Techniques et outils d'ingénierie logicielle

Cours 2 — compilation

John Gliksberg — Bull

Objet du cours

- Maîtriser un *shell* et ses automatisations
- Avoir un environnement de développement productif
- Savoir compiler un programme
- Appréhender le cycle de vie d'un programme
- Gestion de code source
- Déboguer un programme
- Profiler un programme

Contents

1. Règles du jeu
2. Les outils de l'ingénieur·e informaticien·ne
3. Le système d'exploitation
4. Anatomie d'un *OS Tree* Linux
5. « norme de la hiérarchie des systèmes de fichiers (FHS) »
6. Le shell et son invite de commande
7. À l'aide
8. Éditer la ligne de commande (readline)

9. Compiler un programme
10. Makefile

Règles du jeu

Matière **pratique**
TP **notés**

Matière **pratique**
PC **à chaque séance**

Environnement type **UNIX**
(Linux, BSD, Max, WSL)

Accès facile **à tout moment** au terminal

Participez
mais lisez les manuels

Pas de magie, explications **spécifiques**

Testez ce que vous dites

Testez ce que vous pensez

Montrez que vous avez compris

« Pas le temps de corriger ma config » **interdit**

La peur n’évite pas le danger

La peur est la petite mort

Les outils de l'ingénieur·e informaticien·ne

Le terminal, l'éditeur de texte, le VCS
le compilateur, le débogueur, le profileur

L’invite de commande, les commandes, les raccourcis

Le système d’exploitation, l’environnement de bureau

Le PC, le clavier

Les mains, la dextérité

La tête, le sens critique, la curiosité, la parole
la lecture, la mémoire, la capacité d'abstraction

Les manuels
(et StackOverflow, Wikipédia, LLM)

Le système d'exploitation

- Un noyau, des drivers, des systèmes de fichier
- un environnement graphique
  - KDE, Gnome, Sway, XFCE
- des programmes inclus
- un gestionnaire de paquets
- un ecosystème
- des choix, par des gens

- C’est votre maison, prenez-en soin
- Modifiez le, testez en plusieurs
- Soyez prêt·e à perdre tous les fichiers

Il existe différents desktop environments (DEs)

- **KDE** : Un environnement de bureau complet et polyvalent, très configurable
- **Gnome** : Un environnement fait pour être intuitif, le choix par défaut de nombreuses distributions, mais peut être plus complexe à configurer que KDE ou XFCE
- **XFCE** : Léger et facile à comprendre mais moins puissant que KDE/Gnome

Considérez les tiling window-managers (TWM)

- **i3** : Un gestionnaire dynamique de fenêtres qui est léger et configurable, mais peut être difficile à prendre en main au premier abord
- **Sway** : Comme i3, sur Wayland
- **awesomewm**, **bspwm**, **dwm**, **Hyprland**
- **PaperWM** : un scrolling TWM pour Gnome
- **FancyZones** : pour Windows
- **yabai** : pour MacOS

Tous les fichiers sont dans un grand arbre de dossiers

qui commence à la racine « / »

Il y a des chemins absolus `/home/`*`john`*`/.config/nvim/init.lua`

et des chemins relatifs `Téléchargements/meme.gif`

`../../img/../src`

`./.zshrc`

Il peut y avoir ~ n'importe quel caractère

Les *dotfiles* sont « cachés »

Anatomie d'un *OS Tree* Linux

```
$ tree -L 1 /
/
├── bin
├── boot
├── dev
├── etc
├── home
├── init
├── lib
├── lib64
├── lost+found
├── mnt
├── opt
├── proc
├── root
├── run
├── sbin
├── srv
├── sys
├── tmp
├── usr
└── var
```

Voici les répertoires à la racine « / » d'un Linux

(Dépend de la distribution !)

Métaphore : tout est un fichier

```
$ ls /dev
autofs           loop0         ram14     stdout  tty32  tty58        vcsa
block            loop1         ram15     tty     tty33  tty59        vcsa1
bsg              loop2         ram2      tty0    tty34  tty6         vcsa2
btrfs-control    loop3         ram3      tty1    tty35  tty60        vcsa3
char             loop4         ram4      tty10   tty36  tty61        vcsa4
console          loop5         ram5      tty11   tty37  tty62        vcsa5
core             loop6         ram6      tty12   tty38  tty63        vcsa6
cpu_dma_latency  loop7         ram7      tty13   tty39  tty7         vcsu
cuse             loop-control  ram8      tty14   tty4   tty8         vcsu1
disk             mapper        ram9      tty15   tty40  tty9         vcsu2
dxg              mcelog        random    tty16   tty41  ttyS0        vcsu3
fd               mem           rtc       tty17   tty42  ttyS1        vcsu4
full             mptctl        rtc0      tty18   tty43  ttyS2        vcsu5
fuse             mqueue        sda       tty19   tty44  ttyS3        vcsu6
hpet             net           sdb       tty2    tty45  ttyS4        vfio
hugepages        null          sdc       tty20   tty46  ttyS5        vga_arbiter
hvc0             nvram         sdd       tty21   tty47  ttyS6        vhost-net
hvc1             ppp           sde       tty22   tty48  ttyS7        vhost-vsock
hvc2             ptmx          sg0       tty23   tty49  uinput       virtio-ports
hvc3             ptp0          sg1       tty24   tty5   urandom      vport0p0
hvc4             ptp_hyperv    sg2       tty25   tty50  userfaultfd  vport0p1
hvc5             pts           sg3       tty26   tty51  vcs          vport0p2
hvc6             ram0          sg4       tty27   tty52  vcs1         vsock
hvc7             ram1          shm       tty28   tty53  vcs2         zero
hwrng            ram10         snapshot  tty29   tty54  vcs3
kmsg             ram11         snd       tty3    tty55  vcs4
kvm              ram12         stderr    tty30   tty56  vcs5
log              ram13         stdin     tty31   tty57  vcs6
```

« NORME DE LA HIÉRARCHIE DES SYSTÈMES DE FICHIERS (FHS) »

/bin — Binaires de base

/boot — Image du noyau (kernel), images de boot, configuration du bootloader

/dev — Appareils (devices) matérialisés sous forme de fichiers

/etc — Configurations

/home — Répertoires utilisateurs

/lib — Bibliothèques partagées principales (libc)

/mnt — Points de « montage » temporaire (par exemple une clef USB)

/opt — Logiciels commerciaux

/proc — Informations sur les processus et l'état de la machine

/root — « home » du super-utilisateur

/sbin — Binaires destinés au super-utilisateur uniquement

/sys — Configuration du kernel et matérielle (également via des fichiers)

/tmp — Répertoire temporaire généralement attaché à un ramfs

/usr — Majorité des programmes (a.k.a. « préfixe »)

/var — Données modifiées par les programmes : (/var/log, /var/mail, ...)

Le shell et son invite de commande

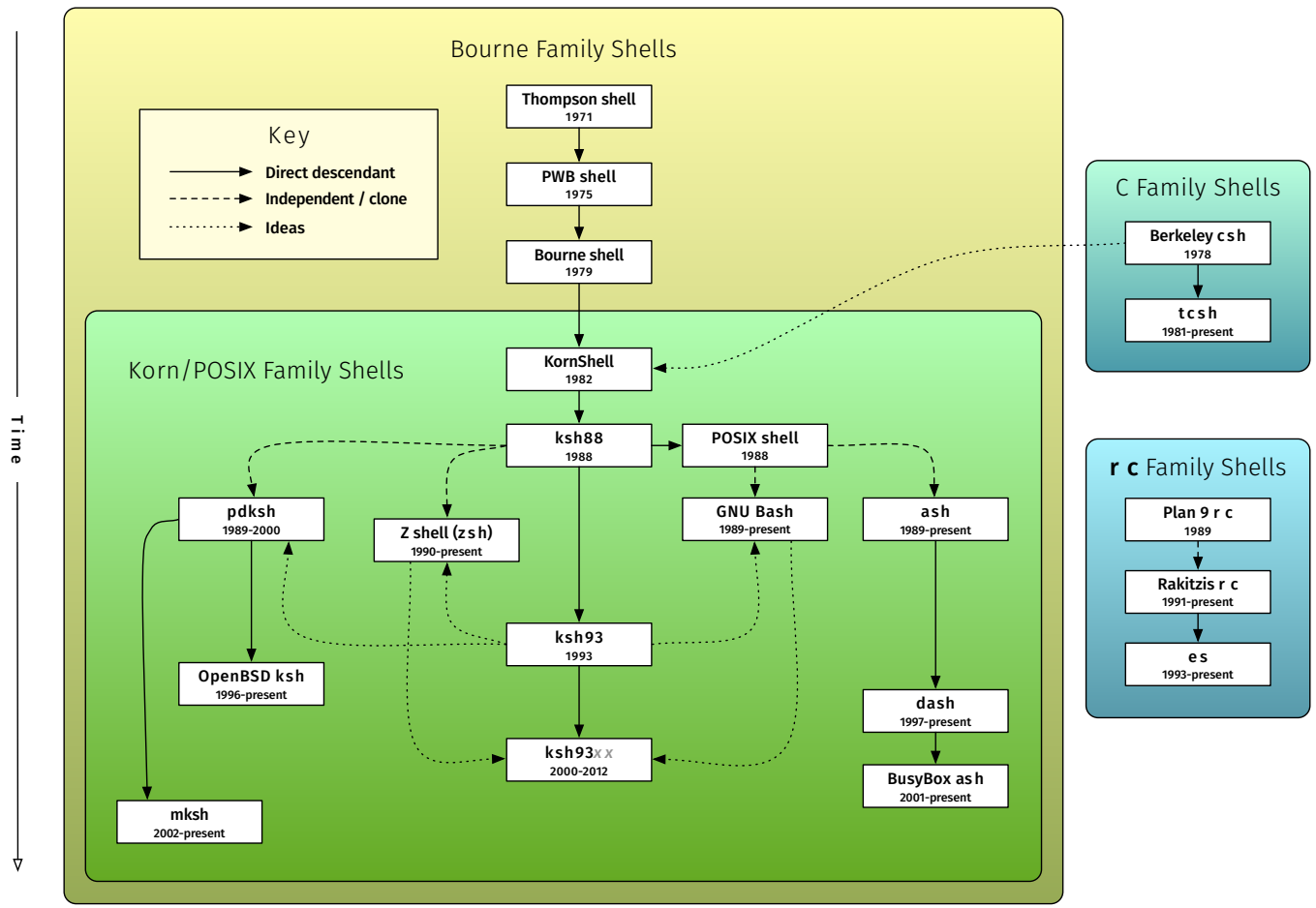
Le shell sert à :

- Lancer des programmes
- Leur passer des arguments
- Entrer des données sur leur entrée standard (*stdin*)
- Afficher leur sorties standard (*stdout*) et d'erreur (*stderr*)

Le shell est utilisé dans un pseudo-terminal qui est capable de gérer des séquences spéciales de caractères pour :

- L'affichage (échappement curseur et couleur)
- La gestion des processus (^C, ^Z)

Choix du shell



Compatibilité POSIX

Bash très utilisé

- bashismes dans le shell OK
- dans les scripts ⚠

Voir aussi

- Tcl
- Fish
- PowerShell
- NuShell
- Xonsh

Il vous invite (*prompt*) à écrire des commandes

```
john@superbecane:~$
```

```
~          /home/john
~john      /home/john
~elsa      /home/elsa
```

Les commandes sont une list de chaînes de caractères

Les commandes sont structurées selon des conventions

```
john@superbecane:~$ find src -name '*.h'
```

Le shell permet de construire ces chaînes de caractères automatiquement

```
$ which ls
/usr/sbin/ls

$ ls -l "$(which ls)"
-rwxr-xr-x 1 root root 158480 Sep 25 13:39 /usr/sbin/ls

$ test "Mon document" = "Mon document" && echo ok || echo ko
ok
$ document="Mon document"
$ test $document = "Mon document" && echo ok || echo ko
```

→ ?

Les commandes peuvent êtres compilées

```
$ file `which ls`
/usr/sbin/ls: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=c4559d6cae6c36bc3cc7b1f97fe3379ada0b52a2, for GNU/Linux 4.4.0,
stripped
```

Ou bien interprétées (shebang)

```
$ file `which ex`
/usr/sbin/ex: POSIX shell script, ASCII text executable

$ cat `which ex`
#!/bin/sh
exec nvim -e "$@"
```

Les commandes peuvent êtres enchaînées (pipeline)

```
file /bin/* | grep -v -i elf | grep -v symbolic
```

```
for $arg
do
  ... "$arg"
done
```

```
while test "$#" -gt 0
do
  ... "$1"
  shift
done
```

```
case $foo in
  -h ) ... ;;
  *document | *papier )
    ...
    ;;
  * )
    return 1
    ;;
esac
```

À L’AIDE

Le manuel

```
$ man [<section>] <page>
```

```
man ls
man pwd
man man
man man-pages
man firefox
man 1 ls
man 7 signal
```

See also **ls**(**1**), **signal**(**7**).

Le manuel s'ouvre avec **less**(**1**) par défaut

- Flèches, PageDown/PageUp, `d`/`u`, `g`/`G`
- `q`
- `h`
- `/ -d`, `n`/`N`

Ou bien … `MANPAGER=mupdf man -Tpdf less`

```
cat -h
cat --help
```

CTRL + C → SIGINT

CTRL + Z → SIGTSTP

- jobs(1)
- bg(1)
- fg(1)

```
$ jobs
[1]+  Stopped                 vim slides.typ
[2]-  Stopped                 xeyes

$ bg %2
[2]- xeyes &

$ jobs
[1]+  Stopped                 vim slides.typ
[2]-  Running                 xeyes &
```

CTRL + \ → SIGQUIT (⚠)
Voir signal(7)

CTRL + S/Q → TTY XOFF/XON

Éditer la ligne de commande (readline)

CTRL + A/E — Curseur au début/fin de ligne

CTRL + W — Couper le dernier mot avant curseur

CTRL + U/K — Couper du curseur au début/fin de ligne

CTRL + Y — Coller

CTRL + L — clear l’écran

CTRL + R — Chercher un commande dans l'historique

ALT + . — Reprendre le dernier argument

La variable PS1

Permet de configurer l'apparence du prompt

A une syntaxe souvent spécifique au shell pour les placeholders

Peut prendre de la couleur

Générateurs sur le web (« PS1 generator <shell> »)

Bash :

```
export PS1="\u@\h:\w$ "
```

```
[john]@[superbecane]:[/etc] $
```

Zsh :

```
export PS1="%n@%m %~ $ "
```

Les alias


```
alias ll="ls -lh"
alias gs="git status"
alias dc="docker-compose"
```

La configuration du shell

Les shells ont tous un fichier de configuration dans lequel mettre les commandes à lancer au démarrage

- Bash: ~/.bashrc
- Zsh: ~/.zshrc
- ...

Rendre persistante la configuration du shell entre les démarages

C'est un simple fichier lancé au début de chaque session de shell

Pour prendre en compte les changements en direct :

```
$ . ~/.<shell>rc
```

Compiler un programme

Langages compilés vs langages interprétés

Langages interprétés

- BASIC, Shell, Python, Ruby, Scilab, ...
- Exécutés avec le code sans traduction
- Souvent plus faciles à apprendre

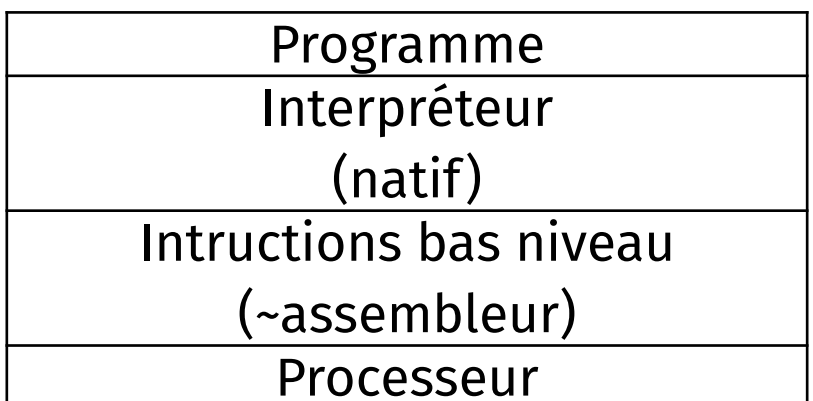
Langage compilés

- Fortran, C, C++, Rust, Zig, ...
- On exécute un binaire compilé
- Souvent plus performants

Contre-exemples

- Compilation intermédiaire (bytecode, JIT)
- Cython
- Go run, TCC

Langage interprété



Langage compilé

Programme	Compilateur	Binaire (natif)
		Intructions bas niveau (~assembleur)
		Processeur

Il existe de nombreux langages et compilateurs mais il y en a deux principaux, C et C++, GCC et LLVM.

C :

- **gcc** (GNU Compiler Collection)
- **clang** (LLVM)

C++ :

- **g++** (GNU Compiler Collection)
- **clang++** (LLVM)

GCC (GNU Compiler Collection) et LLVM (Low Level Virtual Machine) sont deux ensembles de compilateurs largement utilisés dans le développement logiciel :

GCC :

- Développé par le projet GNU
- Supporte de nombreux langages comme C, C++, Objective-C, etc
- Robuste, stable et compatible avec de nombreuses plateformes
- Distribué sous licence libre (GPL)

LLVM :

- Ensemble de compilateurs modulaires et infrastructure de développement
- Conçu pour être modulaire et flexible
- Base pour créer des compilateurs pour différents langages (par exemple Rust)
- Architecture orientée performance et optimisation du code
- Distribué sous licence libre (UIUC)

	Source C
`gcc -E m.c`	Source préprocessées
`gcc -S m.c`	Code assembleur
`gcc m.c`	Exécutable (*Elf* sous Linux)

utils.h

```h
#ifndef UTILS_H
#define UTILS_H

int fibonacci(int n);

#endif // UTILS_H
```

utils.c

```c
#include “utils.h”

int fibonacci(int n)
{
    if (n <= 1)
        return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

main.c

```c
#include <stdio.h>

#include “utils.h”

int main(void)
{
    int n = 10; // Exemple : Calculer le 10ème terme de la suite de Fibonacci
    int result = fibonacci(n);
    printf("Le %dème terme de la suite de Fibonacci est : %d\n", n, result);
    return 0;
}
```

Compilation en un binaire

On Compile

```
gcc ./main.c ./utils.c -o fibo
    [~~~~Fichiers~~~~]    [binaire]
```

On Lance !

```
./fibo
```

Notion de bibliothèque dynamique

Une bibliothèque dynamique :

- aussi connue sous le nom de DLL (Dynamic Link Library) sous Windows
- ou de fichier .so (Shared Object) sous Unix/Linux
- est un ensemble de fonctions et de routines précompilées

Contrairement aux bibliothèques statiques, les bibliothèques dynamiques sont chargées en mémoire au moment de l'exécution du programme.

Avantages des bibliothèques dynamiques

Economie d'espace :

- Elles permettent de partager le code entre plusieurs programmes
- réduisant ainsi la taille totale des programmes

Mises à jour facilitées :

- Les mises à jour de la bibliothèque peuvent être effectuées indépendamment des programmes qui l'utilisent

Chargement à la demande :

- Les bibliothèques dynamiques ne sont chargées en mémoire que lorsque nécessaire
- ce qui peut réduire le temps de démarrage
- et économiser de la mémoire.

Faiblesses des bibliothèques dynamiques

Résolution d'une partie du code au runtime (incompatibilité, instabilité)

Nécessité de maintenir les search path :

- /usr/lib
- `$LD_LIBRARY_PATH` (environnement plus complexe)

Surcoût lors du premier appel : passage par la Procedure Linkage Table (PLT)

Notion de bibliothèque dynamique

Compilation : `utils.h` `main.c` → `libutils.so`

Exécution : `main` `libutils.so`

Compilation en un binaire

On Compile

```
$ gcc ./main.c ./utils.c -o fibo
      [~~~~Fichiers~~~~]    [binaire]
```

On Lance !

```
$ ./fibo
```

On regarde les bibliothèques de notre fibo (nous sommes déjà link à des libs) :

```
$ ldd fibo
  linux-vdso.so.1 (0x00007ffcbd620000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa0090c6000)
  /lib64/ld-linux-x86-64.so.2 (0x00007fa0092db000)
```

Construisons libutils.so

On Compile

```
gcc -shared -fPIC utils.c -o libutils.so
```

- `-shared` :

  Produire une bibliothèque partagée (.so) plutôt qu'un exécutable

- `-fPIC` :

  Position Independent Code

  (essentiel pour permettre le partage des bibliothèques entre plusieurs processus en mémoire)

Contriuisons fibo en nous linkant à libutils.so

On Compile

```
gcc main.c -o main -L. -lutils
```

- `-L.` : Rechercher les bibliothèques partagées dans le répertoire courant « . ».
- `-lutils` : Lier le programme avec une bibliothèque appelée libutils.so. (Le préfixe « lib » et le suffixe « .so » sont automatiquement ajoutés.)

Tentons de lancer


```
./main
./main: error while loading shared libraries: libutils.so: cannot open shared
object file: No such file or directory
```

Le binaire (au runtime ne trouve pas libutils.so)

```
ldd main
        linux-vdso.so.1 (0x00007ffc101fb000)
        libutils.so => not found
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd347b9d000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fd347db2000)
```

Au démarrage du programme, le loader doit charger les bibliothèques :

- **rpath** : chemin fourni lors de la compilation.
- **$LD_LIBRARY_PATH**: chemins de recherche dans une variable d'environnement
- **Fichiers de configuration ld** :

```
/usr/local/cuda/targets/x86_64-linux/lib
/usr/local/cuda-12/targets/x86_64-linux/lib
/usr/local/cuda-11.0/targets/x86_64-linux/lib
...
```

- **Chemins standards** : Si la bibliothèque n'est pas trouvée dans les étapes précédentes, le système recherche dans les chemins standard, tels que :

```
/lib
/lib64
/usr/lib
/usr/lib64
```

Tenton de lancer, avec LD_LIBRARY_PATH


```
$ LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH ./main
Le 10ème terme de la suite de Fibonacci est : 55

$ LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH ldd main
  linux-vdso.so.1 (0x00007ffd33ffe000)
  libutils.so => /home/john/src/toi2025/fibonaci/libutils.so
(0x00007fe451578000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fe45136a000)
  /lib64/ld-linux-x86-64.so.2 (0x00007fe451584000)
```

Binaire statique (tout à l'intérieur)

On compile

```
gcc ./main.c ./utils.c -o fibo -static
```

On lance !

```
./fibo
```

Fibo est maintenant un binaire statique :

```
$ ldd fibo
n'est pas un exécutable dynamique
```

Makefile

Fichier de configuration de l'outil **make**(**1**)

Facilite la compilation et le link de programmes

Le nom du fichier est toujours « Makefile » (sauf explicite)

On compile avec la commande `make` dans le répertoire où se trouve le Makefile

Reproductibilité de compilation (il est facile d'oublier un flag de compilation)

Principaux arguments à la commande make :

- `-f myMakefile` : changer le nom du fichier
- `-C <chemin_du_projet>` : pour compiler dans un autre répertoire
- `<cible>` : cible de communication

Makefile

- Une cible définit un fichier à construire ou une action
- Une règle est l'ensemble des commandes à exécuter pour réaliser cette cible (lancées dans un shell différent)
- Chaque règle commence par une tabulation
- La première cible est celle exécutée par défaut
- Une cible peut avoir des dépendances : des cibles à résoudre avant
- La résolution de dépendances fonctionne par horodatage

  Permet par exemple de recompiler les .o pour lesquels le fichier .c a été modifié en amont

`$@`	Nom de la cible
`$<`	Nom de la 1ère dépendance
`$^`	Nom de toutes les dépendances

```
main.bin:
  gcc -o main.bin main.c
```

```
main.o: main.c
  gcc -c main.c -o main.o

main.bin: main.o
  gcc -o main.bin main.o
```

```
main.o: main.c main.h
  gcc -c -o $@ $<

main.bin: main.o
  gcc -o $@ $<
```

```
make main.bin
```