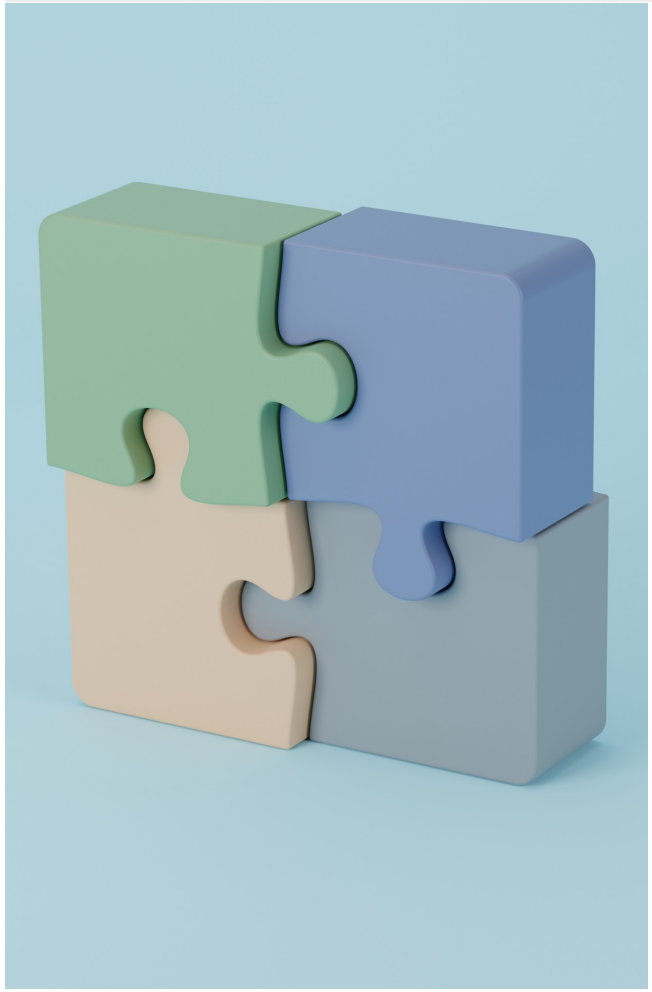


- Réduire l'effort de mise au point et la taille du code en ré-utilisant du code



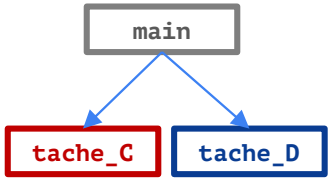

```
main()
{
  tache_C();
  tache_D();
  tache_C();
}
```

Le graphe des appels de fonctions

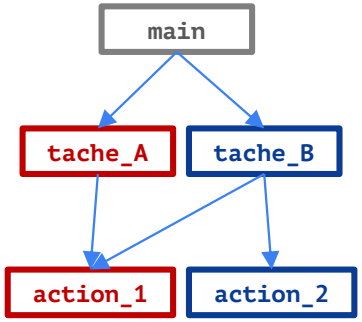


- Offre une vue synthétique des dépendances entre fonctions:

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



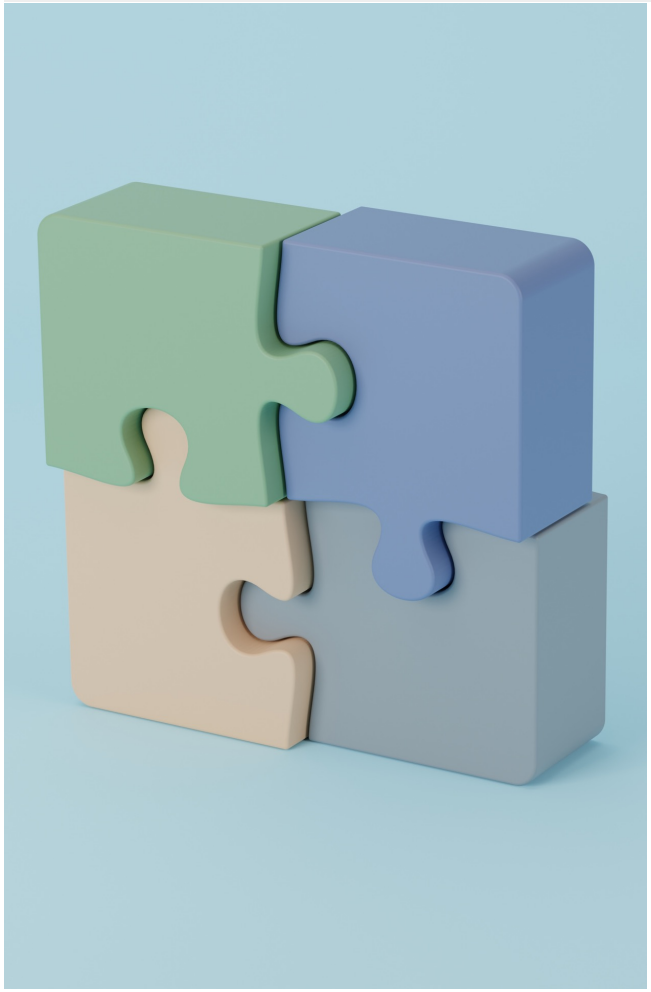

```
main()
{
  tache_A();
  tache_B();
}
```




```
tache_A()
{
  action_1();
}
```

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

Décomposition modulaire



- Abstraction
  - Vue générale claire, déléguer les sous-problèmes
- Ré-utilisation
  - Fonctions utilitaires (ex. math)
- Séparation des fonctionalités
  - Unités logicielles cohérentes (module ou groupe de modules)
- Encapsulation de type « Boîte Noire »
  - Minimiser les dépendances entre modules
- Concentration des dépendances
  - Vis-à-vis de bibliotèques externes

C’est quoi un module ?

*un **module** = une **interface** + une **implémentation***

- Interface
  - Décrit le but ; Contient les **prototypes** des fonctions exportées
  - Fichier en-tête (**.h**)
  - Pour pouvoir appeler ces fonctions dans d’autres modules, il faut et il suffit d’une directive **include** pour inclure cette interface
- Implémentation
  - Définit comment les fonctions sont mises en œuvre (**.cc**)
  - Une même interface (**.h**) peut avoir des implémentations (**.cc**) très différentes

C’est quoi un module ?

*un **module** = une **interface** + une **implémentation***

- Interface
- Implémentation

module calcul

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;
}
```

L’architecture logicielle

- Décrire les dépendances entre les blocs qui constituent le projet (modules, bibliothèques)

module nbjour

date.cc

```
#include “date.h”
int main()
{
  x = date_nb_jour(…);
  // …
}
```

module date

date.h

```
int date_nb_jour(…);
```

date.cc

```
#include “date.h”
int date_nb_jour(…)
{
  // …
}
```

L’architecture logicielle

module nbjour

prog.cc

```
#include “date.h”
int main()
{
  x = date_nb_jour(…);
  // …
}
```

module date

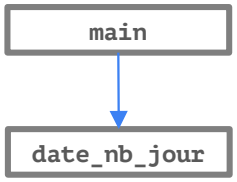
date.h

```
int date_nb_jour(…);
```

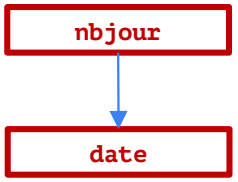
date.cc

```
#include “date.h”
int date_nb_jour(…)
{
  // …
}
```

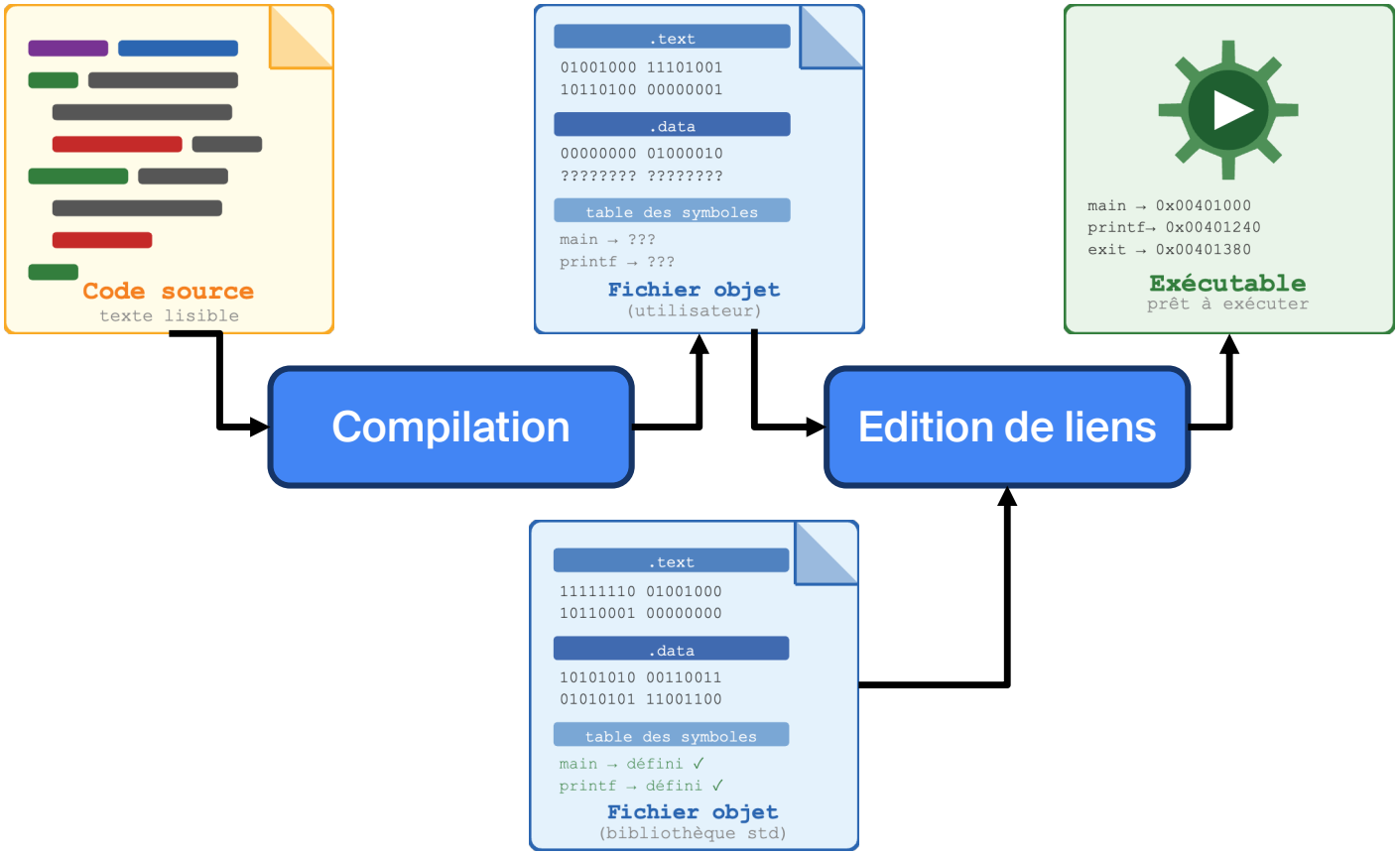
Graphe des appels de fonctions



Architecture logicielle



Rappel : Production d'un exécutable



Programmation modulaire → Compilation séparée

- Compilation simultanée
  - Avantage : garantie de cohérence
  - Inconvénient : durée de compilation

```
$ g++ prog.cc calcul.cc -o prog
```

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

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

Risque d'incohérence lié à la compilation séparée

main.cc

```
#include "add.h"
int main()
{
  return addition(2, 3);
}
```

add.h

```
int addition(int a, int b);
```

add.cc

```
#include "add.h"
int addition(int a, int b)
{
  return a + b
}
```

main.o

add.o

Edition de liens

add.h

```
int addition(long a, long b);
```

add.cc

```
#include "add.h"
int addition(long a, long b)
{
  return a + b
}
```

add.o

Compilation séparée

- Lorsqu'on veut produire un fichier exécutable, il faut considérer explicitement tous les fichiers utilisés pour produire cet exécutable : `.o` + bibliotèques + `.h`, `.cc`
- 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 de dépendances** de tous les fichiers sources et objets est mémorisé dans un fichier `Makefile` (Série 0)

Rappel : L'architecture logicielle

module nbjour

prog.cc

```
#include "date.h"
int main()
{
  x = date_nb_jour(…);
  // …
}
```

module date

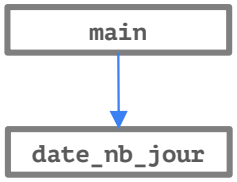
date.h

```
int date_nb_jour(…);
```

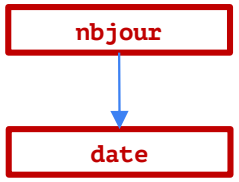
date.cc

```
#include "date.h"
int date_nb_jour(…)
{
  // …
}
```

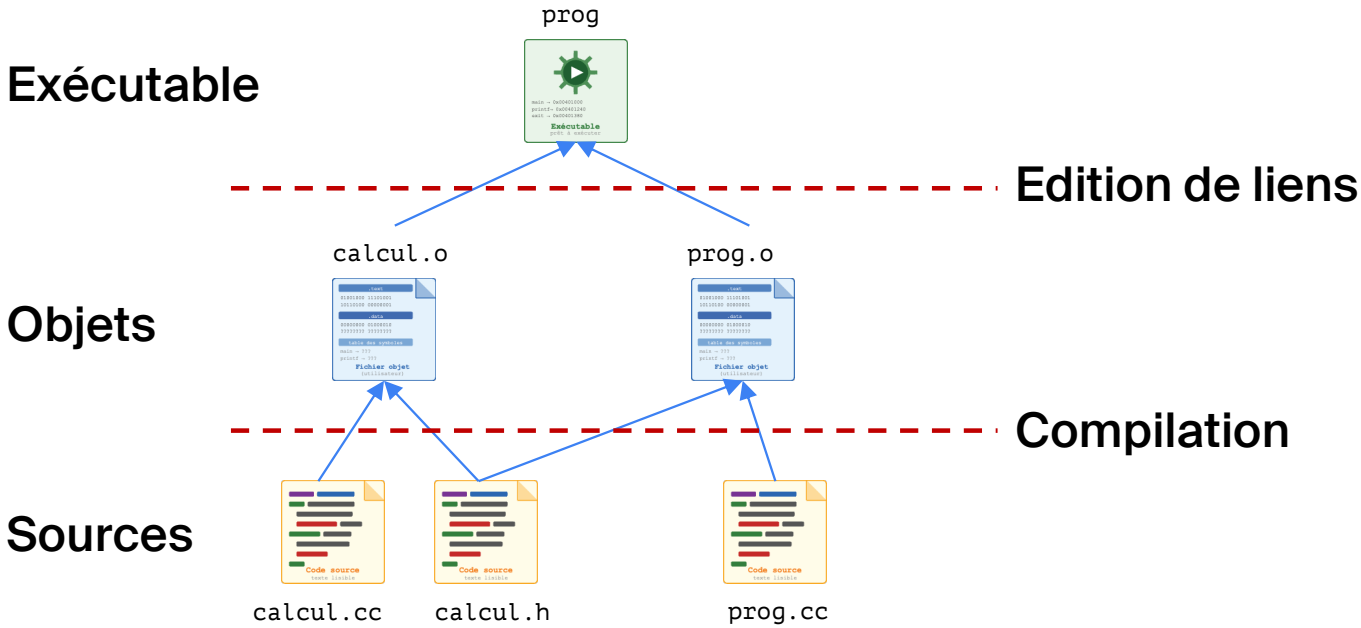
Graphe des appels de fonctions



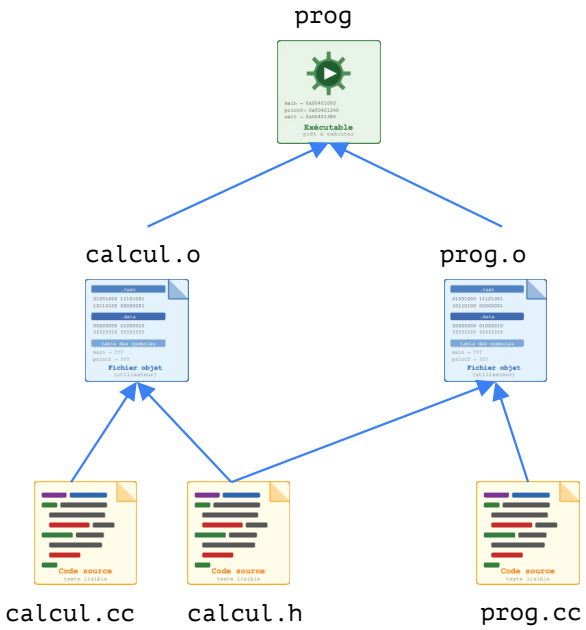
Architecture logicielle



Graphe des dépendances d'un projet



Graphe des dépendances → Makefile




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

Makefile

```
prog: prog.o calcul.o
    g++ prog.o calcul.o -o prog

calcul.o: calcul.cc calcul.h
    g++ -c calul.cc

prog.o: prog.cc calcul.h
    g++ -c prog.cc
```

- La commande **make** examine la 1ère règle
  - Si une **dépendance** est plus récente que la **cible**
  - alors la **commande** est exécutée

Résumé Cours 1

- 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**) document les **prototypes** des fonctions pouvant être appelées dans d'autres modules. Elle décrit seulement le but de ces fonctions mais **pas le comment** car c'est la responsabilité de l'implémentation (**.cc**)
- L'**architecture logicielle** résume les dépendances entre modules
- La commande **make** permet de maîtriser les dépendances entre fichiers

rafael.pires@epfl.ch

Merci