Chap. 8
Modularisation

I2181A
Langage C :
modularisation

Anthony Legrand
José Vander Meulen
Olivier Choquet

La découpe en module

Module =

- une boîte à outils comme une interface java

et

- une classe qui implémente cette interface

Définir un module

- Un fichier entête (extension .h)

  ≈ une interface en Java

- Un fichier source (extension .c)

  ≈ une classe qui implémente cette interface

Les fichiers d'entête

Un **header** comprend toutes les informations **publiques** du module

- définitions de constantes
- définitions de types
- prototypes de fonctions
- spécifications des fonctions
- Attention: pas de code!

Exemple de header


```
#ifndef _PILE_H_
#define _PILE_H_

/* définition de constantes */
#define OK  1

/* définition de types */
typedef ... Pile;

/* déclaration de fonctions */
Pile init ();
int pop (Pile* p);
void push (Pile* p, int val);

#endif   // _PILE_H_
```

Exemple de header


```
#ifndef _PILE_H_
#define _PILE_H_

/* définition de constantes */
#define OK  1

/* définition de types */
typedef ... Pile;

/* déclaration de fonctions */
Pile init ();
int pop (Pile* p);
void push (Pile* p, int val);

#endif   // _PILE_H_
```

+ specs

Eviter la double inclusion

Des directives conditionnelles du préprocesseur permettent d'éviter une double inclusion de modules (lors de l'édition de liens)

```
#ifndef _PILE_H_
#define _PILE_H_

...

#endif    // _PILE_H_
```

Les fichiers sources: inclusions


```
/* fichier pile.c */

#include <stdio.h>
#include <stdlib.h>
#include "pile.h"
```

Les fichiers sources: inclusions


```
/* fichier pile.c */

#include <stdio.h>
#include <stdlib.h>
#include "pile.h"
```

Librairies standard

Répertoire courant

Les fichiers sources: implémentation


```
Pile init () {
   ...
}

int pop (Pile* p) {
   ...
}

void push (Pile* p, int val) {
   ...
}
```

Construction d'une application

La génération d'une application se fait en deux étapes :

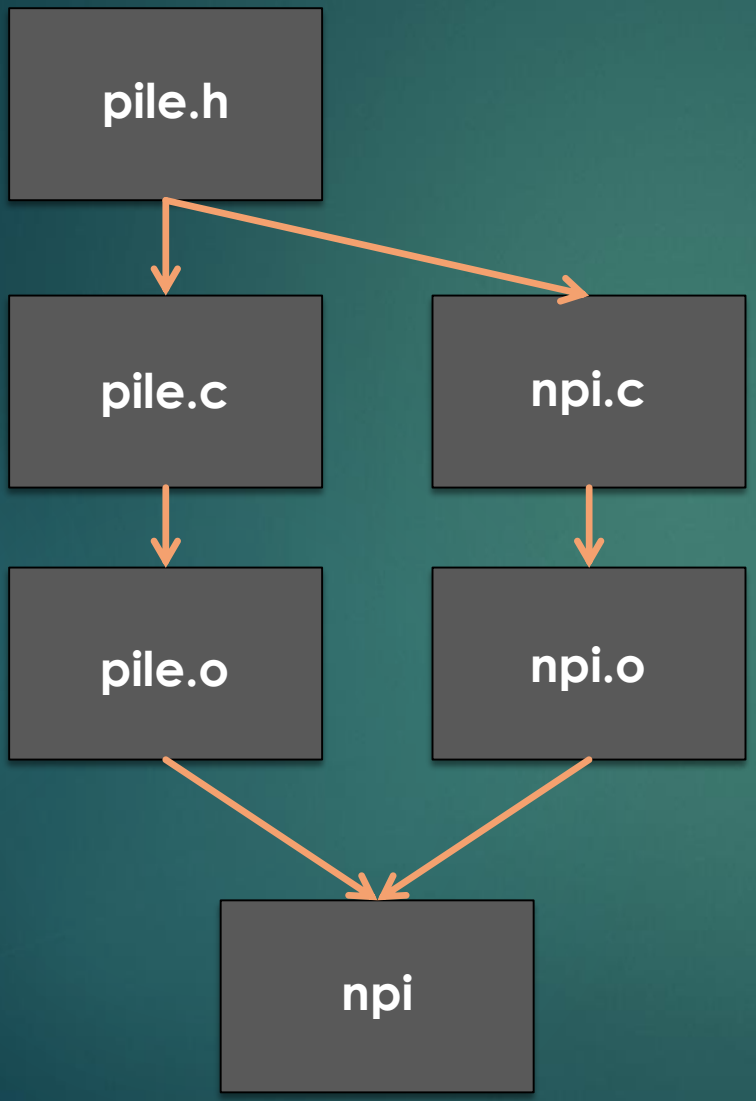
1. **compilation**

   après avoir effectué la précompilation, le compilateur produit des fichiers objets .o en compilant les différentes sources

2. **édition des liens (*linkage*)**

   produit un exécutable en assemblant les fichiers objets .o

Construction d'une application

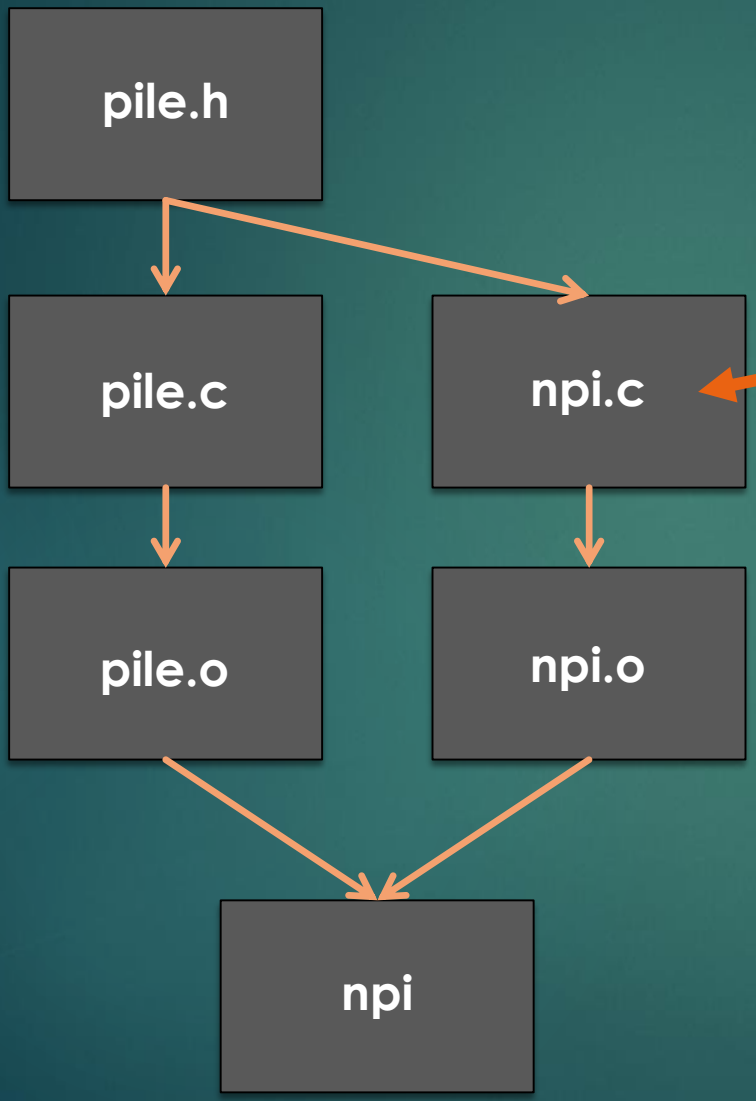


Précompilation (#include, #define…)

Compilation

Edition des liens

Construction d'une application



Précompilation (#include, #define…)

main() ⇒ pas de npi.h

Compilation

Edition des liens

Makefile : Objectifs

- Un *makefile* permet de ne construire que ce qui n'est plus « à jour ».
- La compilation séparée permet de mieux organiser ses projets et de développer des bibliothèques de fonctions.
- Utile pour le développement de grosses applications.

Makefile


```
CFLAGS=-std=c11 -pedantic -Wvla -Werror -Wall

npi : npi.o pile.o
   cc $(CFLAGS) -o npi npi.o pile.o

npi.o : npi.c pile.h
   cc $(CFLAGS) -c npi.c

pile.o : pile.c pile.h
   cc $(CFLAGS) -c pile.c

clean :
   rm *.o
   rm npi
```

Makefile


```
CFLAGS=-std=c11 -pedantic -Wvla -Werror -Wall

npi : npi.o pile.o
   cc $(CFLAGS) -o npi npi.o pile.o

npi.o : npi.c pile.h
   cc $(CFLAGS) -c npi.c

pile.o : pile.c pile.h
   cc $(CFLAGS) -c pile.c

clean :
   rm *.o
   rm npi
```

options de compilation

Makefile


```
CFLAGS=-std=c11 -pedantic -Wvla -Werror -Wall

npi : npi.o pile.o
   cc $(CFLAGS) -o npi npi.o pile.o

npi.o : npi.c pile.h
   cc $(CFLAGS) -c npi.c

pile.o : pile.c pile.h
   cc $(CFLAGS) -c pile.c

clean :
   rm *.o
   rm npi
```

règle d'édition de liens

Makefile


```
CFLAGS=-std=c11 -pedantic -Wvla -Werror -Wall

npi : npi.o pile.o
   cc $(CFLAGS) -o npi npi.o pile.o

npi.o : npi.c pile.h
   cc $(CFLAGS) -c npi.c

pile.o : pile.c pile.h
   cc $(CFLAGS) -c pile.c

clean :
   rm *.o
   rm npi
```

règles de compilation

Makefile


```
CFLAGS=-std=c11 -pedantic -Wvla -Werror -Wall

npi : npi.o pile.o
   cc $(CFLAGS) -o npi npi.o pile.o

npi.o : npi.c pile.h
   cc $(CFLAGS) -c npi.c

pile.o : pile.c pile.h
   cc $(CFLAGS) -c pile.c

clean :
   rm *.o
   rm npi
```

règle utilitaire

Makefile


```
CFLAGS=-std=c11 -pedantic -Wvla -Werror -Wall

npi : npi.o pile.o
   cc $(CFLAGS) -o npi npi.o pile.o

npi.o : npi.c pile.h
   cc $(CFLAGS) -c npi.c

pile.o : pile.c pile.h
   cc $(CFLAGS) -c pile.c

clean :
   rm *.o
   rm npi
```

cible (target)

Makefile

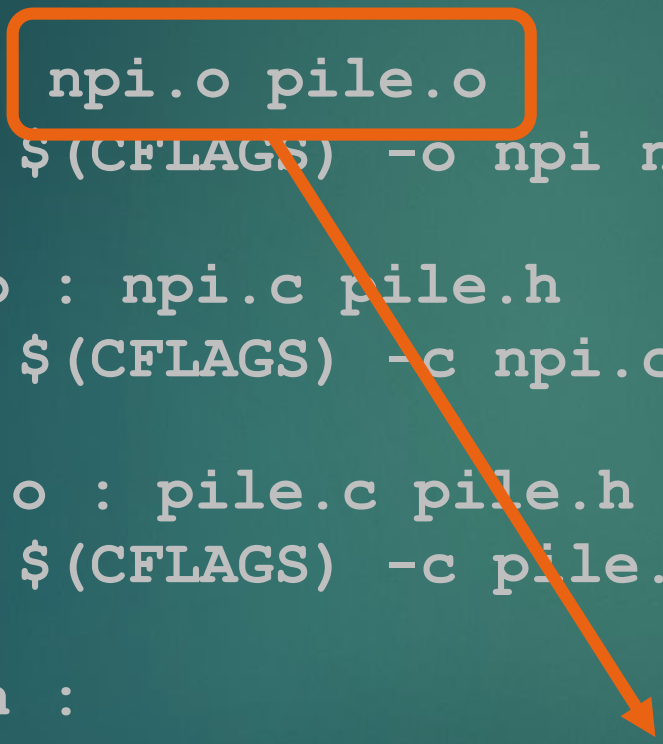

```
CFLAGS=-std=c11 -pedantic -Wvla -Werror -Wall

npi : npi.o pile.o
   cc $(CFLAGS) -o npi npi.o pile.o

npi.o : npi.c pile.h
   cc $(CFLAGS) -c npi.c

pile.o : pile.c pile.h
   cc $(CFLAGS) -c pile.c

clean :
   rm *.o
   rm npi
```



dépendances

Makefile


```
CFLAGS=-std=c11 -pedantic -Wvla -Werror -Wall

npi : npi.o pile.o
   cc $(CFLAGS) -o npi npi.o pile.o

npi.o : npi.c pile.h
   cc $(CFLAGS) -c npi.c

pile.o : pile.c pile.h
   cc $(CFLAGS) -c pile.c

clean :
   rm *.o
   rm npi
```

Commande exécutée si dépendance plus récente que cible ou cible non existante

Makefile

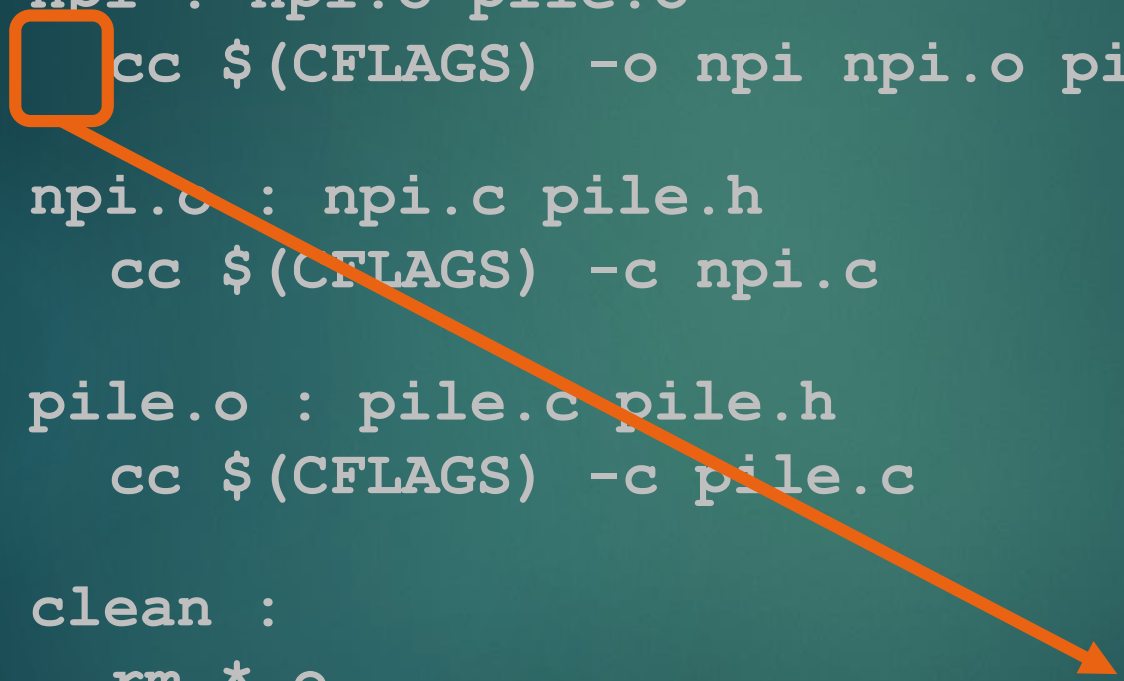

```
CFLAGS=-std=c11 -pedantic -Wvla -Werror -Wall

npi : npi.o pile.o
    cc $(CFLAGS) -o npi npi.o pile.o

npi.o : npi.c pile.h
    cc $(CFLAGS) -c npi.c

pile.o : pile.c pile.h
    cc $(CFLAGS) -c pile.c

clean :
    rm *.o
    rm npi
```



Tabulation obligatoire!

Makefile

Liste de règles qui ont

- une condition de dépendance (optionnelle)

```
npi : npi.o pile.o
```

- une action (optionnelle)

```
cc $(CFLAGS) -o npi npi.o pile.o
```

Commande make

- **`make`**

  mise à jour de **`npi`**

- **`make npi.o`**

  mise à jour de **`npi.o`**

- **`make clean`**

  provoque l'effacement des modules objets et de l'exécutable → à exécuter avant publication/partage du code ou après modification du *makefile* lui-même

Makefile avec règle ‘all’


```
CFLAGS=-std=c11 -pedantic -Wvla -Werror -Wall

all : npi tree

tree : tree.o pile.o
   cc $(CFLAGS) -o tree tree.o pile.o

npi : npi.o pile.o
   cc $(CFLAGS) -o npi npi.o pile.o

...
```

- **`make all`** ou **`make`**

  mise à jour de tous les exécutables

- Plus d'infos sur `make` et `makefile` :

  https://linuxpedia.fr/doku.php/dev/makefile

Différents types de modules

Distinction entre des modules :

- **bibliothèque**

  fichiers .h et .c mais pas de main() = « boîte à outils »

- **configuration**

  uniquement header .h = fichier définissant des constantes, des macros et/ou des types

- **application**

  uniquement source .c avec main() = fichier permettant de générer un programme exécutable