Programmation C avancée

Concepts & Outils
pour le développement

maj 01/2023

gestion de sources

- Un gestionnaire de sources permet de conserver l'historique des modifications apportées à un ensemble de fichiers.
- Il permet à un ensemble d'utilisateurs d’interagir sur un même code source.
- Tous les gestionnaires de code sources sont basés sur les principes de diff et patch
- Il existe deux grandes catégories de gestionnaires :
  - les centralisés
  - les décentralisés

Les gestionnaires centralisés

- Historiquement, cvs (concurent versioning system) et son successeur svn (subversion)
- Les gestionnaires centralisés reposent sur un serveur central qui archive toutes les modifications apportées au code.
- Chaque modification incrémente un numéro de révision
- Les commandes de base de svn :
  - checkout, update, infos, status, commit, diff, revert

svn

- Chaque utilisateur interagit avec le serveur, il n'y a pas d'échange direct entre deux utilisateurs.
- Une méthodologie est associée à l'utilisation de svn, vous retrouverez cet méthodologie dans tout projet de développement (open source ou entreprise) => « svn red book »

svn : méthodologie

- Il est possible d'utiliser SVN juste pour le répertoire de développement principal.
- Seulement, comment faire si on a un « gros » développement à produire: si on transmet les modifications intermédiaires, le programme devient instable (ne compile plus par exemple...)
- Comment faire également pour gérer les versions:
  - On sort une version 1.0 qui évolue en 1.1 puis 1.2
  - On sort la version 2.0 (« casse » la compatibilité avec 1.x)
  - Un bug est trouvé dans la version 2.0: il faut le corriger dans la version courante mais également dans la version 1.x!

Un mot sur les versions...

- Il n'y a pas de « règles » universelles mais un certain nombre de pratiques.
- L'approche par numérotation est la plus répendue :

2.10.5

Major.Minor.Patch

Version.SousVersion.Patch

- Dans le cas d'une bibliothèque, on peut suivre les principes suivant :
  - Tant que l'API i.e. les fichiers d'entête et les fonctions (noms/signatures) ne changent pas, on garde le même numéro de version
  - On peut ajouter des fonctionalités : on incrémente le numéro de sous version
  - On peut corriger des bugs : on incrémente le numéro de patch

→ Un changement de version (Major) implique (sauf exception) une non compatibilité

Un mot sur les versions….

- Il existe d'autres stratégies de nommage/numérotation des versions.
- Par exemple, sur la base d'une date :
  - Ubuntu 18.04 => 2018 / Avril
- Parfois un mix :
  - Version.SousVersion.date(YYYYMMDD)
- On veille cependant à ce qu'il y ai un logique d'ordre
- Il est aussi apprécié que la logique de numérotation suit un ordre (lexicographique et/ou numérique).

svn : méthodologie

- La gestion des versions implique de maintenir plusieurs « répertoires » de développement en parallèle.
- Une méthodologie classique organise les sources ainsi:
  - / « racine »
  - /trunk : contient la version en cours des sources (dev.)
  - /branches/: des copies de trunk (« svn copy »)
  - /branches/1.0 : copie de trunk pour release (tests), retour de modif dans le trunk avec « svn merge » si compatible
  - /tags/1.0.0: version **figée** d'une branche qui est publiée, sert de référence. La branche correspondante est étiquetée (tag). **pas de commit/modifs dans ce répertoire**
  - /branches/modif_allali_155/: copie temporaire pour de « grosses » modifications avec retour dans le trunk par « merge ». Synchronisation régulière depuis le trunk (« merge »)

svn : la création d’un dépôt

- Le création d'un dépôt se fait à l'aide de la commande « svnadmin create nom_de_depot »
- possibilité d'ajouter l’exécution de script avant/pendant et après les « commits »
- possibilité d’envois de mails lors des commits
- facile à mettre en place sur son compte:
- cd ~/.depots/ ; svnadmin create SVN
- cd ~/; svn co file:///$HOME/.depots/SVN
- via ssh:
- svn co svn+ssh://allali@ssh.enseirb.fr/.depots/SVN

contrôle dans svn

- Il est possible d'avoir une approche hiérarchique dans svn en subdivisant le répertoire « branches » en répertoires et en ajustant les droits d’accès (lecture/ecriture) :
  - seuls les masters peuvent commiter dans « trunk »
  - les dev travaillent dans des branches
- Cela permet de faire de la revue de code

Les gestionnaires dé-centralisés

- Dans ce cas, il n'y a pas de dépôt central.
- Chaque utilisateur gère son propre dépôt.
- Un protocole permet l'échange de modifications (commit) entre deux utilisateurs.
- Exemple : git
- Commandes de base :
  - clone, add, commit, push, pull, fetch, merge, branch, checkout

git : les commits, pull et push

- Dans git, les commits sont locaux (il n'y a pas de dépôt central).
- On peut transmettre un ensemble (suite) de *commit* à un autre «dépôt» avec la commande *push*
- On peut réceptionner un ensemble de *commit* depuis un autre dépôt avec la commande *pull*.
- On peut ajouter/supprimer autant de dépôts connectés à notre dépôt avec « remote ». A faire dans les deux dépôts si l’on souhaite une symétrie.

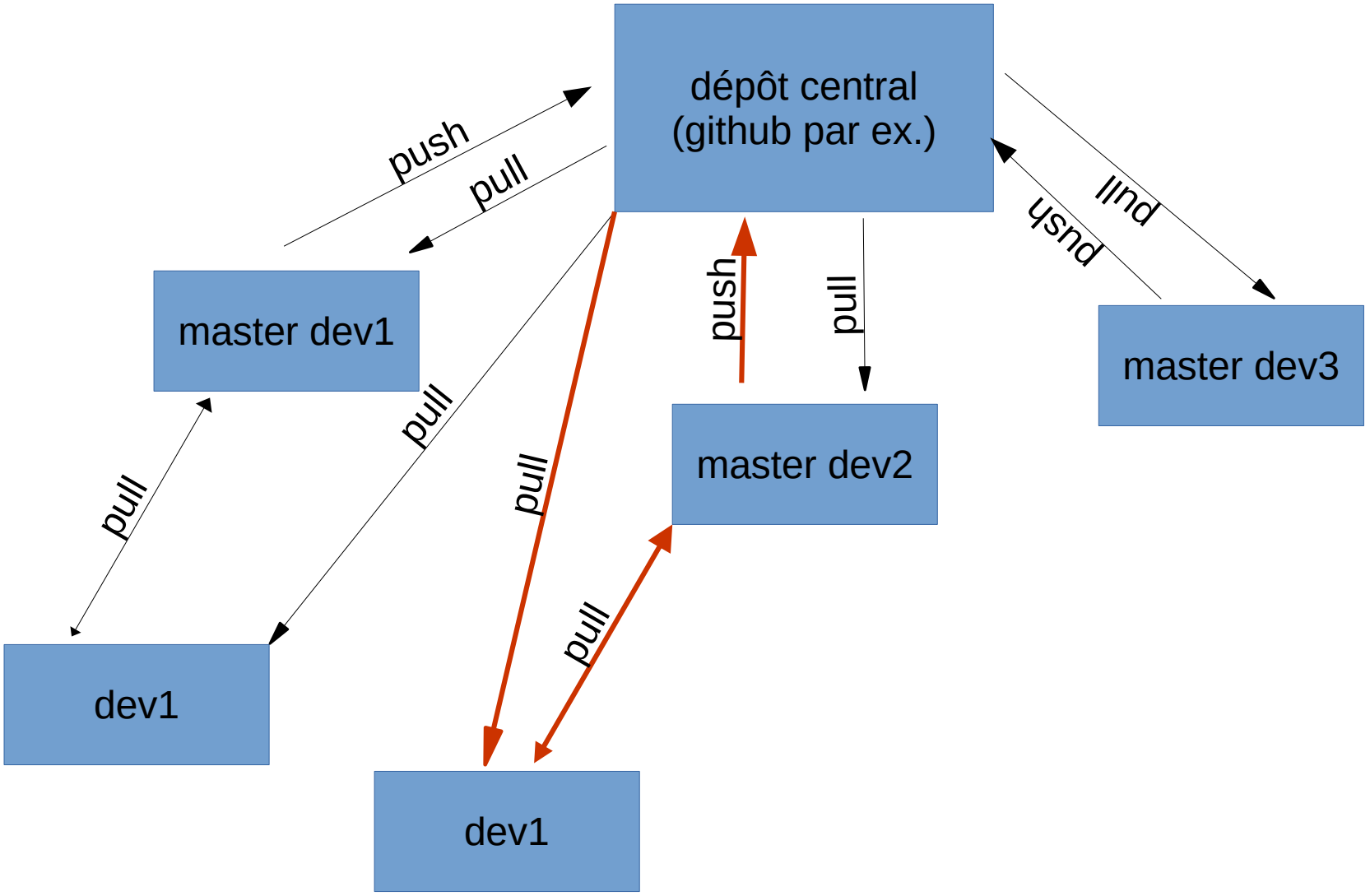
git

- git intègre une gestion native des branches :
  - création :
    - git branch b2
    - git checkout b2
    - ou bien git checkout -b b2
  - la fusion :
    - on bascule dans la branche qui doit recevoir les modifs
    - git checkout master ; git merge b2
    - Les modifications doivent avoir été *commitées* dans b2
  - la délétion : git branch -d b2
- La branche par défaut s'appelle master (paramètrable lors de la création d'un dépôt).

re-centralisation

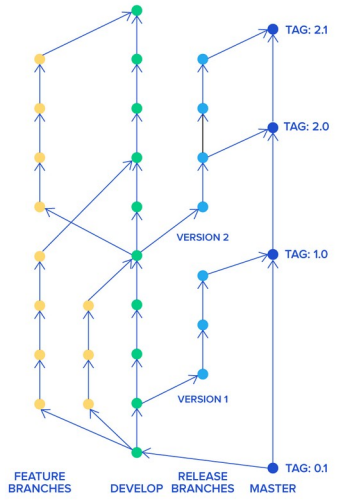
- L'avantage d'être en décentralisé et de pouvoir faire des « commit » sans connexion à un serveur.
- Pour la plupart des projets, il est cependant nécessaire d'avoir une référence : on utilise alors un dépôt git comme tel (github par exemple). Un tel dépôt s'appelle un « bare », il a la spécifité d'accepter les *push*
- On peut ensuite mettre en place un système de propagation hiérarchique des « commits ».

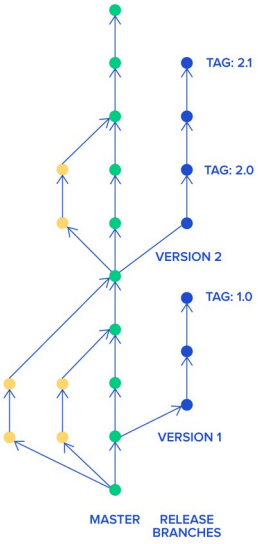
git


dépôt central
(github par ex.)
push
pull
master dev1
pull
pull
dev1
push
pull
master dev2
pull
dev1
llnd
ysnd
master dev3

git: méthodologie

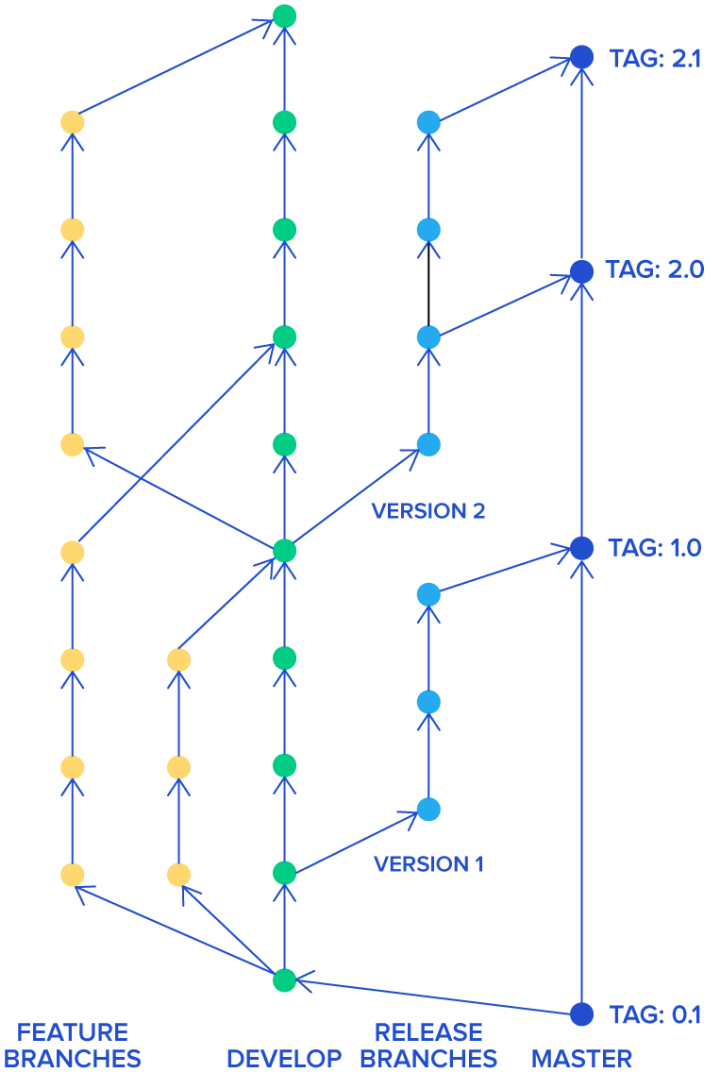
- L'utilisation de git repose nativement sur les branches
- Il y a deux approches actuellement:
  - git flow
  - trunk based





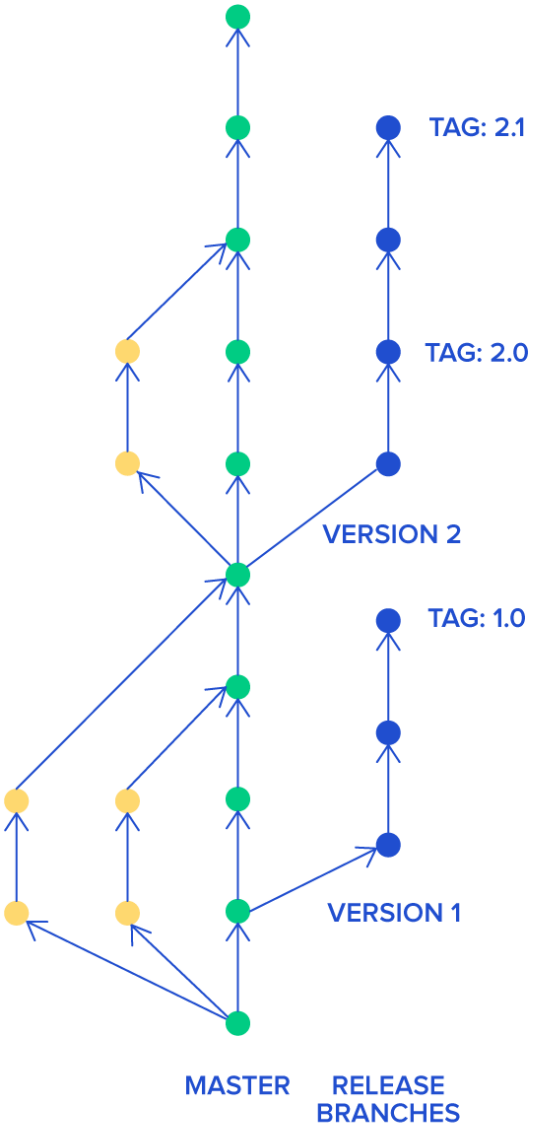
GitFlow

- *GitFlow* suppose des branches avec une durée de vie assez longue: il y a des aller-retour entre ces branches et une branches de développement (et les autres branches), elle même en synchronisation avec une branche de pré-release et la branche master de qui contient le code stable



Trunk Based

- *trunk based* privilégie les branches à durée de vie courte synchronisée avec master/trunk
- les release sont maintenues à coté.



GitFlow ou trunk based

- Les deux méthodologies ont leurs avantages et inconvénients
- L'environnement, le type de projet, le nombre de développeurs sont autant de facteurs qui plaident pour l'un ou l'autre des solutions

git ou svn?

- Différences majeures entre svn et git est :
  - centralisé / dé-centralisé
  - support natif du système de branches dans git
  - commit locaux dans git
- aujourd'hui git est de loin de le gestionnaire de sources le plus utilisé

Les autres gestionnaires

	Open Source	Centralisé	Décentralisé
CVS	•	•	
SVN	•	•	
GIT	•		•
SourceSafe		•	
Mercurial	•		•
Bazaar	•		•
BitKeeper			•
Team Foundation Server		•	

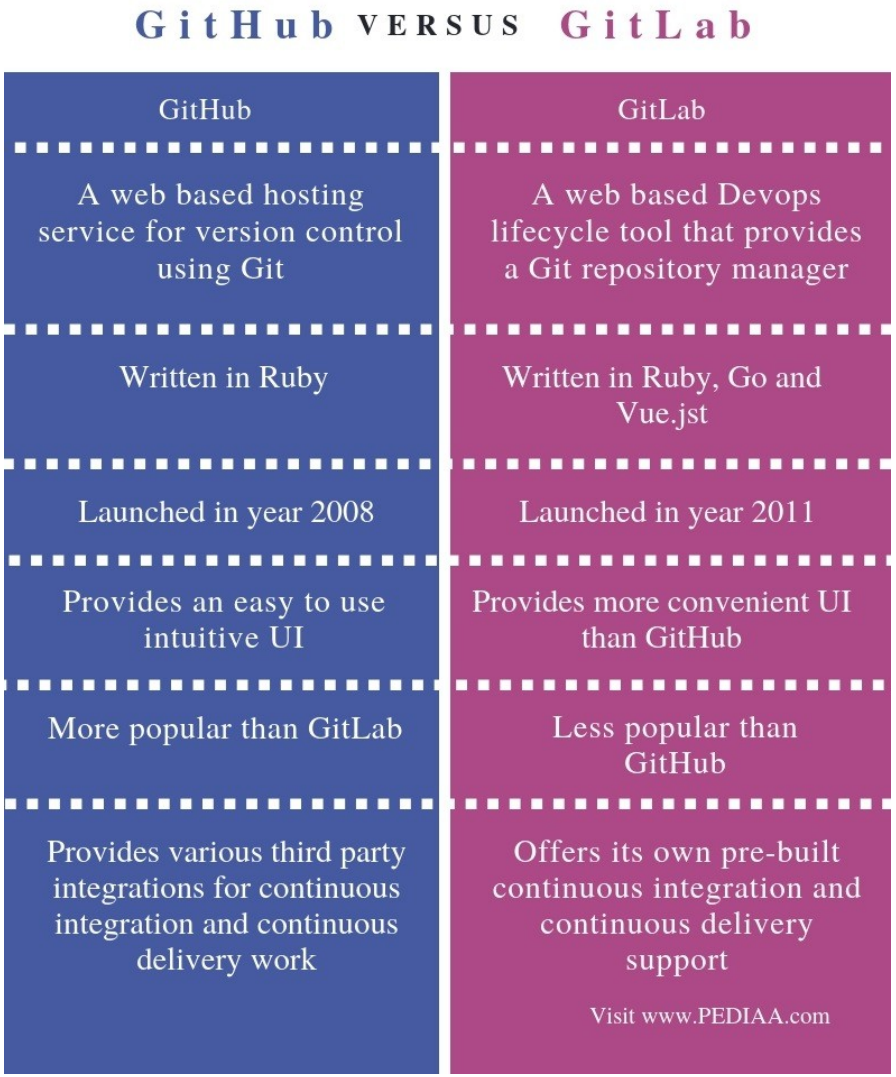
Il en existe bien d’autres (voir : https://en.wikipedia.org/wiki/List_of_version-control_software)

les plateformes

- De nombreuses plateformes proposent l'hébergement de dépôts telles que github et gitlab

  mais aussi bitbucket...

- ajout de services tels que l'intégration continue, suivi de bugs, wiki etc....



The compiling trivia

#QDLE#Q#ABCD*E#25#

- Quelle commande permet de compiler le fichier toto.c en objet ?
  - A. gcc toto.c
  - B. gcc -E toto.c
  - C. gcc -S toto.c
  - D. gcc -c toto.c
  - E. gcc -fPIC toto.c

The compiling trivia

#QDLE#Q#AB*CD#25#

- Quelle commande permet de récupérer le source après l'étape de pré-compilation ?
  - A. gcc toto.c
  - B. gcc -E toto.c
  - C. gcc -S toto.c
  - D. gcc -c toto.c

The compiling trivia

#QDLE#Q#A*BCD#25#

- Quelle commande permet de lier toto.o et main.o en un exécutable ?
  - A. gcc toto.o main.o
  - B. gcc -shared toto.o main.o
  - C. gcc -static toto.o main.o
  - D. gcc -exec toto.o main.o

The compiling trivia

#QDLE#Q#ABCD*#25#

- Quelle commande permet de lier toto.o et main.o en une bibliothèque statique?
  - A. gcc -shared -o libfoo.a toto.o main.o
  - B. gcc -static -o libfoo.a toto.o main.o
  - C. gcc toto.o main.o -lfoo
  - D. ar rcs libfoo.a toto.o main.o

The compiling trivia

#QDLE#Q#ABCDEF*G#30#

- Pour pouvoir regrouper des objets (.o) dans une bibliothèque dynamique, il faut les compiler avec la ou les options :
  - A. -c
  - B. -E
  - C. -s
  - D. -fPIC
  - E. réponse A&B
  - F. réponse A&D
  - G. réponse B&E

The compiling trivia

#QDLE#Q#A*BCD#25#

- Quelle commande permet de lier toto.o et main.o en une bibliothèque dynamique?
  - A. gcc -shared -o libfoo.so toto.o main.o
  - B. gcc -dynamic -o libfoo.a toto.o main.o
  - C. gcc toto.o main.o -lfoo
  - D. ar rcsD libfoo.so toto.o main.o

Automatisation

- La compilation manuelle n'est pas possible sur un grand projet :
  - homogénéisation des options de compilation
  - gestion des dépendances (que doit-on recompiler?)
  - commandes et options spécifiques à une plate-forme.
  - erreurs manuelles
  - ...
- On doit se munir d'outils pour automatiser cette étape.

Makefile

- make est un outil qui permet d'automatiser la création et la mise à jour de fichiers.
- make repose sur un fichier de description : Makefile
- Un Makefile est un ensemble de règles formatées :

  cible : source1 source2 source3 ...

  ```
  commande1
  commande2
  ... .
  ```

- Si « cible » n'existe pas où est moins récente que l'une des sources, alors les commandes sont exécutées.

Makefile : exemple

```
image_thumbail.jpg : image.jpg
    convert -size 80x80 image.jpg image_thumbnail.jpg
image.jpg : image.png
    convert image.png image.jpg
image.png : image.svg
    inkscape -z -e image.png image.svg
```

- make est récursif : si un fichier source n'existe pas, il cherche une règle pour le créer.
- initialement, make cherche à créer une seul cible : la première du Makefile ou celle(s) indiquée(s) sur la ligne d'appel : make image.png
- Si une source est manquante et qu'il n'y a pas de règle pour la créer : erreur
- Si une des commandes renvoie une valeur différente de 0, make s'arrête (tips : `commande || true`)

Makefile : variable

- make supporte la déclaration de variable :

  `NOM=VALEUR`

- valeur peut comporter des espaces

  `NOM-=VALEUR`

- Si NOM est dans l'environnement, alors c'est la valeur de l'environnement qui est utilisée, sinon c'est VALEUR.
- $(NOM) est remplacé par VALEUR.

Makefile : variable, exemple

```
CONVERT=convert
THUMB_SIZE-=80x80
image_thumbail.jpg : image.jpg
    $(CONVERT) -size $(THUMB_SIZE) image.jpg image_thumbnail.jpg
image.jpg : image.png
    $(CONVERT) image.png image.jpg
image.png : image.svg
    inkscape -z -e image.png image.svg
```

$> THUMB_SIZE=60x60 make

Makefile : spéciales

- Quelques variables spéciales pour l'écriture des commandes :

```
cible : source1 source2
```

  - $@ : cible
  - $< : source1
  - $^ : source1 source2

- Ainsi :

```
image_thumbail.jpg : image.jpg
	$(CONVERT) -size $(THUMB_SIZE) image.jpg image_thumbnail.jpg
```

- Devient :

```
image_thumbail.jpg : image.jpg
	$(CONVERT) -size $(THUMB_SIZE) $< $@
```

Makefile : règles génériques

- Il est possible d'écrire des règles génériques :

```
%.jpg : %.png
	convert $< $@
```

- Permet de convertir tout fichier png en jpg à l'aide du programme *convert*

```
%_thumb.jpg : %.jpg
	convert -size 80x80 $< $@
```

- Il est enfin possible de séparer la liste des dépendances et la règle de création.

Makefile et compilation

```
CC=gcc
CFLAGS-=-Wall

prog : example.o hash.o
    $(CC) -o $@ $^
exemple.o : exemple.c hash.h
hash.o : hash.c hash.h
%.o :
    $(CC) $(CFLAGS) $< -o $@
```

- La dernière règle explique comment générer un fichier .o
- les deux règles au dessus donnent les dépendances

Dépendances

- C’est au développeur d’indiquer les dépendances.
- Ainsi, si un fichier header est modifié, tout fichier source qui inclut directement ou indirectement ce header doit être recompilé
- On peut demander à gcc d’analyser les fichiers et produire les règles de dépendances:

```
$> gcc -MM hash.c exemple.c
hash.o: hash.c hash.h
exemple.o: exemple.c hash.h
```

- Le résultat peut être inclus dans le Makefile avec la fonction *include*

include et gcc -MM

```
CC=gcc
CFLAGS=-Wall

prog: hash.o exemple.o
	$(CC) $^ -o $@

include dep
dep: hash.c exemple.c hash.h
	gcc -MM $^ > dep
```

```
pg106$ make
make: 'prog' is up to date.
pg106$ touch exemple.c
pg106$ make
gcc -MM hash.c exemple.c hash.h > dep
gcc -Wall   -c -o exemple.o exemple.c
gcc hash.o exemple.o -o prog
pg106$ touch hash.h
pg106$ make
gcc -MM hash.c exemple.c hash.h > dep
gcc -Wall   -c -o hash.o hash.c
gcc -Wall   -c -o exemple.o exemple.c
gcc hash.o exemple.o -o prog
pg106$ touch hash.c
pg106$ make
gcc -MM hash.c exemple.c hash.h > dep
gcc -Wall   -c -o hash.o hash.c
gcc hash.o exemple.o -o prog
pg106$
```

Makefile : PHONY

- On peut vouloir écrire des règles qui ne génèrent pas de fichier :

```
all

install

clean

distclean
```

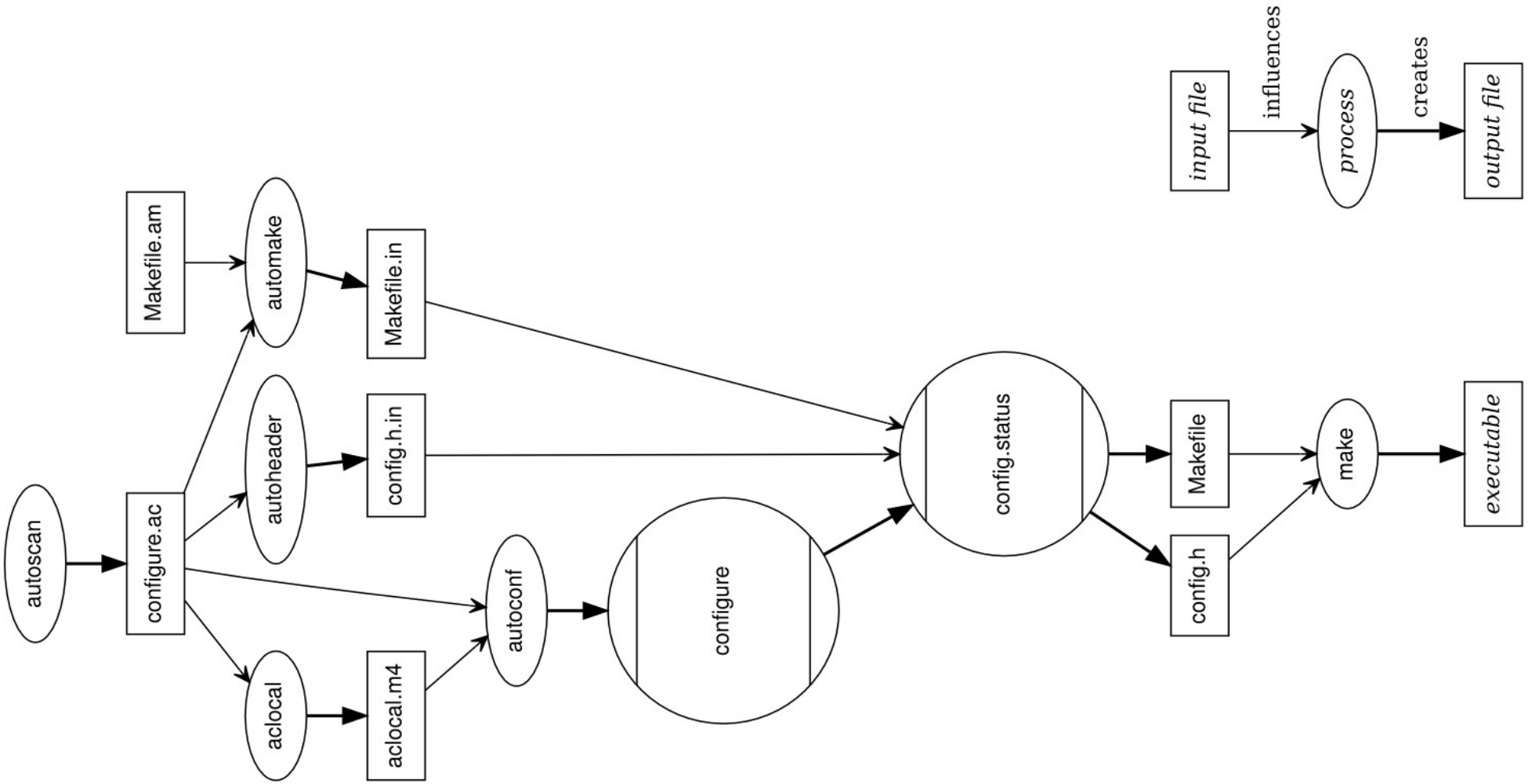
- On indique cela à make :

```
.PHONY=all install clean distclean
```

Makefile, automake...

- Makefile n'est pas spécifique à la compilation de programme
- Il ne gère pas nativement la dépendance entre les fichiers sources
- La prise en compte de l'environnement (compilateur, options, bibliothèques…) peut ce faire :
  - par l'édition des variables du Makefile (options de compilations, répertoire d'installation...)
  - par l'écriture de plusieurs Makefile (un par système)
  - par l'utilisation d'un générateur :
    - automake / autoconf
    - cmake

auto-tools : automake / autoconf



Auto-tools : principe général

- Le développeur écrit un fichier *configure.ac* :

```
AC_INIT([penguin], [2019.3.6], [seth@example.com])
AC_OUTPUT
AM_INIT_AUTOMAKE
AC_CONFIG_FILES([Makefile])
AC_PROG_CXX
```

- Le programme *autoconf* va utiliser les informations pour générer un script *configure*
- Le développeur écrit également un fichier *Makefile.am* :

```
bin_PROGRAMS = penguin
penguin_SOURCES = penguin.cpp
```

- Le script configure va utiliser ce fichier pour produire un **Makefile**

cmake

- CMake est un outil simplifié permettant la compilation de sources C et C++.
- C'est un outil multi-plateformes sous licence BSD
- Nécessite la présence d'un fichier **CMakeLists.txt**

- gestion automatique des dépendances
- Simple d'utilisation / facile à prendre en main

cmake : hello world !

- Un seul fichier main.c que l'on souhaite compiler en un programme « hello_world » :

```
project(HelloWorld)
cmake_minimum_required(VERSION 3.0)

add_executable(hello_world main.c)
```

- CMakeList.txt peut être découpé sur plusieurs répertoires avec des inclusions
- Un projet <<HELLO>> avec une bibliothèque dans le répertoire Hello et un programme d'exemple dans le répertoire Demo

cmake

- ./CMakeLists.txt

```
cmake_minimum_required (VERSION 3.0)
project (HELLO)

add_library(Hello hello.c)

add_executable (helloDemo demo.c demo_b.c)
target_link_libraries (helloDemo Hello)
```

- Si je structure mon projet en deux répertoires: Hello pour la bibliothèque et Demo pour l’utilisation...

cmake

- ./CMakeLists.txt:

```
cmake_minimum_required (VERSION 3.0)
project (HELLO)

add_subdirectory (Hello)
add_subdirectory (Demo)
```

- ./Hello/CMakeLists.txt

```
add_library (Hello hello.c)
```

- ./Demo/CMakeLists.txt

```
include_directories (${HELLO_SOURCE_DIR}/Hello)
link_directories (${HELLO_BINARY_DIR}/Hello)
add_executable (helloDemo demo.c demo_b.c)
target_link_libraries (helloDemo Hello)
```

cmake:cross platform make

- cmake est un système de compilation cross-plateformes. Il ne compile pas directement mais génère des fichiers dans différents formats :
  - Makefile
  - projet Visual Studio
  - Borland Makefile
  - projet Xcode
  - Kate
  - …
- cmake utilise les fichiers CMakeLists.txt et génère des fichiers en fonction de la plate-forme de compilation (Makefile, visual, xcode….).