Makefile / makedepend

AP2 - programmation objet en C++

Semestre 2, année 2009-2010

Département d'informatique
IUT Bordeaux 1

Février 2010

Le projet

Situation : écrire un projet de simulation de courses de véhicules.
Décomposition proposée :

- un programme principal
- une classe pour les véhicules
- des fonctions pour les calculs physiques

Les sources

Avec cette décomposition, 5 fichiers sources

- programme principal :
  `jeu.cc`
- classe véhicules :
  `Vehicule.cc`, `Vehicule.h`
- fonctions physiques :
  `physique.cc`, `physique.h`

Les dépendances

- fabrication de l'exécutable
  `jeu` := éd. liens (`jeu.o`, `Vehicule.o`, `physique.o`, bib C++)
- #include's entre sources

Source	fichiers inclus
jeu.cc	Vehicule.h, physique.h
Vehicule.cc	Vehicule.h, physique.h
physique.cc	physique.h

Un makefile pour commencer

Makefile

```
jeu : jeu.cc Vehicule.cc physique.cc
	g++ -o jeu jeu.cc Vehicule.cc physique.cc
```

- critique : recompile tous les sources à chaque modification
- remède : passer à la compilation séparée

Avec compilation séparée

Makefile

```
objets = jeu.o Vehicule.o physique.o

jeu : $(objets)
        g++ -o jeu $(objets)

jeu.o : jeu.cc Vehicule.h physique.h
        g++ -c jeu.cc

Vehicule.o: Vehicule.cc Vehicule.h physique.h
...
```

- critique : lourd et fastidieux.
- simplifier en utilisant les actions par défauts

Action par défaut

- Par défaut, la commande `make` fabrique un module objet (.o) à partir d'un source C++ (.cc) en appelant le compilateur C++.
- Il est donc inutile de le dire dans le Makefile.

avec actions par défaut

Makefile

```
objets = jeu.o Vehicule.o physique.o

jeu : $(objets)
        g++ -o jeu $(objets)

jeu.o : jeu.cc Vehicule.h physique.h
Vehicule.o: Vehicule.cc Vehicule.h physique.h
...
```

- Les dépendances se déduisent des “#include” des sources
- fabrication automatique par makedepend

Utiliser makedepend

1. **Makefile initial**

```
objets = jeu.o Vehicule.o physique.o

jeu : $(objets)
        g++ -o jeu $(objets)
```

2. on lance : `makedepend *.cc`

3. **lignes ajoutées automatiquement**

```
# DO NOT DELETE
jeu.o: Vehicule.h physique.h
physique.o: physique.h
Vehicule.o: Vehicule.h
```

Dépendances implicites

Comment fabriquer `jeu.o` ?

- le fichier `jeu.cc` existe dans le répertoire
- `make` en déduit une dépendance implicite

  `jeu.o : jeu.cc`
  - qui s'ajoute aux dépendances trouvées par `makedepend`
  - et qui suggère l'emploi du compilateur C++

variables cible et dépendances

On peut simplifier la règle de fabrication de l'exécutable

```
jeu : $(objets)
        g++ -o jeu $(objets)
```

avec les variable de Makefile

- $@ : la cible
- $^ : la liste des dépendances

résultat

```
jeu : $(objets)
        g++ -o $@ $^
```

variables

Mieux, employer des variables prédéfinies :

résultat

```
jeu : $(objets)
        $(LINK.cc) -o $@ $^
```

La variable LINK.cc utilise d'autres variables

`$(CXX) $(CXXFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)`

notamment : `CXX = g++`

Options de compilation

En agissant sur ces variables, on modifie les options de compilation pour tout le Makefile

résultat

```
CXXFLAGS = -Wall      # avertissements à la compilation
LDFLAGS  = -s         # strip pendant l'édition des liens

jeu : $(objets)
        $(LINK.cc) -o $@ $^
```

Compilation C++ par défaut

La règle de fabrication des fichiers objets à partir des sources C++ est la suivante

extrait de “make -p”

```
%.o: %.cc
        $(COMPILE.cc) $(OUTPUT_OPTION) $<
```

$< = la première dépendance
La variable COMPILE.cc contient

```
$(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
```

Compilation par défaut

Comment fabriquer `jeu.o` ?

- le fichier `jeu.cc` existe dans le répertoire
- `make` en déduit une dépendance implicite

  `jeu.o : jeu.cc`
  - qui s'ajoute aux dépendances trouvées par `makedepend`
  - et qui suggère l'emploi du compilateur C++

Un exemple plus complet

On fait du “test driven development” : pour chaque module, en écrit un programme de test unitaire.
Permet de faire rejouer tous les tests systématiquement au lieu de se les “retaper” à la main.
Exemple : si le module `util.cc` contient une fonction factorielle `int fac(int n} {...}`

test unitaire de la factorielle

programme de test :

testutil.cc

```
#include <cassert>
#include "util.h"

int main() {
  assert( fac(0) == 1);
  assert( fac(1) == 1);
  assert( fac(5) == 120);
  ...
}
```

retour au projet

Maintenant, 7 fichiers source

- programme principal :
  `jeu.cc`
- classe véhicules :
  `Vehicule.cc`, `Vehicule.h`, testVehicule.cc
- fonctions physiques :
  `physique.cc`, `physique.h`, `testphysique`

et 3 exécutables :

- `jeu`, `testVehicule`, `testphysique`

Makefile

```
jeu : jeu.o Vehicule.o testphysique.o
      $(LINK.cc) -o $@ $^

testVehicule : testVehicule.o Vehicule.o testphysique.o
      $(LINK.cc) -o $@ $^

testphysique : testphysique.o
      $(LINK.cc) -o $@ $^
```

on lance `makedepend *.cc`, et le Makefile est prêt...

Encore mieux

```
%: %.o
	$(LINK.cc) -o $@ $^

jeu : jeu.o Vehicule.o testphysique.o
testVehicule : testVehicule.o Vehicule.o testphysique.o
testphysique : testphysique.o
```

(définition d'une règle)

avec quelques améliorations

Makefile

```
CXXFLAGE = -Wall

%: %.o
	$(LINK.cc) -o $@ $^

jeu           : jeu.o Vehicule.o testphysique.o
testVehicule : testVehicule.o Vehicule.o testphysique.o
testphysique : testphysique.o

depend:
	makedepend *.cc

clean:
	-rm *.o *~

# DO NOT DELETE
```