Techniques de production de programmes

par Matthieu Herrb

Version 1.2
Avril 1999

Centre National de la Recherche Scientifique
Laboratoire d'Analyse et d'Architecture des Systèmes

Table des matières

Chapitre I

Organisation des fichiers sources

Dans un projet de taille importante, il n'est pas question d'avoir un seul fichier source contenant l'ensemble du code. Le fichier en question serait beaucoup trop lourd à manier dans un éditeur de texte, et de plus les recompilations prendraient beaucoup trop de temps.

De plus, si l'on songe aux possibilités de réutilisation du logiciel, il est important de garder des éléments autonomes, utilisant une interface clairement définie avec le reste du système.

I.1 Unités de compilation

On appelle *unité de compilation* un ensemble de fichiers sources destiné à être compilé en un seul appel du compilateur. Il est important de noter que l'unité de compilation comprend en plus du fichier `.c`, tous les fichiers d'entête inclus dans ce fichier principal.

Le résultat de la compilation d'une unité de compilation produit un fichier objet (suffixe `.o`). L'ensemble des fichiers objets formant un programme sont ensuite rassemblés par l'*édition des liens* qui essaye de résoudre l'ensemble des références aux symboles non définis des différents fichiers objet.

I.2 Bibliothèques

Lorsque le nombre d'unités de compilation augmente trop, il est souhaitable de les regrouper en paquets plus importants : les bibliothèques. Sous Unix, une bibliothèque est identifiée par le suffixe `.a` et porte souvent un nom commençant par `lib`[1].

Une bibliothèque Unix se construit en deux étapes :

1. rangement des fichiers objets dans la bibliothèque (commande `ar`).

1. En anglais, bibliothèque = *library*

2. construction de la table des matières de la bibliothèque, utilisée par l'éditeur de liens pour accélérer les recherches (commande `ranlib`).

Les systèmes Unix dérivés de Unix System V (dont Solaris 2.x de Sun) ont intégré les deux étapes dans la commande `ar`, rendant `ranlib` inutile. Néanmoins, pour préserver la portabilité, on garde la référence à `ranlib` dans les fichiers de construction de programme.

Pour choisir comment découper une bibliothèque en unités de compilation, voici quelques éléments :

– S'il n'y a pas de variables globales, ou que les variables globales sont publiques, le plus efficace est de faire un fichier source par fonction de la bibliothèque.
– Si plusieurs fonctions partagent des données privées, il est préférable de les conserver dans un seul fichier source, à condition que la taille de ce dernier reste raisonnable (quelques centaines de lignes au maximum). Si non, il vaut mieux découper.
– Essayer de toujours maintenir les fonctions dépendant d'autres bibliothèques dans des fichiers séparés, afin de rester indépendant de ces autres bibliothèque si ces fonctions ne sont pas utilisées.

Il existe un second type de bibliothèque, appelé bibliothèques partagées ou bibliothèques dynamiques, identifié par l'extension `.so`. Il n'en sera pas question ici.

I.3 Fichiers d'entête

On associe un fichier d'entête (suffixe `.h`) à une bibliothèque ou à un sous-ensemble d'une bibliothèque.

Un fichier d'entête est inclus par les autres fichiers source à l'aide de la directive `#include "fichier.h"`.

Un fichier d'entête C contient :

– les définitions des types (`typedef`) manipulés par la bibliothèque,
– les constantes utilisées par la bibliothèque, définies par des énumérations ou des macros,
– les déclarations des variables globales exportées par la bibliothèque,
– les prototypes des fonctions exportées par la bibliothèque.

Ce fichier est destiné à être inclus par tous les fichiers sources qui utilisent la bibliothèque. Il doit également être inclus dans les fichiers source de la bibliothèque elle-même, afin de vérifier à la compilation la cohérence des déclarations.

Dans le cas de bibliothèques complexes, on peut avoir en plus du fichier d'entête public décrit ci-dessus, un ou plusieurs fichiers d'entête privés, inclus uniquement par les fichiers source de la bibliothèque, mais pas par ceux qui l'utilisent.

I.4 Édition des liens

Lorsqu'un programme est composé de plusieurs unités de compilation, l'étape d'*édition des liens* permet de rassembler les morceaux pour produire le fichier exécutable.

Sous Unix, le programme qui réalise cette opération (l'édition des liens) est `ld`, mais il est rarement appelé directement. Il est plus commode d'utiliser la commande `cc` qui fait l'interface.

I.4.1 Fonctionnement

Le fonctionnement de l'éditeur de liens en langage C est le suivant :

– Il utilise les fichiers qu'on lui passe en argument *dans l'ordre* où ils sont donnés.
– Il commence en cherchant le symbole `main`, correspondant au programme principal.
– Quand il a trouvé le fichier contenant le symbole `main`, il ajoute tous les symboles non résolus de ce fichier à sa liste de symboles à trouver.
– Chaque fois qu'un symbole est trouvé, il est retiré de la liste des symboles non résolus, et ajouté à la liste des symboles connus. Tous les symboles non résolus du nouveau fichier sont ajoutés à la liste des symboles non résolus.
– Si un symbole défini dans un fichier utilisé se trouve déjà dans la liste des symboles connus, l'erreur « symbol multiply defined » est émise.
– Si arrivé à la fin de la liste des fichiers, il reste des symboles non résolus, l'erreur « undefined symbol » est émise.
– Une fois que tous les symboles sont résolus, le fichier exécutable est produit en remplaçant chaque symbole par sa valeur réelle.

I.4.2 Recherche des bibliothèques

Lorsqu'on utilise `cc` pour appeler l'éditeur de liens indirectement, l'option -l permet de désigner des bibliothèques système, qui sont cherchées dans le répertoire `/usr/lib`. L'option -L permet d'ajouter des répertoires pour la recherche de ces bibliothèques.

Si l'option `-lxxx` est spécifiée, l'éditeur de liens cherchera la bibliothèque `libxxx.a` dans `/usr/lib` et les répertoires indiqués par les options `-L`.

Ainsi lorsqu'un programme est compilé avec l'option `-lm`, c'est la bibliothèque `/usr/lib/libm.a` qui est utilisée.

La variable d'environnement `LD_LIBRARY_PATH` permet d'ajouter des répertoires au chemin de recherche des bibliothèques.

I.4.3 Importance de l'ordre

Comme cela a été dit au § I.4.1, l'éditeur des liens parcourt les bibliothèques une seule fois, dans l'ordre où elles sont données sur la ligne de commande.

Ce mode de fonctionnement provoque parfois quelques désagréments pour le programmeur :

– Si un fichier objet ou une bibliothèque est placé à endroit où aucun de ses symboles n'ont été référencés, il ne sera pas utilisé. Si plus tard des références à ses symboles apparaissent, ils resteront non définis.
  En particulier, les fichiers objets et les bibliothèques placés avant celui contenant la définition de la fonction main() sont ignorés.
– Si deux bibliothèques contiennent des appels croisés à partir de fichiers différents, l'édition des liens ne peut pas se faire directement : il faut spécifier deux fois l'une des bibliothèques. Cela se produit lorsqu'il y a un cycle dans le graphe d'appel des fonctions et que les nœuds du cycle sont définis dans des fichiers séparés appartenant à plusieurs bibliothèques.

Chapitre II

Utilisation de make

Le programme `make` permet d'automatiser la recompilation d'un programme composé de plusieurs modules objet ou de plusieurs bibliothèques.

II.1 Un exemple

Commençons ce chapitre par un exemple qui illustrera autant le rôle de `make` que son utilisation. Considérons un programme de calcul numérique qui lit un fichier de données, en extrait quelques courbes à l'aide de formules de calcul matriciel et les affiche. Ce programme `calcul` peut être décomposé en quatre fichiers source et trois fichiers d'en-tête :

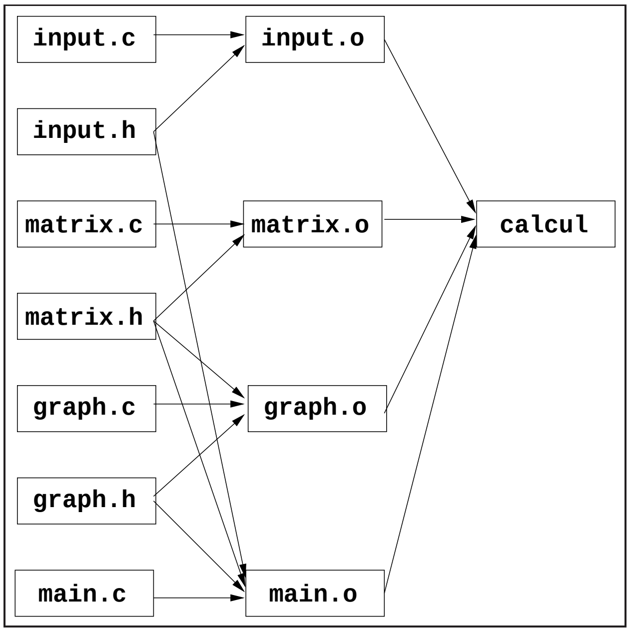
- `input.c` contenant les fonctions de lecture du fichier de données,
- `matrix.c` contenant les fonctions de calcul matriciel utilisées,
- `graph.c` contenant les fonctions d'affichage graphique,
- `main.c` contenant le programme principal,
- `input.h`, `matrix.h` et `graph.h` les fichiers d'en-tête correspondant aux trois premiers fichiers source.

Supposons de plus que les routines graphiques utilisent quelques fonctions de calcul matriciel. Les inclusions de fichiers d'en-tête autres que les fichiers standard seront donc les suivantes :

Fichier source	Fichiers inclus
input.c	input.h
matrix.c	matrix.h
graph.c	graph.h matrix.h
main.c	input.h matrix.h graph.h

Pour compiler le programme il est nécessaire de compiler chaque fichier source, puis de réaliser l'édition des liens à l'aide de la séquence de commandes suivante :

Fig. II.1 – *Graphe de dépendances*




```
cc -c input.c
cc -c matrix.c
cc -c graph.c
cc -c main.c
cc -o calcul main.o input.o graph.o matrix.o -lm
```

Chaque fois qu'un fichier est modifié, il faut réexécuter tout ou partie de ces commandes pour prendre en compte les modifications :

- si `input.h` est modifié, il faut recompiler `input.c` et `main.c`,
- de même, si `graph.h` est modifié, il faut recompiler `graph.c` ainsi que `main.c`,
- si `matrix.h` est modifié, il faut recompiler `matrix.c` et `main.c` et en plus `graph.c`,
- si l'un des fichiers `.c` est modifié, il faut le recompiler,
- enfin, après toute recompilation d'un fichier, il faut refaire l'édition des liens.

Cette description en langage naturel correspond à un *graphe de dépendances* représenté sur la figure II.1. Remarquez que les fichiers objets (`.o`) sont représentés explicitement comme noeuds du graphe. Cela est indispensable pour exprimer correctement le problème des dépendances.

En fait chaque fois que l'on touche à un fichier source, il faut parcourir ce graphe pour retrouver quels sont les fichiers à recompiler. On donne ce graphe exprimé dans une syntaxe particulière à `make` et il se charge de déterminer quels sont les fichiers qui doivent être recompilés, puis d'appeler les commandes qu'on lui à indiquées pour réaliser ces compilations.

FIG. II.2 – *Makefile pour* `calcul`

```
calcul: main.o input.o matrix.o graph.o
        cc -o calcul main.o input.o matrix.o graph.o -lm

main.o: main.c input.h matrix.h graph.h
        cc -c main.c

input.o: input.c input.h
        cc -c input.c

matrix.o: matrix.c matrix.h
        cc -c matrix.c

graph.o: graph.c graph.h matrix.h
        cc -c graph.c
```

II.2 Le premier Makefile

Le fichier `Makefile` est utilisé pour donner en entrée de `make` les dépendances et les commandes utilisées pour les recompilations. La figure II.2 présente le fichier `Makefile` pour notre exemple.

Ce fichier reprend exactement la description en langage naturel que nous avons vue ci-dessus, et que nous avons affinée en dessinant le graphe :

- le programme `calcul` dépend des quatre fichiers objets `main.o`, `input.o`, `matrix.o` et `graph.o`. Pour l'obtenir à partir de ces quatre fichiers, il faut exécuter la commande :

  ```
  cc -o calcul main.o input.o matrix.o graph.o -lm
  ```

  Le fait que la ligne décrivant `calcul` soit la première indique à make que c'est lui le programme à produire.
- le fichier objet `main.o` dépend des fichiers sources `main.c`, `input.h`, `matrix.h` et `graph.h` et la commande pour l'obtenir est :

  `cc -c main.c`
- le fichier objet `input.o` dépend des fichiers sources `input.c` et `input.h` et la commande pour l'obtenir est :

  `cc -c input.c`
- le fichier objet `matrix.o` dépend des fichiers sources `matrix.c` et `matrix.h` et la commande pour l'obtenir est :

  `cc -c matrix.c`
- le fichier objet `graph.o` dépend des fichiers sources `graph.c`, `matrix.h` et `graph.h` et la commande pour l'obtenir est :

  `cc -c graph.c`

Pour utiliser ce fichier `Makefile` il suffit de le créer dans le répertoire qui contient les fichiers sources de d'exécuter la commande :

```
make
```

Normalement, si notre programme a déjà été compilé comme indiqué plus haut, il ne se passe rien. Voyons maintenant ce qui se passe si l'on modifie quelque chose dans un fichier, par exemple dans le `input.h`, puis que l'on tape la commande `make` :

`Make` constate que `input.h` a été modifié plus récemment que `main.o`. Il doit donc exécuter la commande :

```
cc -c main.c
```

pour mettre à jour `main.o`. De même, `input.o` est moins récent que le fichier modifié `input.h` et `make` va exécuter la commande :

```
cc -c input.c
```

Enfin, `make` va constater que `main.o` et `input.o` sont plus récents que `calcul` et il va donc exécuter :

```
cc -o calcul main.o input.o matrix.o graph.o -lm
```

pour refaire l'édition des liens et produire un nouvel exécutable.

II.3 Éléments d'un Makefile

Cette section présente les composantes de base d'un fichier `Makefile` qui décrit comment compiler un programme.

On peut donner le nom que l'on veut à un *Makefile*. `Makefile` est simplement le nom par défaut utilisé sur la plupart des systèmes d'exploitation. Si vous donnez un autre nom à votre Makefile, il faudra l'indiquer à `make` avec l'option « `-f` *nom* ».

Un Makefile est une sorte de programme avec des déclarations, des instructions et des commentaires. Voici les cinq types d'éléments différents que l'on peut trouver dans un Makefile :

– des *commentaires* introduits par le caractère « `#` », ils se prolongent jusqu'à la fin de la ligne. Comme cela se retrouve presque dans tous les langages, les commentaires sont ignorés par `make`. Ils sont destinés à l'utilisateur, pour documenter son Makefile.

– des *règles explicites* qui décrivent quand et comment refaire un ou plusieurs fichiers, appelés les *cibles* de la règle. Dans notre exemple, nous avions cinq règles explicites.

  Chaque règle explicite est composée de deux parties : la première décrit les fichiers dont dépendent les cibles (les *dépendances*) et la seconde décrit les commandes à exécuter pour produire la cible.

– des *règles implicites* qui décrivent de manière générique quand et comment refaire des fichiers, sur la base de leurs noms. Elles permettent d'éviter d'écrire de nombreuses règles explicites très semblables, rendant ainsi les Makefiles plus synthétiques.

– des *définitions de variables* qui associent une chaîne de caractères à un nom de variable. Les références à ces variables sont ensuite remplacées par leur valeur pendant l'évaluation du Makefile. Les variables servent principalement à définir des listes de fichiers auxquelles appliquer le même traitement et à positionner des options.

– des *directives* qui permettent de demander à `make` d'inclure le contenu d'un autre fichier ou de décider (à partir de la valeur d'une variable) quelle(s) partie(s) du Makefile prendre en compte ou ignorer.

Lorsque `make` est appelé sans argument, il prend comme cible à construire la cible de la première règle explicite qu'il trouve dans le fichier.

II.4 Règles

Nous avons déjà vu des exemples de règles. Les règles explicites ou implicites ont la même structure :

CIBLES : *DÉPENDANCES*
COMMANDE
...

Les *cibles* sont des noms de fichiers à construire ou à mettre à jour, séparés par des espaces. Les *dépendances* sont également des noms de fichiers, dont dépendent les cibles, séparés par des espaces. On trouve un « `:` » pour séparer les deux.

Cibles et dépendances doivent obligatoirement se trouver sur la même ligne, mais on peut utiliser le caractère « `\` » à la fin d'une ligne pour indiquer à `make` d'ignorer la coupure.

Les lignes contenant les *commandes* viennent ensuite. Elles commencent obligatoirement par une tabulation, et on peut en mettre autant que nécessaire. Une commande doit tenir sur une seule ligne, mais, là aussi, on peut utiliser l'antislash pour prolonger les lignes. La liste des commandes se termine par une ligne qui ne contient pas une tabulation comme premier caractère.

Attention, comme les espaces et tabulations avant un commentaire sont ignorés, une ligne ne contenant qu'un commentaire termine toujours une liste de commandes.

De manière générale, `make` est très sensible au bon usage des espaces et des tabulations. Il ne s'agit pas de les mélanger ou de les utiliser n'importe où pour obtenir le bon résultat. (Ce qui est assez embêtant, puisque la plupart des éditeurs de texte ne savent pas trop montrer la différence).

Dans une règle explicite, les cibles, dépendances et les commandes n'utilisent que des noms de fichiers explicites, alors que dans les règles implicites le caractère « `%`» est utilisé pour désigner un ensemble de fichiers de manière générique.

Un mot commençant par le caractère dollar « `$` » est utilisé pour désigner une variable. Pour insérer un dollar dans une règle ou une commande, il suffit de le doubler : « `$$` ».

Le comportement général d'une règle est, comme nous l'avons déjà vu, de construire les cibles à partir des sources décrits dans les dépendances, en exécutant les commandes qui sont données dans la règle. Avant d'exécuter les commandes d'une règle, `make` regarde si les sources listées dans les dépendances apparaissent comme cible dans une autre règle du Makefile. Si c'est le cas, `make` applique d'abord cette nouvelle règle avant d'exécuter les commandes de la première règle, et ainsi de suite récursivement jusqu'à trouver des dépendances qui n'apparaissent dans aucune cible. Les commandes des différentes règles utilisées sont alors exécutées dans l'ordre inverse.

De manière générale, un fichier ne peut apparaître qu'un seule fois comme cible dans les règles d'un Makefile. Il y a toutefois une exception : si une règle ne contient pas de commandes, on dit que c'est une règle de dépendance. Toutes les règles de dépendance trouvées pour une même cible sont concaténées pour déterminer les dépendances d'un fichier cible. Par contre, si deux règles contiennent des commandes pour le même fichier, le comportement de `make` n'est pas défini. La plupart des versions de make n'utiliseront que la première règle trouvée, mais n'est pas garanti. Voyez la section II.12 pour apprendre comment utiliser plusieurs règles pour construire une cible.

Ce principe est valable aussi pour les règles implicites : une seule règle implicite doit s'appliquer pour une cible donnée. Par contre les règles explicites ont toujours la priorité : s'il existe une règle explicite pour construire une cible ou pour décrire ses dépendances, les règles implicites qui pourraient correspondre à la même cible sont ignorées.

II.5 Commandes

En général, `make` affiche les commandes avant de les exécuter, et s'arrête dès qu'une commande a produit une erreur, ou qu'elle est interrompue. Un caractère ajouté en tête d'une commande permet de contrôler ces comportements.

II.5.1 Affichage des commandes

Si une commande est précédée par le caractère « `@` », make n'affiche pas cette commande avant de l'exécuter. Ceci est utilisé notamment lorsque la commande est elle-même une commande d'affichage telle que `echo`. Cela permet d'éviter d'afficher deux fois le message.

Exemple :

```
prog: a.o b.o c.o
	@echo 'Edition des liens...'
	cc -o prog a.o b.o c.o
```

L'option « `-s` » de `make` permet de supprimer globalement l'affichage des commandes.

II.5.2 Exécution de commandes

Les commandes décrites dans une règle de Makefile sont exécutées une par une par l'interpréteur de commandes du système (le *shell*). Sur Unix, c'est `/bin/sh` qui est utilisé, indépendamment de l'interpréteur choisi par un utilisateur donné, sauf si le Makefile indique un autre interpréteur.

C'est l'interpréteur de commandes qui défini ce que l'on peut trouver dans une commande. À condition de tout faire tenir sur une seule ligne (avec l'aide de « `\` »), on peut utiliser des constructions complexes telles que tests, boucles,...

Si plusieurs commandes sont listées dans une règle, elles sont exécutées séquentiellement, chaque fois par un nouvel interpréteur de commandes. Il faut faire attention à ce détail lorsque l'on veut modifier l'environnement du shell ou le répertoire courant dans les commandes.

```
dir/prog: a.o b.o
	cd dir
	cc -o prog ../a.o ../b.o
```

est incorrect. Un premier interpréteur de commandes exécute la commande `cd` puis se termine. Le répertoire courant de `make` n'a pas changé. La seconde commande, prévue pour s'exécuter dans le sous-répertoire `dir` est en réalité exécutée dans le répertoire au-dessus, et va vraisemblablement provoquer une erreur.

Par contre la règle suivant est correcte :

```
dir/prog: a.o b.o
        cd dir ; \
        cc -o prog ../a.o ../b.o
```

En effet, la commande est composée une seule ligne (le \ masquant le retour à la ligne) et est donc interprétée en une seule fois par le shell. Remarquez que la commande aurait pu également être écrite directement sur une seule ligne. Par contre, le point-virgule est indispensable pour séparer les deux commandes aux yeux de l'interpréteur.

II.5.3 Traitement des erreurs

`Make` vérifie le résultat de l'exécution de chaque commande. S'il n'y a pas d'erreur, il passe à la commande suivante. Après avoir exécuté avec succès la dernière commande, une règle est terminée avec succès : `make` considère que la cible a été correctement construite.

S'il une erreur se produit lors de l'exécution d'une commande (code de retour différent de zéro), `make` abandonne l'exécution de la règle en cours.

Il peut arriver qu'une erreur dans l'exécution d'un commande ne soit pas fatale au bon déroulement de la suite. Par exemple, la commande `mkdir` provoque une erreur si le répertoire à créer existe déjà. Ce n'est pas forcément une erreur dans les commandes d'une règle de Makefile.

On peut utiliser le caractère « `-` » en tête d'une commande pour indiquer à `make` d'ignorer l'échec de cette commande.

Exemple :

```
clean:
        -rm *.o
        @echo 'done'
```

L'option `-i` permet de dire à `make` d'ignorer toutes les erreurs. Ce n'est pas très utile.

En général, lorsqu'une erreur se produit, make s'arrête complètement. Il y a des cas où un Makefile peut décrire la production de plusieurs fichiers qui sont relativement indépendants. L'option « `-k` » permet à `make` de n'interrompre que la chaîne des commandes dépendant d'une règle et d'examiner la règle suivante. Si durant le déroulement d'une nouvelle règle, make rencontre de nouveau la règle qui a provoqué l'erreur, aucune commande ne sera exécutée pour cette nouvelle règle. Par contre si la nouvelle règle ne rencontre pas la règle qui a provoqué l'erreur, elle sera déroulée normalement et les commandes associées seront exécutées.

II.5.4 Interruption de make

Lorsque `make` est interrompu (Par un Contrôle-C ou un signal), il interrompt à son tour l'exécution de la commande en cours. Si le fichier cible de la règle en cours d'exécution avait déjà été modifié, `make` le détruit avant de s'arrêter, pour éviter de laisser traîner un fichier cible qui n'est sans doute pas complet.

Si ce comportement n'est pas souhaité (par exemple parce que la cible reste utilisable même si elle est incomplète), on peut la déclarer dans les dépendances d'une cible spéciale : « `.PRECIOUS` ».

II.6 Utilisation des variables

Une variable est un nom défini pour représenter une chaîne de caractères, sa *valeur*. Une variable est remplacée par sa valeur chaque fois qu'elle apparaît sous forme de référence dans un Makefile, dans tous les éléments des règles ou les définitions d'autres variables et les directives de contrôle.

Les variables sont principalement utilisées pour représenter des listes de fichiers, des noms de programmes et leurs options, mais elles peuvent être utilisées pour tout ce que l'on peut imaginer.

La manière dont une variable est remplacée par sa valeur fait qu'elles sont plus proches des *macros* de certains langages (comme le pré-processeur C ou le LISP) de de véritables variables.

II.6.1 Noms des variables

Le nom d'une variable peut être composé de n'importe quelle séquence de caractères, à l'exception de « `:`, `#`, `=` » et des espaces ou tabulations. Il convient cependant d'éviter les noms de variables contenant autre chose que des lettres, des chiffres et du caractère « souligné (`_`) ». `Make` fait la différence entre majuscules et minuscules dans les noms de variables. Traditionnellement, on utilise des noms entièrement en majuscules pour les variables dans les Makefiles sous Unix.

II.6.2 Référence

Pour faire référence à la valeur d'une variable, on utilise un dollar suivi du nom de la variable entre parenthèses. (Certaines variantes de make autorisent également les accolades, par analogie avec les shells Unix). « `$(PIPO)` » est une référence à la variable `PIPO`.

Voici un exemple classique d'utilisation de variable :

```
OBJECTS = a.o b.o c.o
prog: $(OBJECTS)
	cc -o prog $(OBJECTS)

$(OBJECTS): common.h
```

Un dollar suivi d'un caractère autre qu'une parenthèse ouvrante (ou une accolade ouvrante) est considéré comme la référence à une variable dont le nom est constitué d'un seul caractère. Ainsi `$x` désigne la variable `x`.

Cet usage est fortement déconseillé, sauf pour les variables définies automatiquement par `make`.

Les références aux variables sont substituées récursivement jusqu'à ce qu'il n'apparaisse plus de nouvelle référence. Si `make` détecte une boucle, une erreur se produit. Ainsi le Makefile :

```
FOO = $(BAR)
BAR = $(UGH)
UGH = Huh?

all:
	@echo $(FOO)
```

affiche « `Huh?` » : `$(FOO)` est remplacé par `$(BAR)` qui est lui-même remplacé par `$(UGH)` qui est finallement remplacé par `Huh?`.

II.6.3 Définition

Pour définir la valeur d'une variable, on écrit son nom en début de ligne, suivi du signe « `=` » suivi du texte de remplacement. Les espaces en tête du texte de remplacement ne sont pas pris en compte. Par exemple :

```
OBJS = a.o b.o c.o
```

défini la variable `OBJS` dont la valeur est « `a.o b.o c.o` ».

Comme nous l'avons déja vu, la valeur d'une variable peut contenir des références à d'autres variables.

Tant que l'on n'a pas défini de valeur pour une variable, `make` considère que leur valeur est la chaîne vide.

Il existe un certain nombre de variables auquelles `make` affecte automatiquement une valeur. Elles sont décrites plus loin.

Il est souvent utile de pouvoir ajouter un élément à la valeur d'une variable. La construction classique qui peut venir à l'esprit ne marche pas avec la manière dont `make` expanse les variables : cela crée un boucle. La plupart des versions de `make` ont donc un opérateur supplémentaire « `+=` ». Ainsi :

```
OBJS = a.o b.o c.o
OBJS += d.o
```

affecte la valeur «`a.o b.o c.o d.o`» à la variable `OBJS`.

II.6.4 Variables d'environnement

`Make` hérite de toutes les variables d'environnement de l'interpréteur de commande qui l'exécute : toutes les variables d'environnement sont transformées en variables de `make`. Ainsi il est possible de définir une valeur par défaut d'une variable de `make` en définissant une variable d'environnement.

La définition d'une variable dans le Makefile écrase la valeur héritée de l'environnement.

Il est également possible de définir la valeur d'une variable lors de l'appel de make, à l'aide d'un argument du type `variable=valeur`. Ce type de définition remplace les définitions de l'environnement ou du Makefile.

II.6.5 Variables automatiques

On appelle ainsi les variables définies automatiquement par `make` en cours d'exécution. Elles sont particulièrement utiles pour l'écriture des règles implicites, afin de permettre d'expliciter les parties implicites des règles. Ces variables sont calculées pour chaque règle en cours d'évaluation.

Voici une table des principales variables automatiques :

`$@`	Le nom de la cible de la règle. Si une règle implicite comporte plusieurs cibles, c'est la cible qui a déclenché la règle.
`$<`	Le nom de la première dépendance. Si les commandes pour construire la cible proviennent d'une règle implicite, il s'agit de la première dépendance introduite par la règle implicite.
`$?`	Les noms de toutes les dépendances qui sont plus récentes que la cible, séparés par des espaces.
`$^`	Les noms de toutes les dépendances d'une cible, séparés par des espaces.
`$*`	Le radical associé à une règle implicite. Si la cible est `toto.o` et que la motif dans la règle est `%.o`, alors `$*` vaut `toto`. Cette variable est utile pour créer les noms de fichiers voisins. Dans le cas des règles explicites, il n'y a pas de motif. Le radical est souvent défini comme le nom de la cible sans suffixe. Il vaut mieux cependant éviter d'utiliser `$*` dans les règles explicites.

II.7 Utilisation de règles implicites

Les règles implicites permettent d'écrire en une seule règle un ensemble de règles explicites très semblables. Elles permettent de décrire de manière générique comment produire un type de fichier à partir d'un autre, en se basant sur les noms de fichiers (en général les suffixes) pour identifier les types.

Il y a deux catégories de règles implicites : un ensemble de règles implicites qui sont incorporées à make et des règles implicites définies par l'utilisateur dans son Makefile.

La manière dont sont définies les règles implicites incorporées à `make` varie d'une version de `make` à l'autre, ainsi que la définition exacte de ces règles. C'est pourquoi malheureusement, pour réaliser des Makefiles portables, on est en général conduit à réécrire toutes les règles qui peuvent être utilisées.

II.7.1 Définition d'une règle implicite

Note : Nous ne décrirons ici que la syntaxe moderne des règles implicites, basée sur un motif. La syntaxe plus ancienne basée uniquement sur les suffixes ne sera pas traitée.

Les règles implicites ont la même structure que les règles explicites. La différence est que les règles implicitent contiennent un caractère « `%` » (et un seul) dans le nom de la cible. Le caractère `%` est un motif qui peut être associé à n'importe quelle chaîne non vide. Par exemple, « `%.c` » est un motif qui peut être associé à n'importe quelle chaîne se terminant par « `.c` ». La chaîne associée à la partie variable du motif (le `%`) est appelée le radical.

Une fois qu'une règle a été sélectionnée par un motif, le caractère `%` est remplacé par le radical dans les dépendances de cette règle et les variables automatiques sont définies en fonction du radical.

Ainsi la règle « `%.o: %.c` » définit comment produire n'importe quel `toto.o` à partir du fichier `toto.c`.

Lorsque plusieurs règles implicites s'appliquent pour une cible, make sélectionne la première qui apparait dans le Makefile. L'ordre des règles a donc une importance dans un Makefile.

II.7.2 Exemples

Voici l'exemple de la règle implicite la plus souvent employée pour compiler des programmes en langage C :

```
%.o: %.c
        $(CC) -c -o $@ $(CFLAGS) $(CPPFLAGS) $<
```

Cette règle permet de produire n'importe quel fichier objet (suffixe `.o`) à partir du fichier source correspondant. Elle utilise les variables `CC`, `CPPFLAGS` et `CFLAGS` pour connaître la commande à utiliser, les options à passer au pré-processeur et au compilateur respectivement.

Enfin, les variables automatiques `$@` et `$<` désignent respectivement le nom du fichier à produire et le nom du fichier à compiler.

Un second exemple permet de donner une règle pour réaliser l'édition des liens d'un programme simple :

```
%: %.o $(LIBS)
        $(CC) -o $@ $(LDFLAGS) $< $(LIBS)
```

Cette règle s'applique à n'importe quel fichier à produire à condition que le fichier objet correspondant existe, ou qu'il existe dans le Makefile une règle pour le produire. Si la règle précédente est présente dans le même Makefile, il sera capable de produire le programme `toto` à partir du source `toto.c`.

Pour cela `make` commence par chercher une règle qui permette de produire `toto`. La seconde règle en est capable, si on peut trouver un moyen de produire `toto.o` (ou s'il existe déjà...). Justement, la première règle dit comment produire `toto.o` si on trouve `toto.c` ou un moyen de le produire. Comme `toto.c` existe, la recherche s'arrête, et on peut exécuter les commandes trouvées en remontant la chaîne : on produit d'abord `toto.o` puis on réalise l'édition des liens.

Il n'est pas nécessaire de définir les variables `CC`, `CPPFLAGS`, `CFLAGS` et `LDFLAGS` car elles ont des valeurs par défaut issues des règles incorporées de `make`.

II.8 Directives

Chaque variante de `make` définit sa propre syntaxe pour les directives utilisables dans les Makefiles. La seule directive qui ait été standardisée par la norme POSIX est la directive `include` qui permet d'inclure un fichier dans un Makefile.

La syntaxe de la directive `include` est :

`include` *nom-de-fichier*

en début de ligne. *nom-de-fichier* peut contenir des références à des variables.

Si le nom de fichier obtenu après expansion éventuelle des variables est un chemin absolu (commençant par `/` sur Unix), le fichier est localisé de manière absolue. S'il s'agit d'un chemin relatif, il est recherché d'abord à partir du répertoire courant, puis à partir d'un certain nombre de répertoires système.

Certaines variantes de `make` disposent de l'option `-I` permettant de spécifier d'autres répertoires où chercher les fichiers inclus.

II.9 Construction de bibliothèques avec make

Pour la construction de bibliothèques, `make` a besoin de quelques outils supplémentaires. En effet, les outils Unix standard ne permettent pas facilement de tester si un fichier objet stocké dans une bibliothèque est plus ancien que le fichier source correspondant.

Pour faire cela, il faudrait extraire le fichier objet de l'archive pour faire le test, ce qui est très inefficace et rendrait totalement inintéressante l'utilisation des bibliothèques.

II.9.1 Désigner un élément d'une bibliothèque comme cible

Pour désigner un fichier objet dans une bibliothèque comme cible dans un Makefile, on peut utiliser la construction suivante :

BIBLIOTHÈQUE`(`*ELEMENT*`)`

Cette construction est utilisable dans les cibles et les dépendances, pas dans les commandes. *Bibliothèque* désigne le nom du fichier contenant la bibliothèque, par exemple `libvect.a` et *element* désigne le nom du fichier objet à l'intérieur de l'archive, par exemple `add.o`.

Avec cette extension, on peut écrire une règle pour compiler un fichier C et placer le fichier objet dans une bibliothèque à l'aide de la commande `ar` :

```
libvect.a(add.o): add.c
	cc -c -o add.o add.c
	ar cr libvect.a add.o
	rm add.o
```

Il faut également produire une table des matières pour une bibliothèque lorsque tous ses éléments sont rangés à l'aide de la commande `ranlib`. Cela se fait en désignant la bibliothèque complète comme cible et les éléments comme dépendances :

```
libvect.a: libvect.a(add.o) libvect.a(prod.o) libvect.a(sub.o)
	ranlib libvect.a
```

Certains systèmes construisent automatiquement la table des matières lors de l'ajout d'éléments avec `ar`. Pour garantir la portabilité des Makefiles, il vaut mieux néanmoins prévoir une règle en utilisant une variable comme nom de commande pour construire la table des matières et définir cette variable `RANLIB` comme `:` (ne rien faire) pour les systèmes où cette opération est inutile.

II.9.2 Règles implicites pour bibliothèques

Afin de permettre l'écriture de règles implicites pour la production des bibliothèques, quand `make` cherche à faire un élément d'un bibliothèque, il recherche des règles avec `(element)` comme cible en plus de celles avec `bibliothèque(element)`.

Cela permet d'écrire une règle implicite en utilisant « `(%)` » comme cible qui va s'associer avec n'importe quel fichier dans n'importe quelle bibliothèque. Dans ce contexte, une nouvelle variable automatique est définie et la signification de `$@` est légèrement modifiée :

`$%`	le nom du fichier objet dans une bibliothèque lorsque la cible est un élément de bibliothèque. Par exemple si la cible est `libvect.a(add.o)`, `$%` est `add.o`.
`$@`	Si la cible est un élément de bibliothèque, `$@` est le nom de la bibliothèque, au lieu du nom complet de la cible. Dans l'exemple précédent, `$@` est `libvect.a`.

Avec l'aide de ces nouvelles variables, on peut maintenant écrire une règle implicite pour la gestion des bibliothèques en C :

```
(%.o): %.c
        $(CC) -c -o $% $(CPPFLAGS) $(CFLAGS) $<
        $(AR) cr $@ $%
        rm $%
```

Il est nécessaire de la compléter par une règle qui définit la liste des éléments de la bibliothèque et le nom de la bibliothèque et qui construit la table des matières :

```
$(BIB): $(ELTS)
        $(RANLIB) $@
```

II.10 Appel de make

Nous avons vu tout ce qu'il faut pour écrire un fichier Makefile et utiliser `make`. Voici quelques rappels :

- par défaut `make` utilise le fichier appelé `Makefile` ou `makefile`. L'option `-f` permet de spécifier un autre fichier.
- `make` essaye de construire la cible de la première règle explicite trouvée dans le Makefile. On peut spécifier une autre cible explicitement sur la ligne de commande. Par exemple : « `make calcul` ».
- `make` hérite les variables de l'environnement. On peut forcer la redéfinition d'un variable sur la ligne de commande.
- l'option `-k` ordonne à `make` de ne pas s'arrêter complètement lorsqu'il rencontre une erreur. Il essaye de construire les autres cibles indépendantes de celle qui a causé l'erreur.
- l'option `-s` supprime complètement l'affichage des commandes avant exécution.
- l'option `-n` supprime l'exécution des commandes. `make` affiche les commandes qu'il exécuterait si l'option `-n` n'était pas présente.

Chaque variante de `make` dispose de différentes options pour faciliter la mise au point des Makefiles. Malheureusement, elles ne sont pas standard.

II.11 Conventions

`Make` est très utilisé pour automatiser la production de programmes dans le monde Unix. C'est un outil puissant et très général. Même trop général aux yeux de certains.

Grâce à l'existence de gros projets logiciels dont les sources sont disponibles librement, un usage standard de `make` s'est répandu dans la communauté Unix. Le respect de ces usages facilite l'écriture de nouveaux Makefiles et la lecture des Makefiles des autres.

II.11.1 Utilisation des variables

Éviter au maximum d'utiliser directement des noms de commandes dans les commandes des Makefiles, mais plutot des variables qui permettent de paraméter facilement les Makefiles. Donner à ces variables le nom en majuscule de l'utilitaire standard correspondant. Exemples : `$(CC), $(AR), $(CP), $(INSTALL),`...

Utiliser également des variables pour paraméter les options. En général, le nom de la variable s'obtient en ajoutant `FLAGS` derrière le nom de la commande (par exemple `CPPFLAGS`).

Enfin utiliser des noms de variable pour désigner les répertoires extérieurs utilisés par votre Makefile. En général, on suffixe leur nom par `DIR`: `$(LIBDIR), $(BINDIR)`...

II.11.2 Cibles standard

L'usage veut que l'on définisse un certain nombre de cibles spéciales qui ne correspondent à aucun fichier qui sera créé, mais qui permettent de désigner simplement ce que l'on veut faire. Les cibles suivantes sont couramment employées :

all: toujours placée comme première règle explicite, pour définir ce qui est fait par défaut. Équivalent à la déclaraction `program` en Pascal ou Fortran ou à la fonction `main` du C. Ce n'est pas une contrainte de `make` mais un usage bien établi.

install: installation du programme construit dans le système. Copie les différents fichiers produits à l'endroit à partir duquel ils seront utilisés.

uninstall: défait ce que `install` a fait. Trop peu de Makefiles implémentent cette option.

clean: supprime du répertoire courant tous les fichiers produits par `make all`. Peut laisser certains fichiers sources qui sont produits automatiquement lors de la configuration.

distclean: remet le répertoire source exactement dans l'état où il est distribué par le fournisseur, en supprimant en général tous les fichiers qui sont produits par la configuration.

II.11.3 Une seule cible par répertoire

Ce n'est pas une contrainte de `make`, mais cela facilite beaucoup l'écriture et la maintenance des logiciels. Lorsqu'un projet est composé de plusieurs éléments (exécutables, bibliothèques, documentation), organiser l'ensemble en sous-répertoires de façon à ce qu'il n'y ait qu'une cible à faire par répertoire permet de standardiser les Makefiles en réutilisant les mêmes variables et les mêmes rêgles.

Un Makefile général qui va exécuter récursivement `make` dans chaque sous-répertoire est facile à écrire à l'aide des cibles standard introduites ci-dessus.

II.12 Utilisation avancée

Voici quelques éléments de l'utilisation de `make` qui n'ont pas été abordés, mais qui peuvent s'avérer utiles pour l'écriture de makefiles efficaces.

II.12.1 Substitution de variables

La plupart des versions modernes de `make` sont capables de remplacer une partie du texte de la valeur des variables lors de leur expansion. Cela se réalise avec la syntaxe suivante :

`$(`*VARIABLE* `:` *OLD*`=`*NEW*`)`

Variable est le nom de la variable référencée, *old* est le texte à supprimer de la valeur de la variable et *new* est le texte de remplacement.

Exemple :

```
SRC = toto.c
OBJ = $(SRC:.c=.o)
```

définit une variable `OBJ` qui vaut `toto.o` en remplaçant `.c` par `.o` dans la valeur de `SRC`.

Ce mécanisme est plus puissant que vous ne l'imaginez. En effet, *old* peut contenir un caractère `%` qui agit comme filtre comme dans le cas des règles implicites. Et si la partie *new* contient elle aussi un `%`, il sera remplacé par le radical attrapé par le filtre dans *old*.

Exemple :

```
SRCS = a.c b.c c.c
OBJS = $(SRCS:%.c=%.o)
```

définit la variable `OBJS=a.o b.o c.o`!

On peut utiliser le `%` n'importe où dans la substitution :

```
BIB = bib.a
SRCS = a.c b.c c.c
ELTS = $(SRCS:%.c=$(BIB)(%.o))
```

définit la liste des éléments à ranger dans une bibliothèque : `bib.a(a.o) bib.a(b.o) bib.a(c.o)`.

II.12.2 Appel récursif de make

Comme on l'a vu plus haut, il peut être utile d'appeler récursivement `make`, entre autres pour compiler un projet qui s'étend sur plusieurs répertoires. Appeler récursivement `make`, c'est utiliser une commande dans une règle qui appelle `make`.

Pour cela il ne faut pas utiliser `make` directement, mais toujours utiliser la variable `$(MAKE)`. La valeur de cette variable est le nom exact de la commande utilisée pour appeler `make` avec les options qui lui ont été passées.

Un deuxième bénéfice de l'utilisation de `$(MAKE)` est que l'option `-n` (et les options `-t` et `-q` dont il n'a pas été question) fait bien ce que l'on attend : lorsque `make` rencontre `$(MAKE)` dans les commandes d'un Makefile il exécute cette commande, malgré l'option `-n` qui demande à make de seulement afficher ce qui serait fait. Ainsi on peut suivre toute une arborescence de Makefiles.

Exemple :

```
SUBDIRS = a b c d
all:
	@for i in $(SUBDIRS) ; \
```

```
do ; \
  ( cd $$i; \
    echo "making all in $$i"; \
    $(MAKE) all \
  ) || exit 2; \
done
```

Cette règle utilise une boucle du shell pour exécuter `$(MAKE)` dans chaque sous-répertoire. Pour faire marcher cette construction, il faut surveiller de nombreux points :

- utiliser `@` pour supprimer l'affichage de la commande,
- utiliser des `\` pour maintenir la commande shell sur une seule ligne,
- utiliser `$$` pour insérer un dollar dans la ligne de commande du shell,
- séparer les instructions de la commande shell par des « `;` »,
- créer un sous-shell avec les parenthèses pour exécuter l'appel récursif de `make` dans le bon sous-répertoire sans changer le répertoire courant du shell principal,
- forcer une erreur dans le shell si une erreur est détectée par l'appel récursif de `make`. En effet par défaut la boucle `do` ne s'arrête pas sur une erreur.

II.12.3 Plusieurs règles pour une cible

Il existe un type de règles utilisant un double-double point (`::`) au lieu d'un simple après le nom de la cible. Elles permettent de définir plusieurs règles indépendantes pour une cible.

Si plusieurs règles apparaissent pour une cible, elles doivent toutes être du même type : double double-point ou simple double-point. Nous avons vu plus haut que s'il y a plusieurs règles avec simple double-point, seule la première peut contenir des commandes. Les autres doivent être simplement des règles de dépendance, qui sont concaténées aux dépendances de la règle principale.

Les règles à `::` n'ont d'intérêt que s'il y en a plusieurs pour la même cible. Chaque règle est alors traitée à part, indépendamment des autres. L'ordre dans lequel les règles sont traitées ne devrait pas avoir d'importance.

Ces règles sont utiles lorsqu'il existe plusieurs manières différentes de produire un fichier selon la dépendance qui provoque la mise à jour ou selon des paramètres externes qui se présentent en fonction du système.

Le fonctionnement exact des règles à `::` est plutôt complexe et les occasions où elles sont la seule solution à un problème sont rares, il vaut donc mieux les éviter.

II.13 Production automatique des dépendances

Il reste un dernier point à soulever sur l'utilisation de `make` ; à l'usage on se rend très vite compte de trois choses :

- les dépendances sont difficiles à écrire manuellement. C'est répétitif et volumineux,
- si les dépendances ne sont pas à jour, `make` devient quasiment inutile car on ne peut plus lui faire confiance,
- un makefile bien écrit sépare les règles contenant des commandes (qui sont presque toujours implicites) des règles qui expriment les dépendances.

Lorsque les règles de make concernent essentiellement la compilation de programmes en langage C ou d'autes langages voisins, l'essentiel des dépendances est constitué des règles de dépendance des fichiers objet sur les fichiers source et les différents fichiers d'en-tête inclus par les fichiers source.

L'idée vient alors rapidement de produire ces règles de dépendances automatiquement, ce qui décharge l'utilisateur de `make` d'une tâche lourde et peu passionnante et rend l'utilisation de make plus attrayante.

Plusieurs approches sont possibles et ont été utilisées par les différentes variantes de make. Malheureusement, aucune n'est vraiment standardisée. Néanmoins, il est indispensable de mettre en place un tel outil sur un projet dès qu'il dépasse une certaine taille. Les erreurs dues à une mauvaise synchronisation des sources et des objets sont en effet très difficiles à détecter et très coûteuses à prévenir si la seule façon d'être sûr que tous les objets sont à jour est de tout recompiler. (La compilation de toute l'arborescence X11R6 dure entre deux et quatre heures selon les options sur une SparcStation).

Voici quelques approches possibles :

– Sun Microsystems a proposé une approche utilisant la cible factice `.KEEP_STATE:`. Lorsque cette cible est présente, le `make` de Sun positionne une variable d'environnement qui fait produire au préprocesseur C de Sun un fichier `.make.state` qui contient les règles de dépendances déduites des inclusions de fichier réalisées. Si ce fichier est présent au démarrage de `make`, il est lu.
  Cette approche a l'avantage d'être très facile à mettre en œuvre. Elle a deux inconvénients : elle est spécifique à la chaîne de développement Sun et elle fonctionne avec un temps de retard. Il faut appeler deux fois make pour s'assurer que tout est à jour.
– Le compilateur GNU gcc propose l'option `-MD` qui permet de stocker dans un fichier `.d` les dépendances découvertes durant la compilation. Le principe est voisin de celui de Sun. L'outil make de GNU permet d'include tous les fichiers .d dans le makefile par une seule commande (`include *.d`). Pour fonctionner avec des bibliothèques ou des fichiers objets produits ailleurs que dans le répertoire courant, cette méthode a besoin d'outils supplémentaires qui ne sont pas intégrés dans GNU make ou gcc.
– BSD 4.4, le dialecte d'Unix développé à l'université de Berkeley a développé une variante de make qui intègre un outil de production des dépendances conçu pour fonctionner avec les règles implicites pré-installées. Comme ces règles sont présentes dans des fichiers de configuration modifiables, ce système est adaptable à d'autres environnements que celui de la compilation du système BSD 4.4.
  Le principal défaut de cette approche est que le dialecte de make employé s'éloigne des makes les plus répandus (SYSV et GNU), mais il a été porté à de nombreuses plateformes.
– le Consortium X a développé un outil de production de dépendances destiné à être utilisé avec le système de production de makefiles `imake` utilisé pour la compilation des sources de X11. Cet outil manque de souplesse pour être utilisé de manière plus générale, et même il ne remplit pas complètement son rôle pour assurer la cohérence des compilations de X11.
– au LAAS a été développé un outil qui interagit avec les makes compatibles POSIX. Il repose sur l'appel du pré-processeur pour déterminer les inclusions de fichier. Il dispose d'un certain nombre d'options qui en font à ma connaissance (biaisée puisque j'ai participé activement à son développement) l'un des meilleurs outils de cette catégorie. En particulier, il peut gérer la production de dépendances pour des bibliothèques ou des fichier objets produits n'importe où, il peut instancier « à l'envers » des variables pour produire

des dépendances portables, il peut construire les dépendances de manière incrémentales, pour ne mettre à jour que les dépendances des fichiers modifiés.

Index