Makefile++

Programmation séquentielle en C, 2022-2023

Orestis Malaspinas (A401) et un tout petit peu Michaël El Kharroubi

2023-02-24

Informatique et Systèmes de Communication, HEPIA

make avancé

Rappel: utilité

- Automatiser le processus de conversion d'un type de fichier à un autre, en *gérant les dépendances*.
- Effectue la conversion des fichiers qui ont changé uniquement.
- Utilisé pour la compilation:
  - Création du code objet à partir des sources.
  - Création de l'exécutable à partir du code objet.
- Tout "gros" projet utilise `make` (pas uniquement en `C`).
- Un `Makefile` bien écrit ne recompile que ce qui est **nécessaire**!
- Il existe d'autres outils pour le `C` et d'autres langages (`cmake`, `meson`, `maven`, `cargo`, ...).

Utilisation de `make`

`Makefile` simple

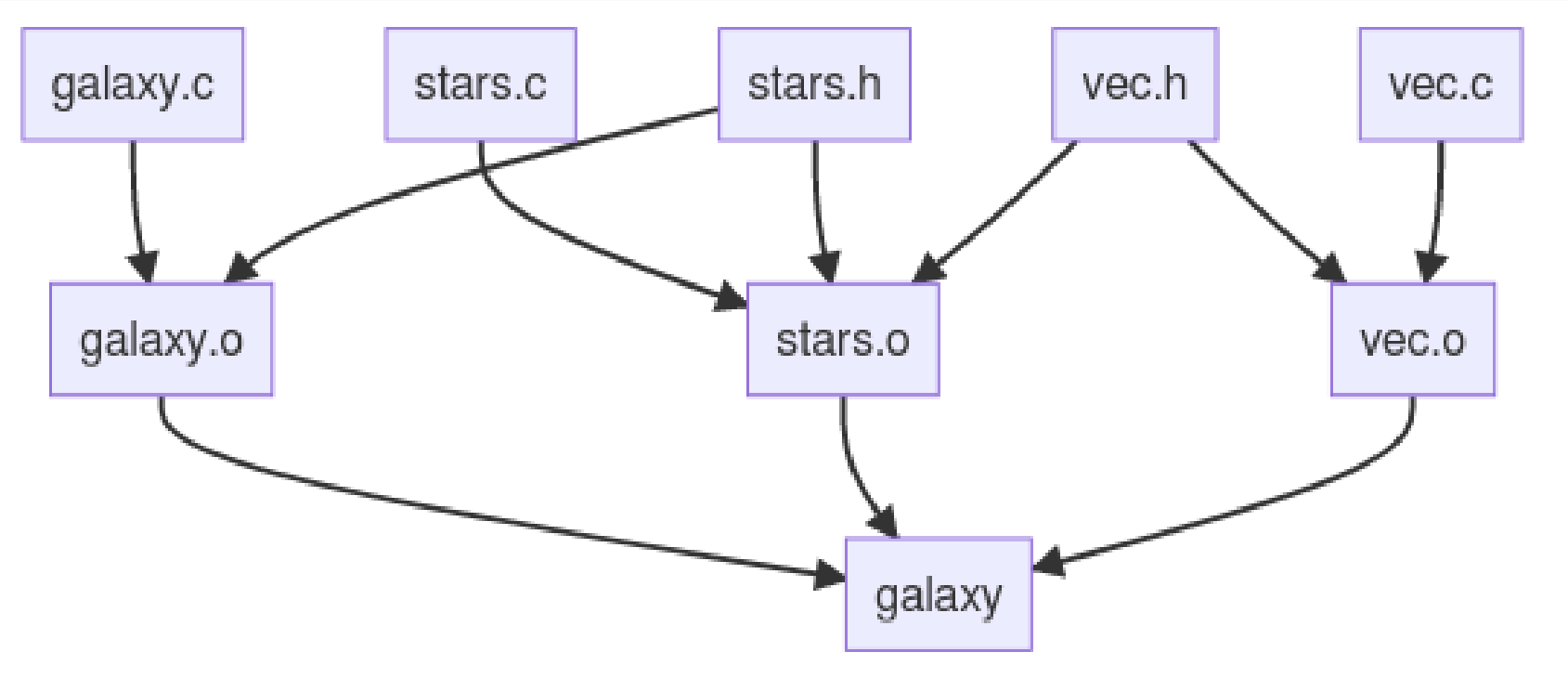
```
galaxy: galaxy.o stars.o vec.o
    gcc -o galaxy galaxy.o stars.o \
        vec.o
galaxy.o: galaxy.c stars.h
    gcc -c galaxy.c
stars.o: stars.c stars.h vec.h
    gcc -c galaxy.c
vec.o: vec.c vec.h
    gcc -c vec.c
clean:
    rm -f *.o galaxy
```

Terminal

```
$ make
gcc -c galaxy.c
gcc -c stars.c
gcc -c vec.c
gcc -o galaxy galaxy.o
    stars.o vec.o
$ make clean
rm -f *.o galaxy
```

Dessinez le diagramme de dépendances de ce `Makefile`.

Diagramme de dépendances



Variables

Variables utilisateur

- Déclaration

```
id = valeur
id = valeur1 valeur2 valeur3
```

- Utilisation

```
$(id)
```

- Déclaration à la ligne de commande

```
make CFLAGS="-O3 -Wall"
```

Variables internes

- `$@` : la cible
- `$^` : la liste des dépendances
- `$<` : la première dépendance
- `$*` : le nom de la cible sans extension

Makefile plus complexe (1/3)

```
# Un Makefile typique

# Le compilateur
CC = gcc

# La variable CFLAGS contient les flags de compilation:
#  -g                  compile avec les infos de debug
#  -Wall               Plein de warning
#  -Wextra             Encore plus de warnings
#  -pedantic           Warning lvl archimage
#  -O0                 Option d'optimisation (0,1,2,3)
#  -std=c11            Utilisation du standard c11
#  -fsanitize=address Utilisation des sanitizers
CFLAGS = -g -Wall -Wextra -pedantic -O0 -std=c11 -fsanitize=address
```

Makefile plus complexe (2/3)

```
# La variable LDFLAGS contient les flags pour l'éditeur
# de liens:
# -lm dit d'éditer les liens avec la lib math
# -lSDL2 dit d'éditer les liens avec la lib SDL2
LDFLAGS = -lm -lSDL2

# Définition des sources
SOURCES = galaxy.c stars.c vec.c
# OBJECTS contient SOURCES avec .c qui devient .o
OBJECTS = $(SOURCES:.c=.o)
# galaxy sera l'exécutable (galaxy.c contient le main)
TARGET = galaxy
# Jusqu'ici on a aucune cible.
```

Makefile plus complexe (3/3)

```
# TARGET est la première cible et sera donc exécuté à
# l'invocation de `make`. Elle dépend de OBJECTS
$(TARGET) : $(OBJECTS)
    $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)
# $@ : la cible, ^$ : la liste des dépendances

# PHONY signifie qu'on ne crée rien avec les cibles
# mentionnées. Ici clean est la cible utilisée pour
# enlever tous les fichier .o et l'exécutable
# exécute toujours clean, même si un ficher `clean` existe

.PHONY: clean

clean: # aucune dépendance
    rm -f $(TARGET) $(OBJECTS)
```

Gestion implicite (1/2)

Question

Pourquoi n'a-t-on pas besoin de générer les OBJECTS?

Réponse

`make` possède une série de règles implicites (voir ce lien pour une liste).

Fonctionnement

Si `make` rencontre une dépendance sans règle, il va voir dans sa liste de règles implicites pour la générer.

```
galaxy: galaxy.o stars.o vec.o
    gcc -o galaxy galaxy.o stars.o vec.o $(CFLAGS) $(LDFLAGS)
# implicitement pour galaxy.c, stars.c et vec.c
    $(CC) -c $< $(CFLAGS) -o $@
```

Gestion implicite (2/2)

Question

Et pour les dépendances des cibles implicites ça se passe comment?

Réponse

On peut définir individuellement les dites dépendances!

Fonctionnement

Quand `make` rencontre une dépendance sans règle, il va voir dans sa liste de règles implicites pour la générer.

```
# pas besoin des .c qui sont implicites
galaxy.o: stars.h vec.h
stars.o: *.h
vec.o: vec.h
```

On peut faire mieux

Il est possible de prendre **tous** les fichiers `.c`

```
SOURCES = $(wildcard *.c)
OBJECTS = $(SOURCES:.c=.o)
```

Ou encore mieux

```
OBJECTS := $(patsubst %.c,%.o,$(wildcard *.c))
```

That escalated quickly: *, %, :=, ...

```
# version "longue"
SOURCES = $(wildcard *.c)
OBJECTS = $(SOURCES:.c=.o)
# version "courte"
OBJECTS := $(patsubst %.c,%.o,$(wildcard *.c))
```

Let's take one step at a time:

- Les *,
- Les %, et leurs différences.
- Les fonctions, ici `wildcard` et `patsubst` (voir respectivement ce lien et ce lien).
- Le symbole := vs =.

Le symbole *

`make` peut “développer” (expand) des `wildcards` (*)

- dans les recettes le * est géré par le shell

```
clean:
    rm -f *.o galaxy
```

- dans les dépendances

```
galaxy.o: *.h
```

mais des fichiers `.h` doivent exister sinon il interprète `*.h` le nom du fichier.

Par contre ile ne peut pas

```
OBJECTS = *.o
# il faut utiliser
OBJECTS := $(wildcard *.o) # retourne tous les fichier .o
```

La différence entre * et %

- Le symbole * sert à générer une *liste* d'objets.
- Le symbole % sert comme *emplacement* (placeholder).

Exemple

```
%.o: %.c # % est la partie avant .c
    $(CC) -o $@ -c $< $(CFLAGS) # la règle pour chaque `%.c`
# équivalent à
galaxy.o: galaxy.c
stars.o: stars.c
vec.o: vec.c
```

Application

```
$(patsubst # Substitution de texte pour chaque
    %.c,\ # Le pattern "avant" le .c
    %.o,\ # Le pattern "avant" le .o
    $(wildcard *.c)\ # Tous les fichiers .c
    )
```

Le symbole := vs = (1/2)

Deux façon (flavors) d'assigner des variables (voir ce lien):

Le symbole =

- Façon **récursive**:

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?
```

  ici `foo` vaut `Huh?`.

- Valeurs remplacées "récursivement".
- Variables remplacées à chaque appel (lent + imprédictible!).

Le symbole := vs = (2/2)

Deux façon (flavors) d'assigner des variables (voir ce lien):

Le symbole :=

- Façon **simplement développée** (`Simply expanded variable`):

```
x := foo
y := $(x) bar
x := later
```

  ici `y` vaut `foo bar` et `x` vaut `later` (avec des `=`, `y` vaudrait `later bar`).

- Les variables se comportent comme en `C`.
- En particulier dans les *longs* `Makefile` le comportement est plus prédictible.