Programmation Unix 2 – cours n°9

Edouard THIEL

Faculté des Sciences

Université d'Aix-Marseille (AMU)

Janvier 2017

Les transparents de ce cours sont téléchargeables ici :
http://pageperso.lif.univ-mrs.fr/~edouard.thiel/ens/unix/

Lien court : http://j.mp/progunix

Plan du cours n°9

Compilation séparée et `Makefile` :

1. La découpe en modules C
2. Compilation séparée
3. Arbre des dépendances
4. Compilation conditionnelle

1 - La découpe en modules C

Découper un programme C en modules :

- plus lisible ;
- mise au point indépendante ;
- réutilisation pour d'autres programmes ;
- plus besoin de tout recompiler lors de modifications.

Module C

Un module = 1 fichier `.c` [+ 1 fichier `.h`]

- Le fichier `.h` contient :
  - constantes et macros
  - types
  - variables externes
  - prototypes des fonctions

- Le fichier `.c` contient :
  - variables globales au module
  - corps des fonctions

Distinction publique / privé

- Fonction ou donnée publique :
  - sert à d'autres modules ;
  - documentée, versionnée ;
  - fait partie de l'API (interface de programmation).

- Fonction ou donnée privée :
  - ne sert qu'au module lui-même ;
  - peut être modifiée sans préavis ;
  - utilisation externe dangereuse.

Module C + distinction publique / privé

- Le fichier `.h` contient l'API publique :
  - constantes et macros publiques
  - types publique
  - variables externes publiques
  - prototypes des fonctions publiques

- Le fichier `.c` contient l'implémentation publique + une partie privée éventuelle :
  - constantes, macros et types privés
  - variables globales au module
  - corps des fonctions publiques et privées

Exemple 1 / 3

Fichier `max.h` :

```
#ifndef MAX_H
#define MAX_H

int max (int a, int b);

#endif // MAX_H
```

Fichier `max.c` :

```
#include "max.h"

int max (int a, int b)
{
    if (a > b) return a;
    return b;
}
```

`#ifndef .. #define .. #endif` = une *garde*.

But : protéger de l'inclusion multiple.

Exemple 2 / 3

Fichier `point.h` :

```
#ifndef POINT_H
#define POINT_H

typedef struct {
    int x, y;
} Point ;

void affi_point (Point p);

#endif // POINT_H
```

Fichier `point.c` :

```
#include <stdio.h>
#include "point.h"

void affi_point (Point p)
{
    printf ("(%d,%d)\n",
        p.x, p.y);
}
```

Différence entre `#include <..>` et `#include ".."`

Exemple 3 / 3

Fichier `principal.c` :

```
#include <stdio.h>
#include "max.h"
#include "point.h"

int main ()
{
    Point p = { 3, 4 };

    affi_point (p);

    printf ("max = %d\n", max (p.x, p.y));
    return 0;
}
```

Chemin d'inclusion

Rajouter des chemins aux répertoires standards pour les `.h` :

```
gcc -Ichemin1 -Ichemin2 ...  fichier.c ...
```

Exemple pour le module `point` :

au lieu de `#include "point.h"`

on peut faire `#include <point.h>`
et `gcc -I. point.c ...`

2 - Compilation séparée

Chaque fichier `.c` produit un fichier `.o`

Pour les produire : option `-c`

```
gcc -Wall -W -std=c99 -c max.c
gcc -Wall -W -std=c99 -c point.c
gcc -Wall -W -std=c99 -c principal.c
```

Dans n'importe quel ordre.

Contenu d'un fichier `.o` : commande `nm`

```
$ nm max.o
0000000000000000 T max

$ nm point.o
0000000000000000 T affi_point
                 U printf

$ nm principal.o
                 U affi_point
0000000000000000 T main
                 U max
                 U printf
```

Colonne 1 : adresse relative du symbole (ici 64 bits)
Colonne 2 : `U` : symbole indéfini / `T` : symbole défini
Colonne 3 : liste des fonctions appelées

Production d'un exécutable

Phase appelée *édition de lien* :

- assemblage des `.o`
- résolution des symboles : chaque symbole défini une seule fois
- réadressage des symboles
- suppression des symboles inutiles
- ajout des bibliothèques (`libc`, ...)
- ajout de `_start`
- ajout d'un chargeur, selon format (ELF, ...)

Production d'un exécutable

Avec `gcc` :

```
gcc max.o point.o principal.o -o ex1
```

Nom de l'exécutable : par défaut `a.out`

En interne, `gcc` fait appel à `ld` (appelé le *linker*, l'éditeur de liens)

C'est `ld` qui signale les erreurs :

- référence indéfinie
- référence multiple

Exemples d'erreurs

```
$ gcc point.o principal.o -o ex1
principal.o : Dans la fonction "main" :
principal.c:(.text+0x32) : référence indéfinie vers "max"
collect2: error: ld returned 1 exit status

$ gcc max.o max.o point.o principal.o -o ex1
max.o : Dans la fonction "max" :
max.c:(.text+0x0) : définitions multiples de "max"
max.o:max.c:(.text+0x0) : défini pour la première fois ici
collect2: error: ld returned 1 exit status
```

Symboles dans l'exécutable

```
$ nm ex1
...
0000000000400549 T affi_point
000000000040056e T main
000000000040052d T max
                 U printf@@GLIBC_2.2.5
0000000000400440 T _start
...
```

`printf` n'est pas défini dans l'exécutable, mais dans la bibliothèque `libc` :

```
$ ldd ex1
    linux-vdso.so.1 =>  (0x00007fffb07fe000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x...)
    /lib64/ld-linux-x86-64.so.2 (0x00007f3cdda76000)
```

Bibliothèque (library)

Bibliothèque = fichier contenant un ou plusieurs `.o` :

Fichier `"libxxx.a"` : statique
recopiée dans l'exécutable

Fichier `"libxxx.so"` : dynamique ($\simeq$ DLL)
chargée à l'exécution
une seule fois pour tous les exécutables

Manipuler les bibliothèques

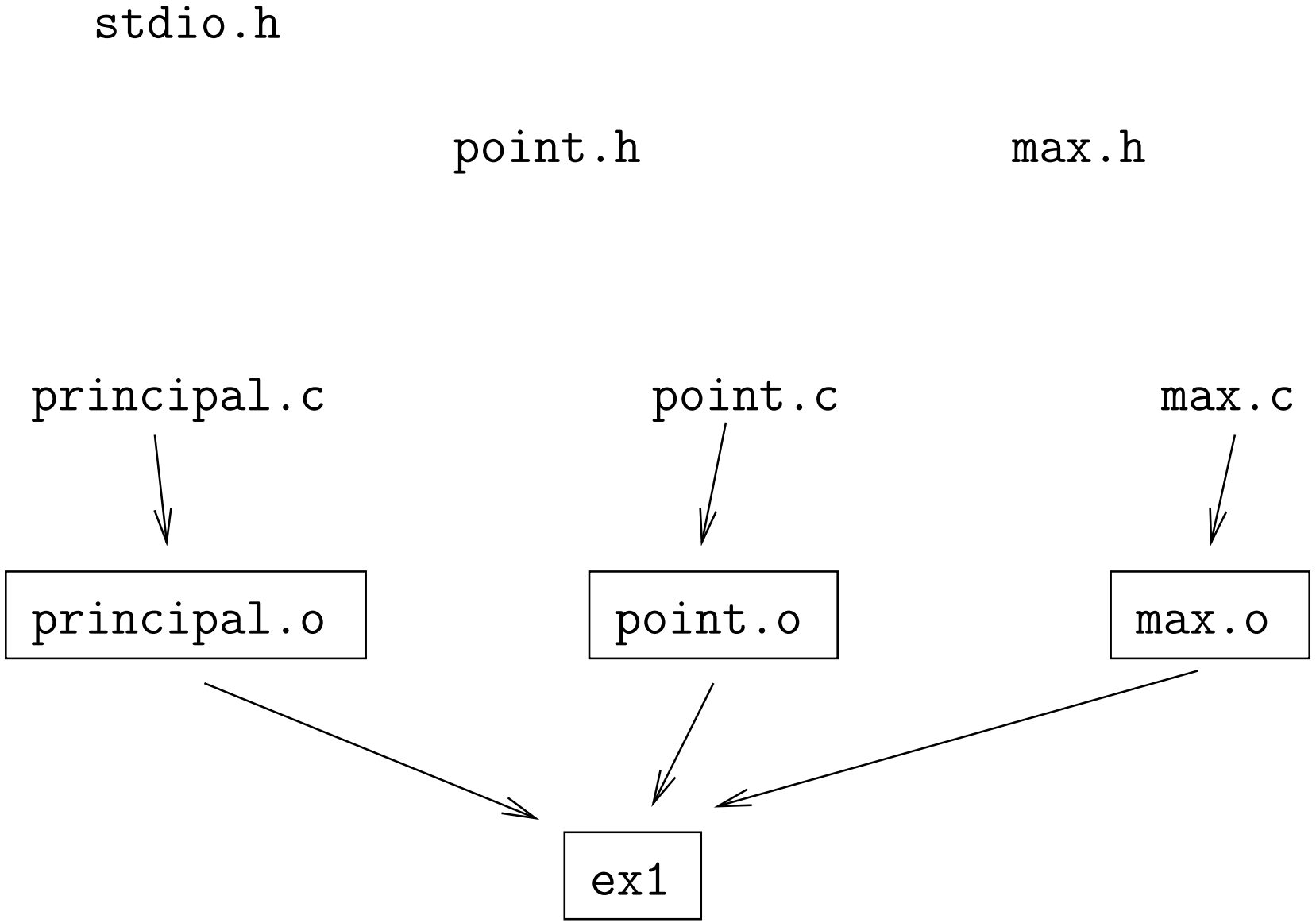
- Construire des bibliothèques : `ar`, `ranlib`
- Compiler avec la bibliothèque `libtoto.a` ou `libtoto.so` située dans `chemin` :

```
gcc ... -ltoto -Lchemin ...
```

- Voir les bibliothèques demandées par un exécutable : `ldd`
- Rajouter un chemin de bibliothèque dynamique à l'exécution :

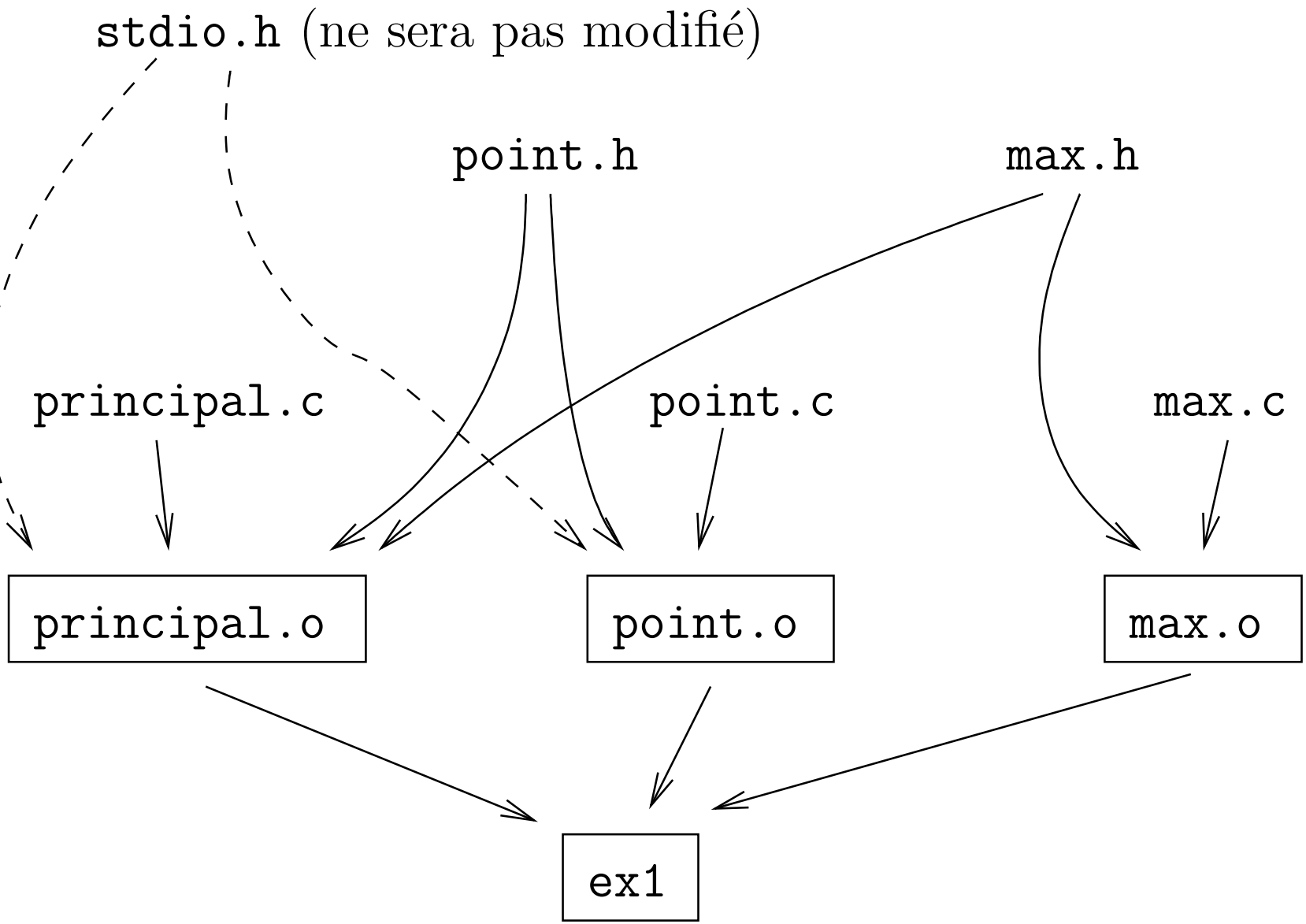
```
export LD_LIBRARY_PATH=chemin
```

3 - Arbre des dépendances



Flêches vers les fichiers fichiers produits (boîtes)

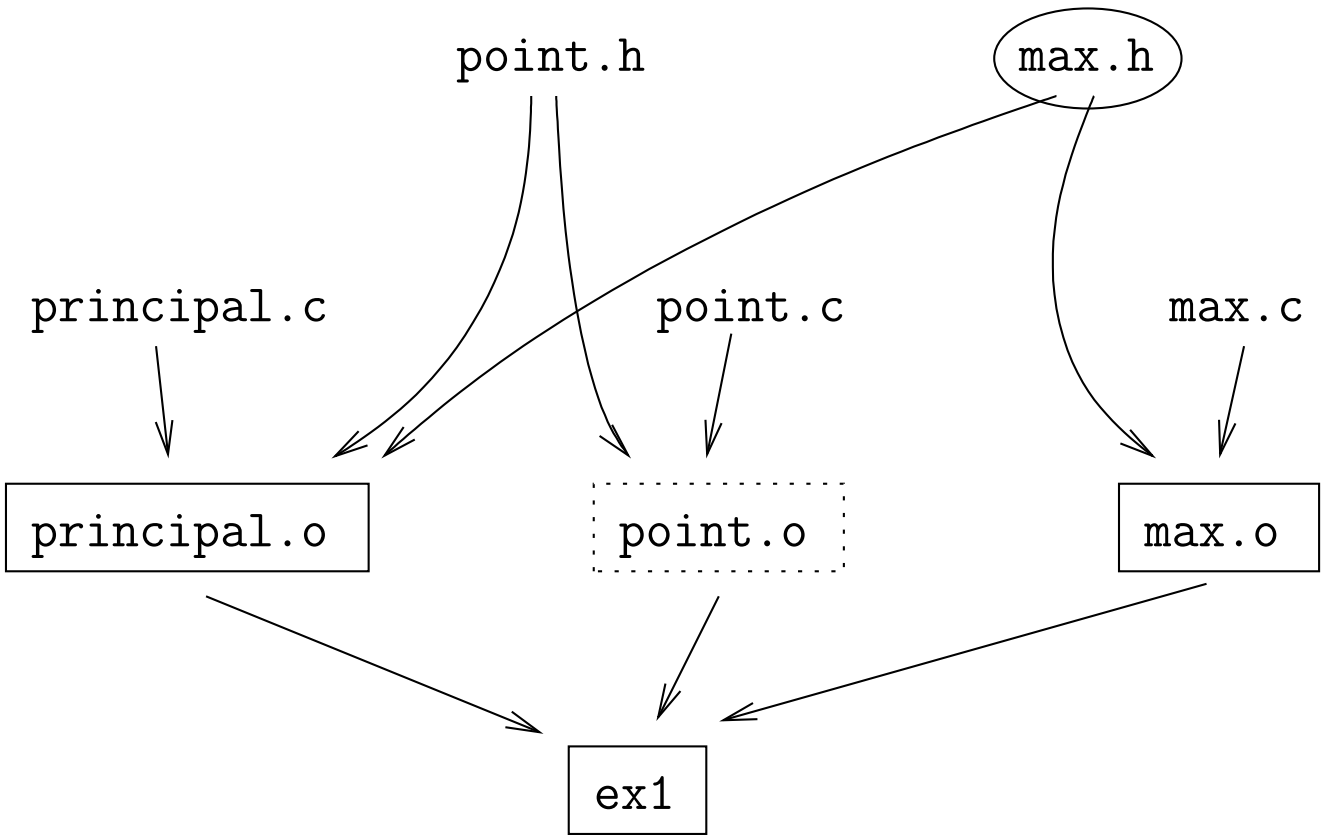
Dépendances avec les `.h`



Utilité

Si je modifie `max.h`, je dois

- recompiler `principal.c` et `max.c`, puis refaire l'exécutable
- mais pas recompiler `point.c`



4 - Compilation conditionnelle

Idée :

- formaliser les dépendances
- éventuellement, calculer les dépendances des `.h`
- automatiser le calcul des recompilations nécessaires

→ Fichier `Makefile` et commande `make`

Fichier `Makefile`

Le fichier `Makefile` est un fichier texte

- placé dans le même répertoire
- on y décrit les dépendances avec des règles
- utilisé par la commande `make`

Syntaxe générale d'une règle :

```
cible :  dépendances
tabulation    commande pour produire la cible
```

Exemple

Fichier `Makefile` :

```
# Makefile de mon programme

ex1 : principal.o point.o max.o
    gcc principal.o point.o max.o -o ex1

principal.o : principal.c point.h max.h
    gcc -Wall -W -std=c99 -c principal.c

point.o : point.c point.h
    gcc -Wall -W -std=c99 -c point.c

max.o : max.c max.h
    gcc -Wall -W -std=c99 -c max.c
```

Utilisation

Pour produire l'exécutable, taper : `make`

Que fait cette commande ?

- elle cherche `Makefile` ou `makefile` dans le répertoire courant ;
- elle cherche la première cible (ici `ex1`)
- puis récursivement :
  - pour chaque dépendance, va à la règle correspondante ;
  - si la dépendance est plus récente que la cible, reconstruit avec la commande.

Exemple (1/2)

On modifie `max.h`, puis on tape `make`

- Première cible : `ex1`
- Va aux cibles :
  - `principal.o`
    - `principal.c`, `point.h`, `max.h` ne sont pas des cibles
    - `max.h` plus récent que `principal.o` $\rightarrow$ recompile `principal.o`
  - `point.o`
    - `point.c` et `point.h` ne sont pas des cibles
    - rien plus récent

```
ex1 : principal.o point.o max.o
    gcc principal.o point.o max.o -o ex1
principal.o : principal.c point.h max.h
    gcc -Wall -W -std=c99 -c principal.c
point.o : point.c point.h
    gcc -Wall -W -std=c99 -c point.c
max.o : max.c max.h
    gcc -Wall -W -std=c99 -c max.c
```

Exemple (2/2)

- Va aux cibles (suite)
  - `max.o`
    - `max.c` et `max.h` ne sont pas des cibles
    - `max.h` plus récent que `max.o` $\rightarrow$ recompile `max.o`
- `principal.o` et `max.o` plus récents que `ex1` $\rightarrow$ recompile `ex1`
- Stop.

```
ex1 : principal.o point.o max.o
    gcc principal.o point.o max.o -o ex1
principal.o : principal.c point.h max.h
    gcc -Wall -W -std=c99 -c principal.c
point.o : point.c point.h
    gcc -Wall -W -std=c99 -c point.c
max.o : max.c max.h
    gcc -Wall -W -std=c99 -c max.c
```

Choisir la cible

On peut demander de construire une ou plusieurs cibles :

```
$ make ex1
$ make point.o max.o
```

Si tout est à jour, `make` ne fait rien :

```
$ make ex1
make: Rien à faire pour "ex1".
```

Séparation des dépendances aux `.h`

```
ex1 : principal.o point.o max.o
    gcc principal.o point.o max.o -o ex1

principal.o : principal.c
    gcc -Wall -W -std=c99 -c principal.c
point.o : point.c
    gcc -Wall -W -std=c99 -c point.c
max.o : max.c
    gcc -Wall -W -std=c99 -c max.c

principal.o : point.h max.h
point.o : point.h
max.o : max.h
```

Intérêt : générer automatiquement la 3$^{\text{ème}}$ partie avec la commande `makedepend`

Commande `makedepend`

(Ubuntu : paquet `xutils-dev`)

Il suffit de remplacer la 3ème partie par

```
depend :
    makedepend principal.c point.c max.c
```

Usage : `make depend`

- Exécute `makedepend principal.c point.c max.c`
- Rajoute à la fin du `Makefile` : `"# DO NOT DELETE"` suivi de la 3ème partie générée par `makedepend`

Si on relance, `makedepend` cherchera le marqueur `"# DO NOT DELETE"` et écrasera la fin.

Exemple avec makedepend

Fin du Makefile :

```
depend :
    makedepend -- -std=c99 -- principal.c max.c point.c \
    2> /dev/null

# DO NOT DELETE

principal.o: /usr/include/stdio.h /usr/include/features.h
principal.o: /usr/include/stdc-predef.h /usr/include/libio.h
principal.o: /usr/include/_G_config.h /usr/include/wchar.h \
    max.h point.h
max.o: max.h
point.o: /usr/include/stdio.h /usr/include/features.h
point.o: /usr/include/stdc-predef.h /usr/include/libio.h
point.o: /usr/include/_G_config.h /usr/include/wchar.h point.h
```

Autre méthode avec gcc -MM

Fin du Makefile :

```
depend :
    gcc -MM -std=c99 point.c max.c principal.c >| depend.txt

-include depend.txt
```

Fichier généré :

```
$ make depend
gcc -MM -std=c99 point.c max.c principal.c >| depend.txt

$ cat depend.txt
point.o: point.c point.h
max.o: max.c max.h
principal.o: principal.c max.h point.h
```