Systèmes d’exploitation

Séance 1

Ce que vous devez savoir pour organiser et compiler vos programmes IGI-3004

Dans un projet en Java

- Le code contient un fichier .java par classe
- Le code est ainsi organisé en entité logique

→ Facilite l'édition du code & augmente sa ré-utilisabilité

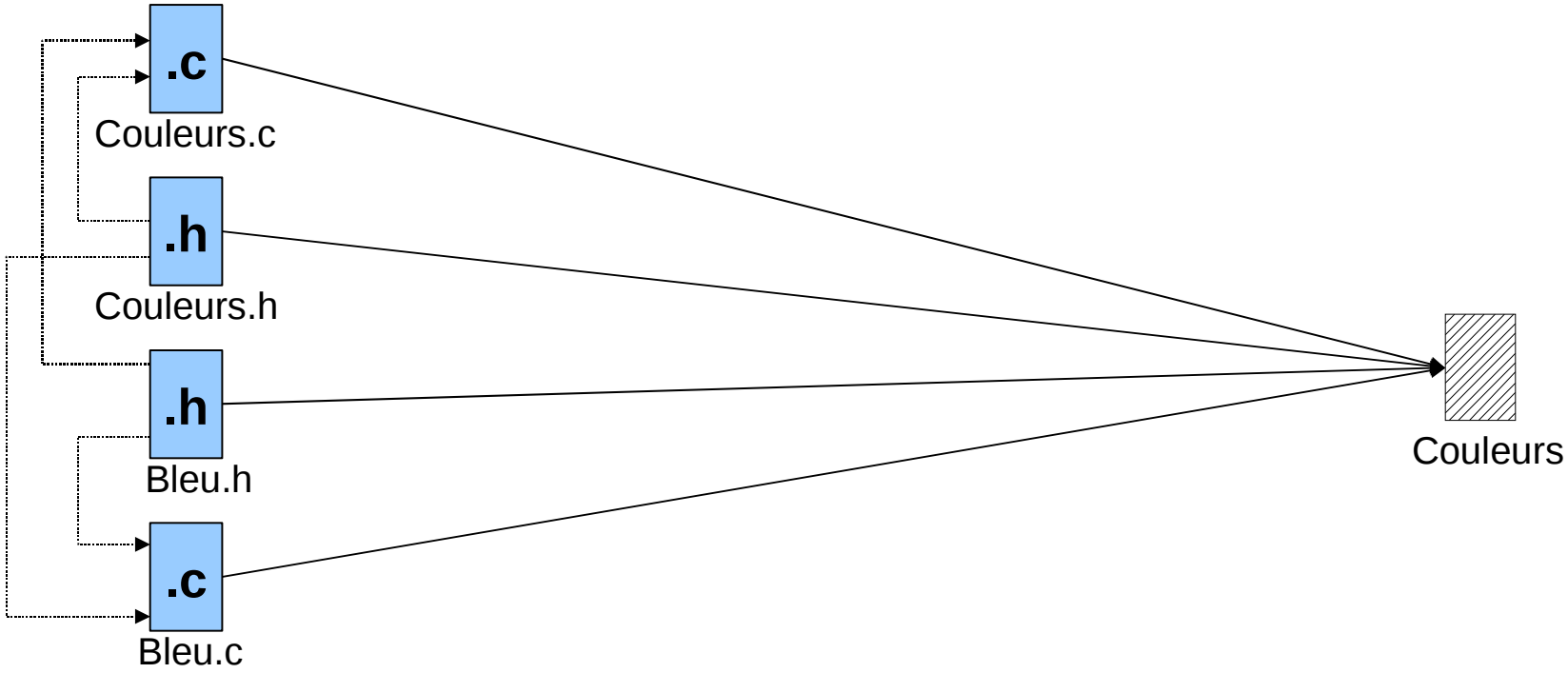
- De plus, si une classe est modifiée, seules cette classe et celles qui en dépendent sont recompilées
- On travaille en général sur une petite portion de code, le reste étant inchangé

→ Gagne du temps à la compilation

Et en C ?

Comment obtenir les mêmes avantages qu'en java ?

Exemple 1 : projet Couleurs avec plusieurs fichiers/bibliothèques



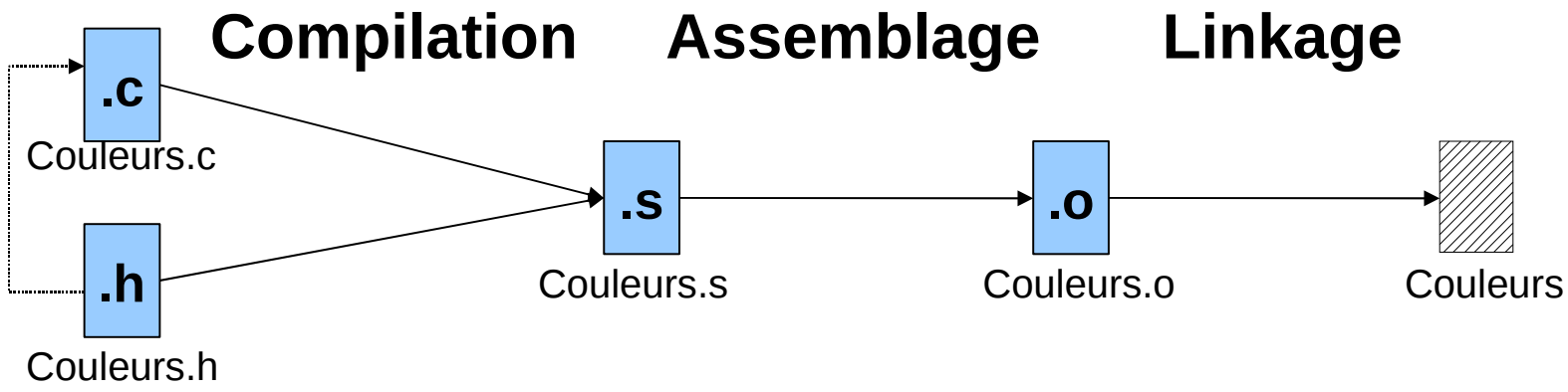
- **Compilation : `gcc Couleurs.c Bleu.c -o Couleur`**
- **Problème : Si l'on ne modifie que Bleu.c, on doit quand même recompiler tous les fichiers !**

Et en C ?

Comment obtenir les mêmes avantages qu'en java ?

- **Compilation séparée et utilisation de 'make' permettent de :**
  - **Gérer de grands programmes**
  - **En gardant traces des changements**
  - **Compiler uniquement les parties qui ont été changées**
  - **Compilation automatique**

Exemple 2 : compilation séparée



1. **Compilation :** C vers assembleur ; produit des fichiers .s

```
gcc -S Couleurs.c
```

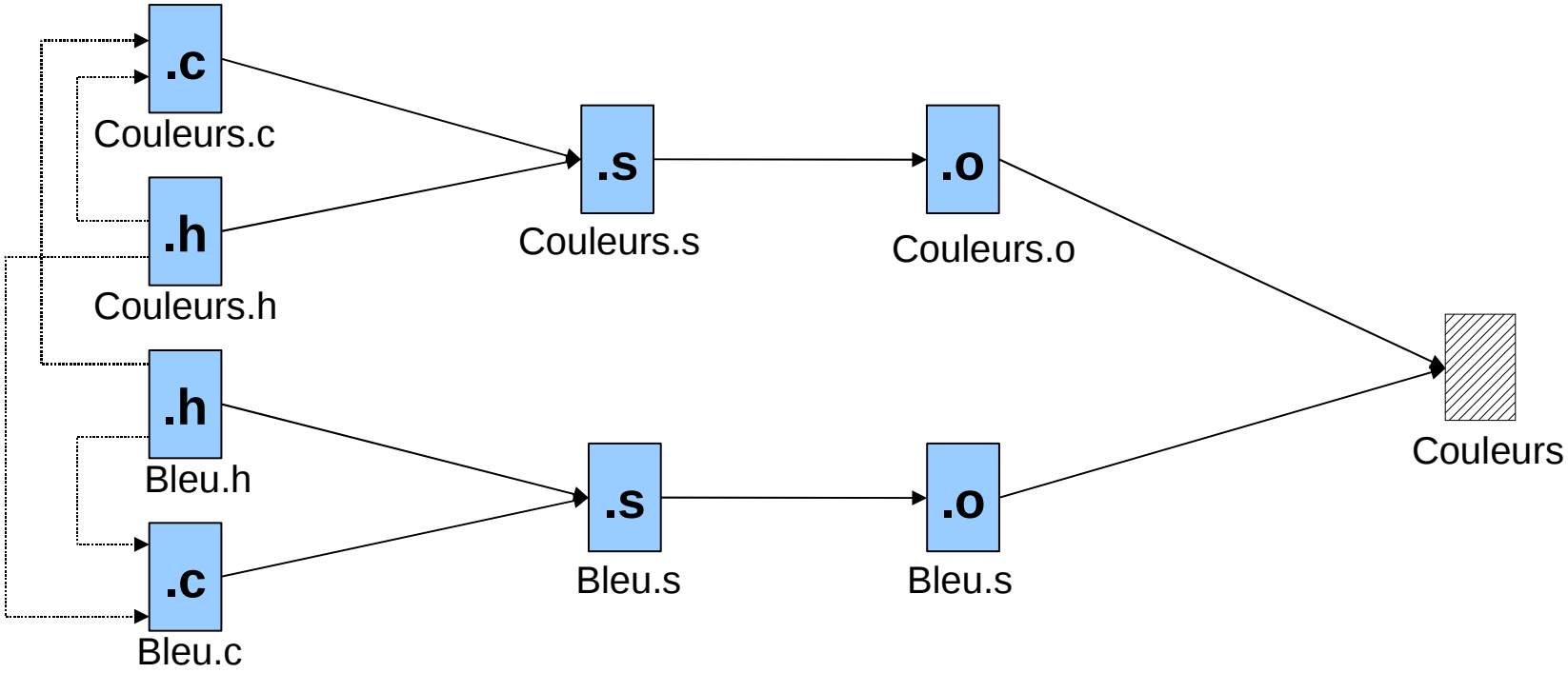
2. **Assemblage :** assembleur vers code objet ; produit des fichiers .o

```
gcc -c Couleurs.s
```

3. **Linkage :** lie le code objet avec certaines librairies qui contiennent certaines fonctions prédéfinies comme printf

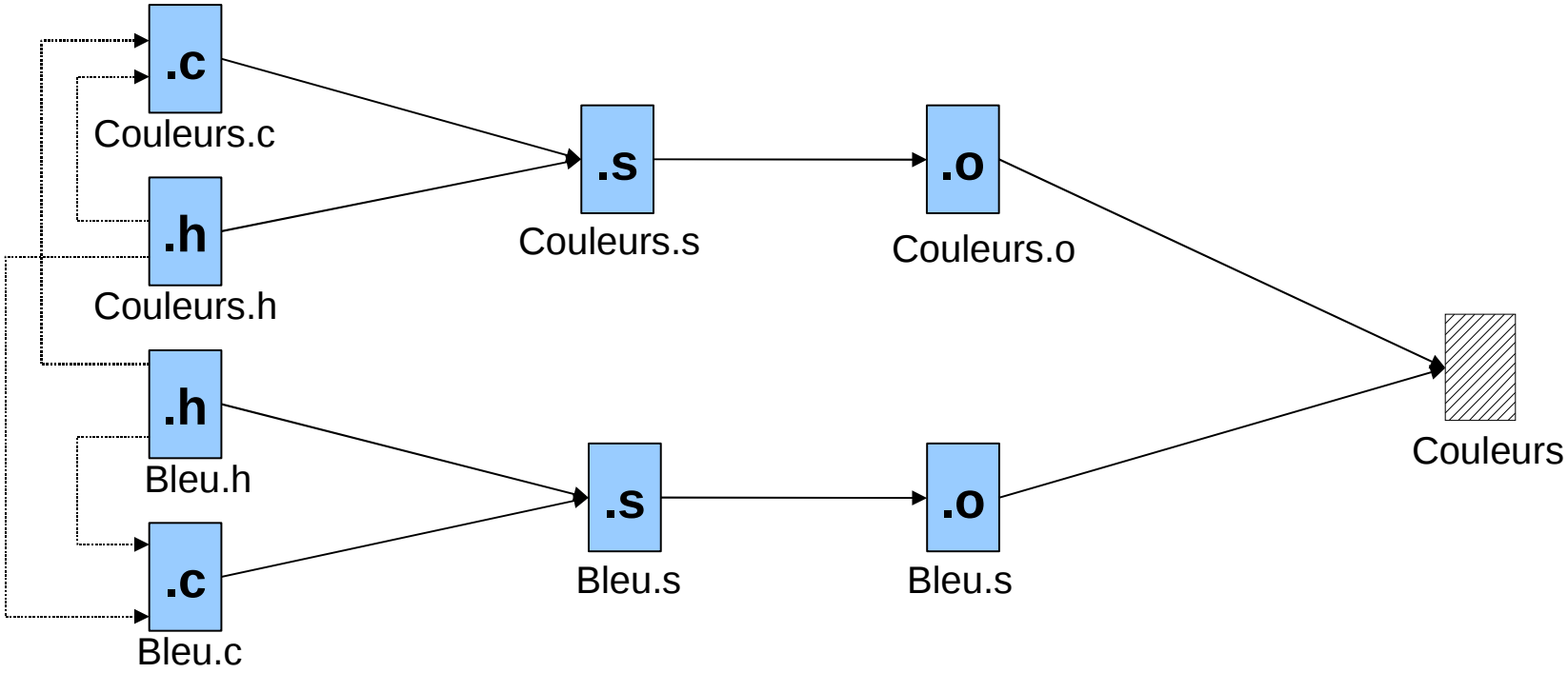
```
gcc Couleurs.o -o Couleurs
```

Retour sur l'exemple 1 : projet Couleurs



- **Compilation :** `gcc Couleurs.c Bleu.c -o Couleur`

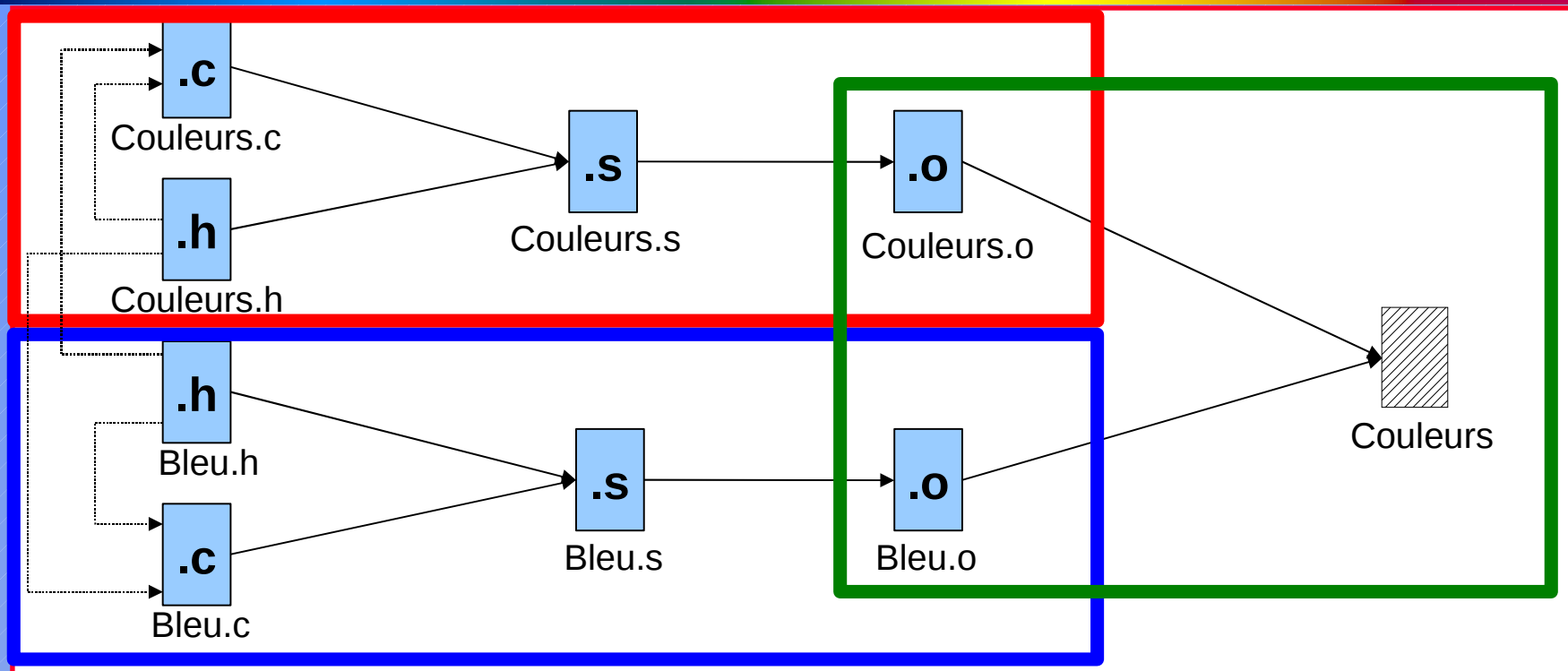
Retour sur l'exemple 1 : projet Couleurs



- **Compilation :** `gcc Couleurs.c Bleu.c -o Couleur`

- **On peut en fait séparer les étapes de compilation**

Retour sur l'exemple 1 : projet Couleurs

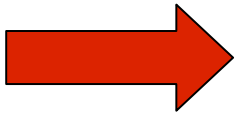


- **On peut séparer les étapes de compilation :**
  - `gcc -c Couleurs.c` **(génère Couleurs.o)**
  - `gcc -c Bleu.c` **(génère Bleu.o)**
  - `gcc Bleu.o Couleurs.o -o Couleurs` **(génère Couleurs)**

Compilation séparée : bilan

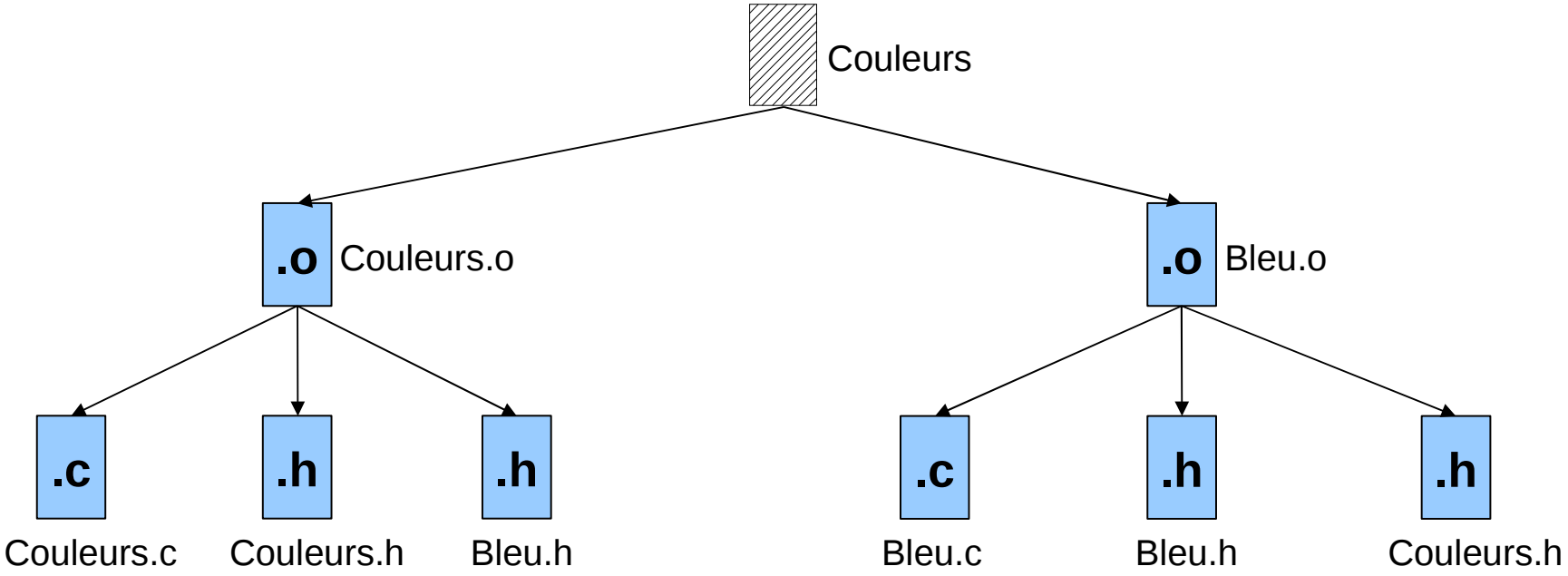
- L'exemple précédent montre que l'on n'est pas obligé de tout recompiler à chaque modification
- On recompile seulement la partie modifiée, et on relance l'étape de linkage

- **Problèmes** :
  - Comment savoir ce qui doit être recompiler
  - Comment automatiser le processus de compilation

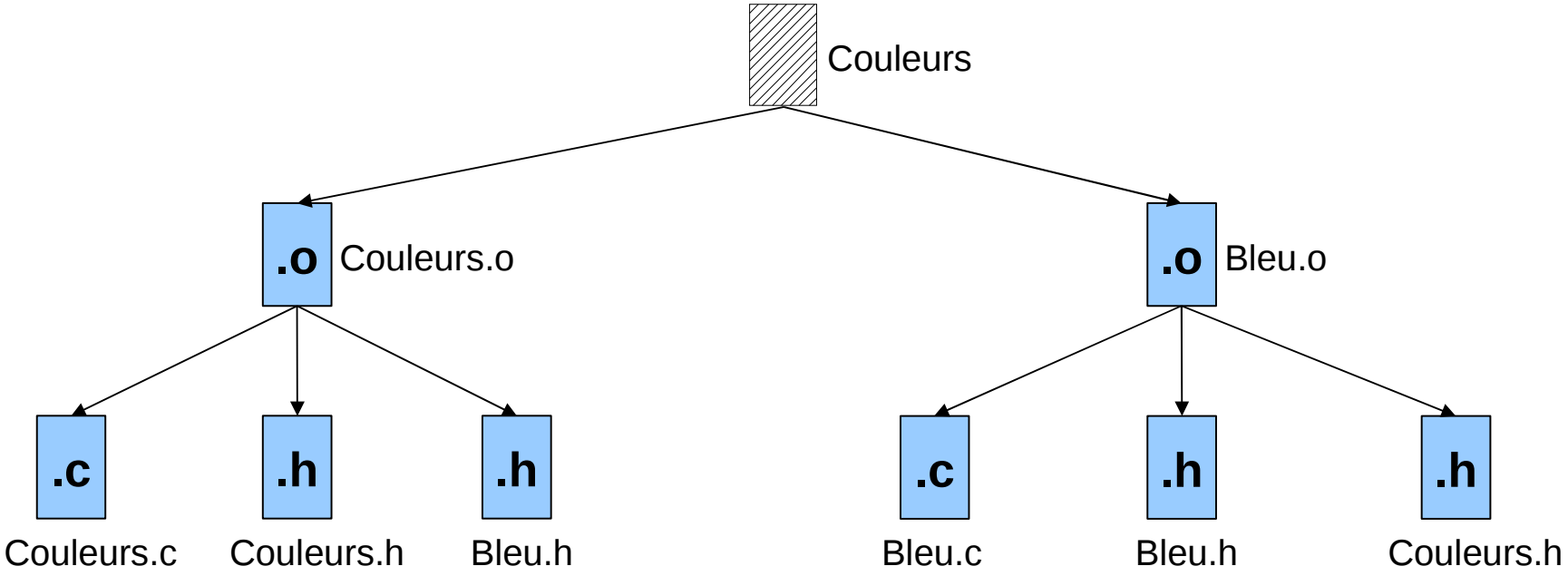


Utilitaire 'make' & fichier makefile

Graphe des dépendances



Graphe des dépendances



- **Le fichier makefile décrit le graphe de dépendances**
- **Pour chaque noeud, on liste les dépendances directes et on indique la commande qui « construit » le noeud**
- **Les feuilles ne sont pas décrites dans le makefile mais les fichiers de mêmes noms sont considérés**

Makefile : syntaxe

- **Un bloc de deux lignes par noeud du graphe :**

  `Nom_Noeud: liste_dépendances`

  `<TAB> Commande`

```
Couleurs.o:     Couleurs.c Couleurs.h Bleu.h
        gcc -c Couleurs.c

Bleu.o: Bleu.c Bleu.h Couleurs.h
        gcc -c Bleu.c

Couleurs:       Couleurs.o Bleu.o
        gcc Couleurs.o Bleu.o -o Couleurs
```

Commande 'make'

- A l'invocation de 'make'
  - la premièr noeud est évalué
- A l'invocation de 'make Nom_Noeud'
  - 'Nom_Noeud' est évalué

Commande 'make'

- A l'invocation de 'make'
  - la premièr noeud est évalué
- A l'invocation de 'make Nom_Noeud'
  - 'Nom_Noeud' est évalué

- L'évaluation d'un noeud se fait en deux étapes:

  1) Analyse de dépendances : si une dépendance correspond à un noeud du makefile, ce noeud est évalué

  2) Exécution de la commande : si 'Nom_Noeud' est moins récent que l'une de ces dépendances

+ sur la commande 'make'

- 'make' offre un certain nombre d'autres possibilités utiles pour compiler un projet
- Des macros permettent de « simplifier » la syntaxe des règles de compilation
  - Voir les liens donnés dans l'énoncé du premier TD machine
- Ce qui a été abordé dans cette séance est largement suffisant pour les TP en IGI-3004