Plan

4. Compilation

Fichiers `Makefile`

L'utilitaire `make` est paramétré par un fichier dont le nom est imposé :

« `Makefile` » ou « `makefile` ».

Il doit se trouver dans le répertoire de travail (là où se trouvent les autres fichiers du projet ou au niveau des répertoires `include` et `src`).

Ce fichier fait partie intégrante du projet.

Tout fichier `Makefile` est constitué de règles. Elles sont de la forme

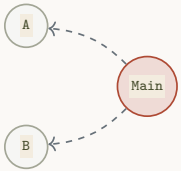
```
CIBLE: DEPENDANCES
→ COMMANDE
  ⋮
→ COMMANDE
```

Le symbole « → » désigne une tabulation.

Fichiers `Makefile` et fonctionnement

– Exemple –

Considérons le projet et le `Makefile` suivants.




```
Prog: A.o B.o Main.o
→ gcc -o Prog Main.o A.o B.o

Main.o: Main.c A.h B.h
→ gcc -c Main.c

A.o: A.c A.h
→ gcc -c A.c

B.o: B.c B.h
→ gcc -c B.c
```

Lorsque l'on exécute la commande `make`, `make` tente de résoudre la 1re règle (l. 1). Pour cela, il doit résoudre ses dépendances. Ensuite, il exécute les commandes de la règle si la cible n'est pas à jour.

Concrètement, pour pouvoir créer l'exécutable `Prog`, il est nécessaire que `A.o`, `B.o` et `Main.o` soient à jour. Une fois qu'ils le sont, il suffit de procéder à l'édition des liens (l. 2).

Fichiers `Makefile` et fonctionnement

Pour savoir si une cible est à jour, `make` regarde les dépendances de la règle et :

- si la dépendance est la cible d'une autre règle, `make` procède **récursivement** à sa résolution ;
- si la dépendance est le nom d'un fichier, `make` compare la **date de dernière modification** de la cible par rapport à celle du fichier.

S'il y a au moins un fichier dans les dépendances avec une date supérieure à celle de la cible, les commandes de la règle sont exécutées.

On peut imposer à `make` de commencer par la résolution de la règle dont la cible est `CIBLE` par

```
make CIBLE
```

Déclarations de types et dépendances

Soient `A` et `B` deux modules tels que `B` dépend (de manière étendue) à `A`, qu'un type `T` soit déclaré dans `A` et que `B` utilise ce type.

Toute modification de `A.h` doit être suivie d'une nouvelle compilation de `B`. En effet, si la déclaration de `T` a été modifiée, `B` doit être recompilé pour la prendre en compte.

En conséquence, dans le `Makefile` du projet doit figurer la règle

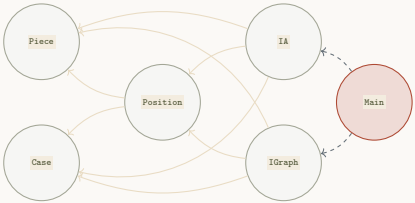
```
B.o: B.c B.h A.h
→ gcc -c B.c
```

pour déclarer que la construction de `B.o` **dépend aussi** de `A.h`.

Attention : ceci ne s'applique pas aux modifications de l'implantation des fonctions de `A` dans `A.c` (comme nous l'avons déjà vu). La cible `B.o` ne dépend ainsi pas de `A.c`. Elle ne dépend que des **déclarations** du module `A` (et donc de `A.h`).

Exemple complet de `Makefile`

– Exemple –



Le `Makefile` du projet dont le graphe d'inclusions (étendues) est donné ci-contre est

```
Echecs: Main.o Piece.o Case.o Position.o IA.o IGraph.o
→ gcc -o Echecs Main.o Piece.o Case.o Position.o IA.o IGraph.o

Main.o: Main.c IA.h IGraph.h
→ gcc -c Main.c

Piece.o: Piece.c Piece.h
→ gcc -c Piece.c
```

```
Case.o: Case.c Case.h
→ gcc -c Case.c

Position.o: Position.c Position.h Piece.h Case.h
→ gcc -c Position.c

IA.o: IA.c IA.h Piece.h Case.h Position.h
→ gcc -c IA.c

IGraph.o: IGraph.c IGraph.h Piece.h Case.h Position.h
→ gcc -c IGraph.c
```

Écriture de `Makefile` simples — résumé

Le `Makefile` d'un projet contenant des modules `A1`, …, `An` et un module principal `Main` est génériquement de la forme

```
NOM: Main.o A1.o ... An.o
→ gcc -o NOM Main.o A1.o ... An.o

Main.o: Main.c EXTRAmain
→ gcc -c Main.c

A1.o: A1.c A1.h EXTRA1
→ gcc -c A1.c

...

An.o: An.c An.h EXTRAn
→ gcc -c An.c
```

où `EXTRAmain` est la suite des noms des fichiers `.h` que `Main.c` inclut et pour tout $1 \leqslant k \leqslant n$, `EXTRAk` est la suite des noms des fichiers `.h` dont le module `Ak` dépend (de manière étendue).

4. Compilation

Variables dans les `Makefile`

Il est possible de **définir des variables** dans un `Makefile` par

```
ID=VAL
```

Ceci définit une variable identifiée par `ID`. Sa valeur est la **chaîne de caractères** `VAL`.

On accède à la valeur d'une variable identifiée par `ID` par

```
$(ID)
```

Ceci substitue à l'occurrence de `$(ID)` la chaîne de caractères qui lui a été attribuée lors de sa définition.

Variables dans les `Makefile` — exemple

Les variables permettent de factoriser les règles d'un `Makefile`.

– Exemple –

Voici un `Makefile` et sa version factorisée à l'aide des variables :

```
Main: Main.o A.o
→ gcc -o Main Main.o A.o -ansi -pedantic -Wall

Main.o: Main.c
→ gcc -c Main.c -ansi -pedantic -Wall

A.o: A.c A.h
→ gcc -c A.c -ansi -pedantic -Wall
```

```
CFLAGS=-ansi -pedantic -Wall

Main: Main.o A.o
→ gcc -o Main Main.o A.o $(CFLAGS)

Main.o: Main.c
→ gcc -c Main.c $(CFLAGS)

A.o: A.c A.h
→ gcc -c A.c $(CFLAGS)
```

Variables dans les `Makefile`

On utilise en général les **noms de variable usuels** suivants :

- `CFLAGS` pour les options de compilation (`CFLAGS=-ansi -pedantic -Wall`) ;
- `LDFLAGS` pour l'inclusion de bibliothèques (`LDFLAGS=-lm -lMLV`) ;
- `CC` pour le compilateur utilisé (`CC=gcc` ou bien `CC=clang`) ;
- `OPT` pour les option d'optimisation de code (`OPT=-O1` ou bien `OPT=-O2` ou encore `OPT=-O3`).

Règles courantes

Observation : la plupart des règles des `Makefile` sont sous l'une de ces deux formes :

1.
```
M.o: M.c DEP2... DEPn
→ gcc -c M.c
```

2.
```
EXEC: DEP1 ... DEPn
→ gcc -o EXEC DEP1 ... DEPn
```

La 1re forme de règle a pour but de construire le fichier objet d'un module `M`. Dans ce cas, `DEP2`, …, `DEPn` sont les `.h` dont le module `M` dépend.

La 2e forme de règle a pour but de construire l'exécutable `EXEC` du projet. Les dépendances `DEP1`, …, `DEPn` sont dans ce cas les fichiers `.o` du projet.

Variables internes dans les `Makefile`

Il est possible de simplifier l'écriture de ces règles courantes au moyen des variables internes. Il y en a trois principales et deux secondaires :

Symbole	Ce qu'il désigne
`$@`	Nom de la cible
`$<`	Nom de la 1re dép.
`$^`	Noms de toutes les dép.
`$?`	Noms des dép. plus récentes que la cible
`$*`	Nom de la cible sans extension

– Exemples –

Sans variable interne.

```
M.o: M.c DEP2 ... DEPn
→ gcc -c M.c

EXEC: DEP1 ... DEPn
→ gcc -o EXEC DEP1 ... DEPn
```

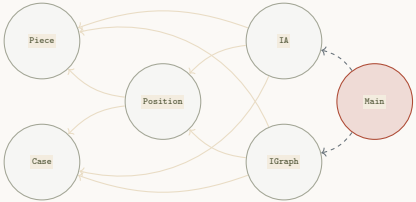
Avec variables internes.

```
M.o: M.c DEP2 ... DEPn
→ gcc -c $<

EXEC: DEP1 ... DEPn
→ gcc -o $@ $^
```

Variables internes dans les `Makefile` — exemple

– Exemple –



L'utilisation des variables et variables internes permet de simplifier le `Makefile` de ce projet vu précédemment.

```
CC=gcc
CFLAGS=-ansi -pedantic -Wall
OBJ=Main.o Piece.o Case.o Position.o IA.o IGraph.o

Echecs: $(OBJ)
→ $(CC) -o $@ $^ $(CFLAGS)

Main.o: Main.c IA.h IGraph.h
→ $(CC) -c $< $(CFLAGS)

Piece.o: Piece.c Piece.h
→ $(CC) -c $< $(CFLAGS)
```

```
Case.o: Case.c Case.h
→ $(CC) -c $< $(CFLAGS)

Position.o: Position.c Position.h Piece.h Case.h
→ $(CC) -c $< $(CFLAGS)

IA.o: IA.c IA.h Piece.h Case.h Position.h
→ $(CC) -c $< $(CFLAGS)

IGraph.o: IGraph.c IGraph.h Piece.h Case.h Position.h
→ $(CC) -c $< $(CFLAGS)
```

Règles génériques

Il est possible de simplifier encore d'avantage l'écriture des `Makefile` au moyen des règles génériques.

Ce sont des règles de la forme

```
%.o: %.c
→ COMMANDE
```

où `COMMANDE` est une commande.

Cette syntaxe simule une règle

```
M.o: M.c
→ COMMANDE
```

pour chaque fichier `M.c` du projet.

Intérêt principal : l'**unique règle**

```
%.o: %.c
→ gcc -c $<
```

permet de **construire** chaque **fichier objet** du projet.

Règles génériques

Attention : la règle

```
%.o: %.c
→ gcc -c $<
```

ne prend pas en compte des dépendances des modules aux fichiers `.h` concernés.

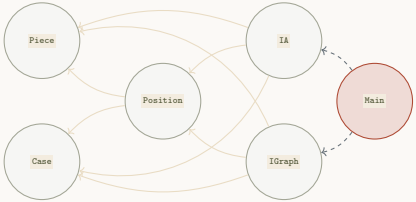
Il faut les **mentionner explicitement** de la manière suivante :

```
M.o: M.c M.h DEP1 ... DEPn
```

pour chaque module `M` du projet. `DEP1`, …, `DEPn` sont les `.h` dont le module `M` dépend.

Règles génériques — exemple

– Exemple –



L'utilisation des règles génériques permet de simplifier encore les `Makefile`.

```
CC=gcc
CFLAGS=-ansi -pedantic -Wall
OBJ=Main.o Piece.o Case.o Position.o IA.o IGraph.o

Echecs: $(OBJ)
→ $(CC) -o $@ $^ $(CFLAGS)

Main.o: Main.c IA.h IGraph.h

Piece.o: Piece.c Piece.h
```

```
Case.o: Case.c Case.h

Position.o: Position.c Position.h Piece.h Case.h

IA.o: IA.c IA.h Piece.h Case.h Position.h

IGraph.o: IGraph.c IGraph.h Piece.h Case.h Position.h

%.o: %.c
→ $(CC) -c $< $(CFLAGS)
```

Règles de nettoyage

Règle de nettoyage :

```
clean:
→ rm -f *.o
```

Cette règle permet, lorsque l'on saisit la commande `make clean`, de supprimer les fichiers `.o` du répertoire courant.

Règle de nettoyage total :

```
mrproper: clean
→ rm -f EXEC
```

Cette règle, où `EXEC` est le nom de l'exécutable du projet, permet de supprimer tous les fichiers regénérables (c.-à-d. les fichiers `.o` et l'exécutable) à partir des fichiers `.c` et `.h` du projet.

Règles d'installation / désinstallation

Règle d'installation :

```
install: EXEC
→ mkdir ../bin
→ mv EXEC ../bin/EXEC
→ make mrproper
```

Cette règle permet de compiler le projet, de placer son exécutable `EXEC` dans un répertoire `bin` et de supprimer les fichiers regénérables.

Règle de désinstallation :

```
uninstall: mrproper
→ rm -f ../bin/EXEC
→ rm -rf ../bin
```

Cette règle permet de supprimer les fichiers regénérables, l'exécutable du projet, ainsi que le répertoire `bin` le contenant.

Plan

4. Compilation

Principes généraux

Une bibliothèque est un **regroupement de modules** offrant des fonctionnalités allant vers un même objectif.

– Exemples –

- La bibliothèque graphique MLV est un ensemble de plusieurs modules (`MLV_audio`, `MLV_keyboard`, `MLV_image`, *etc.*) réunis dans le but d'offrir des fonctions d'affichage graphique et de gérer les entrées / sorties (son, clavier, souris, *etc.*).
- La bibliothèque NCURSES offre divers outils pour concevoir des applications en mode texte.

L'intérêt des bibliothèques est double :

1. il suffit de réaliser une inclusion d'**un seul en-tête** pour bénéficier des fonctionnalités d'une bibliothèque, plutôt que des inclusions de plusieurs en-têtes sans être sûr du fichier précis à inclure ;
2. sous réserve de savoir comment créer des bibliothèques, il est possible de **partager et réutiliser** son propre code entre plusieurs de ses projets, sans avoir à le recompiler.

Bibliothèques statiques

Une bibliothèque statique est un fichier d'extension `.a`.

Lors de son utilisation, son **code est inclus dans l'exécutable** pendant l'édition des liens.

- **Avantage** : tout projet qui utilise une bibliothèque statique peut être exécuté sur une machine où la bibliothèque n'est pas installée.
- **Inconvénient** : l'exécutable est plus volumineux.

Bibliothèques dynamiques

Une bibliothèque dynamique est un fichier d'extension `.so`.

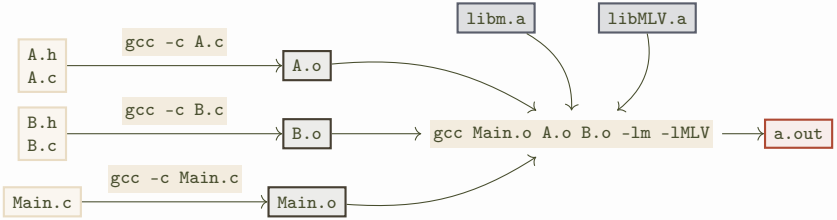
Lors de son utilisation, son code n'est pas inclus dans l'exécutable. C'est lors de l'exécution que les symboles provenant de la bibliothèque sont résolus au moyen de l'**éditeur de liens dynamique**.

- **Avantages** : l'exécutable est moins volumineux. Il n'y a pas de duplication du code de la bibliothèque sur un même système si plusieurs projets l'utilisent.
- **Inconvénient** : tout projet qui utilise une bibliothèque dynamique ne peut être exécuté que sur un système où cette dernière est installée.

Compilation d'un projet utilisant des bibliothèques

On suppose que l'on travaille sur un projet constitué de deux modules `A` et `B` et d'un fichier principal `Main.c`. Ce projet utilise deux bibliothèques statiques : `libm.a` et `libMLV.a`.

La compilation de ce projet se réalise de manière habituelle. La différence porte sur l'étape d'**édition des liens** dans laquelle il est nécessaire de **signaler les bibliothèques utilisées**.



Pour utiliser des bibliothèques qui ne sont pas situées dans le répertoire standard des bibliothèques, on précisera leur chemin `chem` lors de l'édition des liens au moyen de l'option `-Lchem`.

Compilation d'un projet utilisant des bibliothèques

Le nom d'une bibliothèque commence par `lib`. On signale l'utilisation d'une bibliothèque à l'éditeur de liens par `-lNOM` où `libNOM` est le nom de la bibliothèque.

– Exemple –

En supposant que le graphe d'inclusions (étendues) du projet précédent est celui ci-contre, un `Makefile` possible est


```
CC=gcc
CFLAGS=-ansi -pedantic -Wall
LDFLAGS=-lm -lncurses
OBJ=Main.o A.o B.o

Projet: $(OBJ)
→ $(CC) -o $@ $^ $(CFLAGS) $(LDFLAGS)

Main.o: Main.c A.h B.h
A.o: A.c A.h
B.o: B.c B.h A.h

%.o: %.c
→ $(CC) -c $< $(CFLAGS)
```

```
clean:
→ rm -f *.o

mrproper: clean
→ rm -f Projet

install: Projet
→ mkdir ../bin
→ mv Projet ../bin/Projet
→ make mrproper

uninstall: mrproper
→ rm -f ../bin/Projet
→ rm -rf ../bin
```

La bibliothèque standard

La bibliothèque standard `libc.a` (`libc.so`) regroupe les vingt-quatre modules

`assert`,	`complex`,	`ctype`,	`errno`,
`fenv`,	`float`,	`inttypes`,	`iso646`,
`limits`,	`locale`,	`math`,	`setjmp`,
`signal`,	`stdarg`,	`stdbool`,	`stddef`,
`stdint`,	`stdio`,	`stdlib`,	`string`,
`tgmath`,	`time`,	`wchar`,	`wctype`.

Cette bibliothèque est **liée implicitement** lors de toute édition des liens.

Il est donc possible d'utiliser certains des modules de la bibliothèque standard simplement en les incluant (`#include <NOM.h>`), sans avoir à utiliser l'option `-lc`.

L'utilisation de certains modules doit cependant s'accompagner de l'option `-lX` (comme `-lm` pour utiliser `math.h`).