Compilation séparée, Architecture logicielle

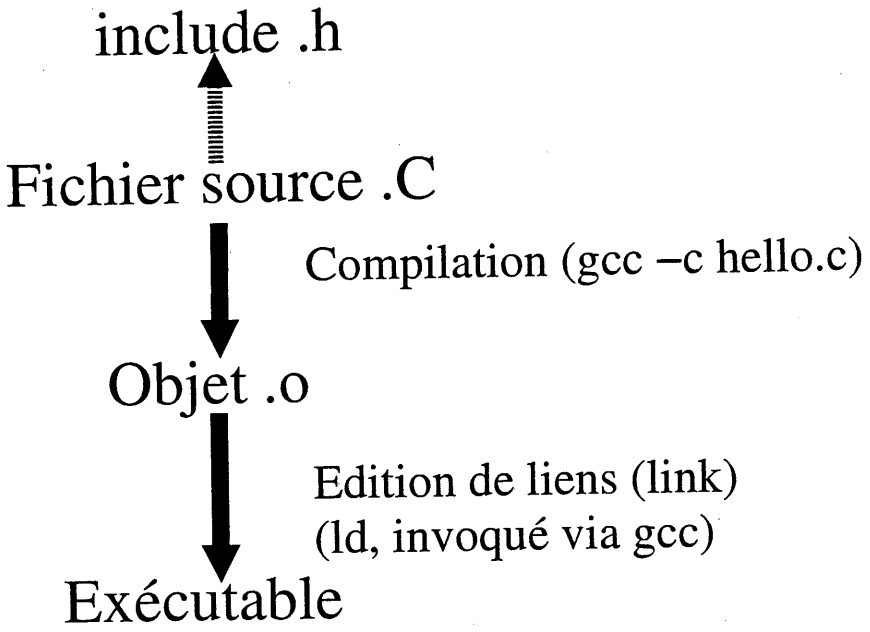
...

Ou l'art de décomposer un projet en programmes et en fichiers

Problématique

- Gérer des projets de taille importante
- Exemple: un noyau unix (linux 2.4.2)
  - 3352 fichiers .c (2.5 millions lignes)
  - 4010 fichiers .h (650 000 lignes)
  - Produit 1 exécutable (714 ko) + modules (.o)
- Favoriser la réutilisabilité du code (plus lisible, plus modulaire)
- Limiter les temps de compilation

Du source à l'exécutable (Cas trivial)



Qu'est ce qu'un .o?

- .o ou .obj, objets
- Ne peut être exécuté directement par la machine
- Les fonctions sont présentes sous forme de langage machine
- Le nom des symboles (fonctions, variables) est préservé

Ce qu'il y a dans un .o

```
#include <stdio.h>

int varGlobale;

void fonction(int j)
{
   j=j;
}

int main()
{
  int i=6;
  fonction(i);
  return 0;
}
```

- Compilation par gcc –c hello.c
- Commande Unix « nm hello.o» pour visualiser la liste des symboles

```
bash$ gcc -c hello.c
bash$ nm hello.o
Symbols from hello.o:

Name     Value  Scope  Type Subspace

$CODE$      |0|static|code    |$CODE$
__main      | |undef |code    |
fonction    |0|extern|entry   |$CODE$
main        |0|extern|entry   |$CODE$
varGlobale |4|undef |common  |$CODE$
```

Organisation des fichiers (rappel)

- Dans les includes (.h) ou header
  - Protection contre les inclusions multiples (`#ifndef`)
  - Prototype des fonctions (`void fonction(int);`)
  - Prototypes des variables globales (`extern int varGlobale`)
  - Types (`struct cplx { double x; double y;}`)
- Dans les sources (.c)
  - Corps des fonctions
  - Déclarations des variables globales (`int varGlobale;`)

Conseils de programmation C

- Mettre dans les include le minimum de chose:
  - Uniquement ce qui est mis à disposition des autres programmeurs
  - Uniquement des prototypes et définitions
  - Inclure uniquement si nécessaire (dans le .C si possible).
  - Protection contre les inclusions multiples (#ifndef)

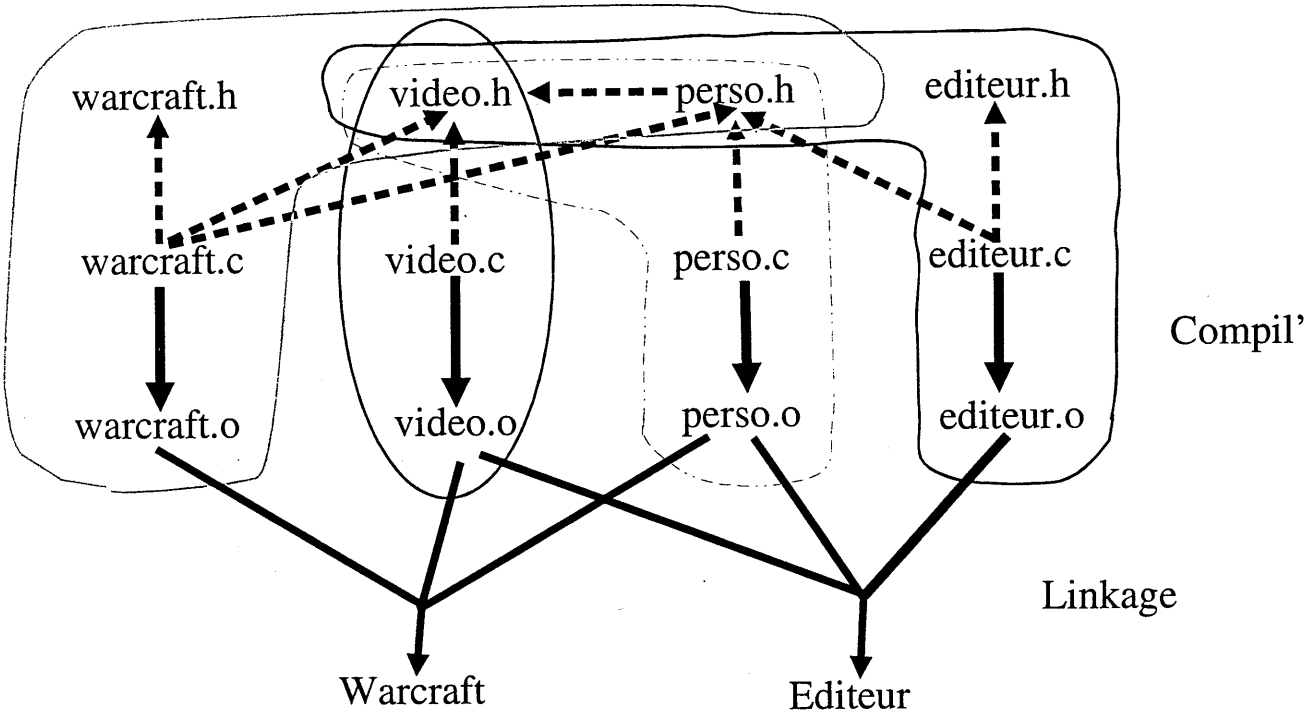
Ce que fait un compilateur

- Vérification de la **syntaxe** (Accolades, mots–clefs, respect de la grammaire)
- Vérification de la **correspondance des types**
- Donc besoin de
  - Prototypes des fonctions,
  - Déclarations de type,
  - Type des variables
- Mais pas du corps des fonctions appelées.
- Génération de code machine pour les fonctions définies localement

Exemple (peu) pratique

- Voir les fichiers sur vos polycopiés
- Programmes principaux:
  - warcraft.c warcraft.h: corps du jeu (contient une fonction 'main()')
  - editeur.c editeur.h: corps de l'éditeur de niveau (contient une fonction 'main()')
- Utilitaires:
  - video.c video.h: ensemble de routines d'affichage
  - perso.c perso.h: gestion des personnages du jeu

Structure des programmes: Cas général



Apr 18, 01 21:00 **warcraft.h** Page 1/1

```
#ifndef _warcraft_h_
#define _warcraft_h_

#endif
```

Apr 19, 01 16:02 **warcraft.c** Page 1/1

```
#include <stdio.h>


#include "video.h"
#include "warcraft.h"
#include "perso.h"

int main()
{
  Perso p;


  p=initPerso("Dragon");
  affiche(p.img);
  printf("Adresse carte video: %p\n",adresseCarteVideo);

  return 0;
}
```

Apr 18, 01 19:29 **editeur.h** Page 1/1

```
#ifndef _editeur_h_
#define _editeur_h_

#endif
```

Apr 18, 01 19:19 **editeur.c** Page 1/1

```
#include <stdio.h>

#include "editeur.h"
#include "video.h"
#include "perso.h"

int main()
{
  puts("j'edite \n");
  return 0;
}
```

Apr 19, 01 16:02 **video.h** Page 1/1

```
#ifndef _video_h_
#define _video_h_

extern char * adresseCarteVideo;
/* Ceci ne suffit pas à réserver l'espace pour
   la variable adresseCarteVideo */

/* Image 8*8 en true color (32 bits) */
typedef int Image[8][8];

void affiche(Image);

#endif
```

Apr 19, 01 16:02 **video.c** Page 1/1

```
#include <stdio.h>
#include "video.h"

char * adresseCarteVideo=NULL;;

void affiche(Image i)
{
  printf("J'affiche\n");
}
```

Apr 18, 01 19:31 **perso.h** Page 1/1

```
#ifndef _perso_h_
#define _perso_h_

#include "video.h" /* Pour type Image */

#define MAX_NAME 50 /* Taille maximale d'un nom */

/* Un personnage du jeu a un nom et une
   représentation graphique  */
typedef struct {
  Image img;
  char nom[MAX_NAME];
}  Perso;

/* Renvoie un Perso initialisé */
Perso initPerso(char *);

#endif
```

Apr 18, 01 19:13 **perso.c** Page 1/1

```
#include "perso.h"

#include <string.h> /* Pour strcpy */

Perso initPerso(char *nom)
{
  Perso tmp;

  strcmp(tmp.nom,nom);
  return tmp;
}
```

Notion de dépendances

- Liste des dépendances
- A dépend de B $\Leftrightarrow$ Si A est modifié, alors il faut refaire (recalculer) B
- Version linéaire du graphe!
- Lourd à gérer
- Mais permet de minimiser les temps de compilation
- Que compiler, dans quel ordre?

Dépendances

- Que dois–je faire (recompiler, lier) si je modifie les fichiers suivants:
  - perso.h?
  - video.c?
  - warcraft.c?
  - video.h?
  - editeur.h?

Le Makefile

- But: Recompiler le strict nécessaire, uniquement quand c'est nécessaire!
- Dépendances
  - `warcraft.o: warcraft.c warcraft.h video.h perso.h`
- Règles
  - `gcc -c warcraft.c -o warcraft.o`
- Syntaxe générale
  - `cible: dépendances`

```
règle 1
règle 2
```

Un Makefile basique

```
#Compilations
perso.o: perso.c perso.h video.h
   gcc -c perso.c -o perso.o
video.o: video.c video.h
   gcc -c video.c -o video.o
warcraft.o: warcraft.c warcraft.h video.h perso.h
   gcc -c warcraft.c -o warcraf.o
editeur.o: editeur.c editeur.h video.h perso.h
   gcc -c editeur.c -o editeur.o

#Editions de lien
warcraft: warcraft.o video.o perso.o
   gcc  warcraft.o video.o perso.o -o warcraft
editeur: editeur.o video.o perso.o
   gcc editeur.o video.o perso.o -o editeur
```

Makefile: Astuces

- Raccourcis:
  - `$@` : nom de la cible
  - `$<` : première dépendance
  - `$^` : toutes les dépendances
  - `$?` : les dépendances plus récentes que la cible
- Variables (comme en shell):
  - `COMPILATEUR = gcc` (affectation)
  - `$(COMPILATEUR) -c warcraft.c`
- Les cibles ne sont pas forcément des fichiers

Makefile: Exemple

```
CC =  gcc

all: warcraft editeur

perso.o: perso.c perso.h video.h
   $(CC) -c $< -o $@
video.o: video.c video.h
   $(CC) -c $< -o $@
warcraft.o: warcraft.c warcraft.h video.h perso.h
   $(CC) -c $< -o $@
editeur.o: editeur.c editeur.h video.h perso.h
   $(CC) -c $< -o $@
warcraft: warcraft.o video.o perso.o
   $(CC) $^ -o $@
editeur: editeur.o video.o perso.o
   $(CC) $^ -o $@
clean:
   rm -f *.o editeur warcraft
```

Utilisation du Makefile

- Appel par l'utilitaire Make:
  - `make all:` construit tous les programmes
  - `make:` construit la première cible rencontrée
  - `make clean:` appelle la règle qui supprime les « .o » et les exécutables
  - `make warcraft.o:` construit l'objet warcraft.o
- Pour les curieux: « man makedepend »...
  - permet de générer automatiquement la liste des dépendances

Règles génériques (Patterns)

```
CC=   gcc
CFLAGS  =   -Wall

all: warcraft editeur

%.o: %.c %.h
    $(CC) $(CFLAGS) -c $< -o $@
%: %.o
    $(CC) $^ -o $@

perso.o: perso.c perso.h video.h
video.o: video.c video.h
warcraft.o: warcraft.c warcraft.h video.h perso.h
editeur.o: editeur.c editeur.h video.h perso.h
warcraft: warcraft.o video.o perso.o
editeur: editeur.o video.o perso.o
```

Variables globales

- Usage **exceptionnel**!
- Si doit être exportée:
  - `extern int varGlobale` dans le header (n'alloue pas la variable mais la définit)
  - `int varGlobale` dans le .C
- Si elle est interne au programme:
  - Uniquement dans le .C
  - `static int varGlobale;`
  - Cache la variable au reste du monde

Makefile: usages surprenants

```
archive: Makefile *.c *.h
    tar cf sauvegarde.tar *.c *.h Makefile
    gzip sauvegarde.tar

depend:
    makedepend -- $(CFLAGS) -- $(SRCS)
```

```
CC      =       gcc

all: warcraft editeur

perso.o: perso.c perso.h video.h
        $(CC) -c $< -o $@
video.o: video.c video.h
        $(CC) -c $< -o $@
warcraft.o: warcraft.c warcraft.h video.h perso.h
        $(CC) -c $< -o $@
editeur.o: editeur.c editeur.h video.h perso.h
        $(CC) -c $< -o $@
warcraft: warcraft.o video.o perso.o
        $(CC) $^ -o $@
editeur: editeur.o video.o perso.o
        $(CC) $^ -o $@
clean:
        rm -f *.o editeur warcraft
```



```
CC      =       gcc
CFLAGS  =       -Wall

all: warcraft editeur

%.o: %.c %.h
        $(CC) $(CFLAGS) -c $< -o $@
%: %.o
        $(CC) $^ -o $@

clean:
        rm -f *.o editeur warcraft

perso.o: perso.c perso.h video.h
video.o: video.c video.h
warcraft.o: warcraft.c warcraft.h video.h perso.h
editeur.o: editeur.c editeur.h video.h perso.h
warcraft: warcraft.o video.o perso.o
editeur: editeur.o video.o perso.o
```



```
CC      =       gcc
CFLAGS  =       -Wall

all: warcraft editeur

%.o: %.c %.h
        $(CC) $(CFLAGS) -c $< -o $@
%: %.o
        $(CC) $^ -o $@

clean:
        rm -f *.o editeur warcraft

# Dependances
# DO NOT DELETE

editeur.o: /usr/include/stdio.h /usr/include/features.h
editeur.o: /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h
editeur.o: /usr/lib/gcc-lib/i386-linux/2.95.2/include/stddef.h
editeur.o: /usr/lib/gcc-lib/i386-linux/2.95.2/include/stdarg.h
editeur.o: /usr/include/bits/types.h /usr/include/libio.h
editeur.o: /usr/include/_G_config.h /usr/include/bits/stdio_lim.h editeur.h
editeur.o: video.h perso.h
perso.o: perso.h video.h /usr/include/string.h /usr/include/features.h
perso.o: /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h
perso.o: /usr/lib/gcc-lib/i386-linux/2.95.2/include/stddef.h
video.o: /usr/include/stdio.h /usr/include/features.h
video.o: /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h
video.o: /usr/lib/gcc-lib/i386-linux/2.95.2/include/stddef.h
```



```
video.o: /usr/lib/gcc-lib/i386-linux/2.95.2/include/stdarg.h
video.o: /usr/include/bits/types.h /usr/include/libio.h
video.o: /usr/include/_G_config.h /usr/include/bits/stdio_lim.h video.h
warcraft.o: /usr/include/stdio.h /usr/include/features.h
warcraft.o: /usr/include/sys/cdefs.h /usr/include/gnu/stubs.h
warcraft.o: /usr/lib/gcc-lib/i386-linux/2.95.2/include/stddef.h
warcraft.o: /usr/lib/gcc-lib/i386-linux/2.95.2/include/stdarg.h
warcraft.o: /usr/include/bits/types.h /usr/include/libio.h
warcraft.o: /usr/include/_G_config.h /usr/include/bits/stdio_lim.h video.h
warcraft.o: warcraft.h perso.h
perso.o: video.h
```

Compilation séparée et Make

Benjamin Drieu (drieu@bocal.cs.univ-paris8.fr) v1.0 - 30 Avril 1997

Cette documentation explique comment utiliser make pour la compilation séparée: comment écrire un fichier Makefile, l'utiliser, et quelles sont les conventions utilisées dans les fichiers Makefile. La version de make à laquelle je ferais référence au cours de ce document est Gnu Make.

Contents

1 Introduction

1.1 Comment marche la compilation

```
        *.c  *.h (Fichiers source)
                    |
                COMPILATEUR
                    |
              *.o (Fichier objet)
                    |
Librairies -->  LIEUR <-- Fichiers objet
                    |
            Fichier executable
```

1.2 Compilation séparée

1.2.1 Utilité

L'utilité est triple:

- La programmation est modulaire, donc plus compréhensible
- La séparation en plusieurs fichiers produit des listings plus lisibles
- La maintenance est plus facile car seule une partie du code est recompilée

1.2.2 Ce que fait *make*

- *make* assure la compilation séparée grâce à gcc
- *make* utilise des macro-commandes et des variables
- *make* permet de ne recompiler que le code modifié
- *make* permet d'utiliser des commandes *shell*, et ainsi d'effectuer une installation

Make est essentiel lorsque l'on veut effectuer un portage, car la plupart des logiciels libres UNIX (c'est-à-dire des logiciels qui sont fournis avec le code source) l'utilisent pour leur installation.

1.3 Makefile ?

Le fichier `Makefile` est un fichier nécessaire à *make*. Un fichier `Makefile` indique à *make* comment exécuter les instructions nécessaires à l'installation d'un logiciel ou d'une librairie.

Le fichier `Makefile` doit se trouver dans le répertoire courant lorsqu'on appelle *make* à l'invite du shell.

Les instructions contenues dans un fichier `Makefile` obéissent à une syntaxe particulière un peu stupide.

2 Règles

Les fichiers `Makefile` sont structurés grâce aux *règles*. Ce sont elles qui définissent ce qui doit être exécuté ou non, et qui permettent de compiler un programme de différentes façons.

2.1 Qu'est-ce qu'une règle ?

Une *règle* est une suite d'instructions qui seront exécutées pour construire une *cible*, mais uniquement si des *dépendances* sont plus récentes.

La syntaxe d'une règle est la suivante:

```
cible: dependances
        commandes
        ...
```

2.2 Cible

La cible est généralement le nom d'un fichier qui va être généré par les commandes qui vont suivre, ou une action gérée par ces mêmes commandes, par exemple `clean` ou `install` (Voir chapitre IX [Cibles] pour plus de détails sur les conventions utilisées dans l'attribution d'un nom à une règle).

2.3 Dépendances

Les dépendances sont les fichiers ou les règles nécessaires à la création de la cible. Par exemple un fichier en-tête ou un fichier source dans le cas d'une compilation C. Dans le cas d'un fichier, la cible n'est construite que si ce fichier est plus récent que la cible.

2.4 Commandes

C'est une suite de commandes shell qui seront exécutées au moment de la création de la cible. Une étrangeté de la syntaxe des fichiers `Makefile` oblige l'utilisateur de *make* à insérer une tabulation au début de chaque ligne, faute de quoi *make* affichera une erreur au moment de son exécution.

La syntaxe de *make* oblige aussi l'utilisateur à ajouter une caractère "*backslash*" ('\') à la fin de chaque ligne dès que les commandes à exécuter dépassent une ligne de texte.

3 Mon premier Makefile !

Maintenant que nous connaissons la syntaxe d'un fichier `Makefile`, nous allons en créer un pour apprendre à les utiliser.

3.1 Fichier exemple Makefile

```
# Mon premier Makefile

all: foobar.o main.c
        gcc -o main foobar.o main.c

foobar.o: foobar.c foobar.h
        gcc -c foobar.c -o foobar.o
```

3.2 Et maintenant ?

Si vous avez enregistré l'exemple ci-dessus dans un fichier `Makefile`, il ne nous reste plus qu'a exécuter *make* dans le même répertoire que celui où vous avez enregistré le fichier.

Make s'exécute tout simplement en lançant la commande:

```
$ make all
```

Make va alors interpréter le fichier `Makefile` et exécuter les commandes contenues dans la règle ***all***, une fois que les dépendances *foobar.o* et *main.c* seront vérifiées.

C'est à dire dire que si `foobar.o` ou `main.c` sont plus récents que le fichier `main`, *make* recompilera `main`.

Notez que si j'avais simplement tapé: `"make"` le résultat serait le même car quand *make* est exécuté sans argument, *make* exécute la première règle recontrée.

4 Macro-commandes et variables

Les habitué(e)s de la programmation C ne seront pas dépaysé(e)s par le concept des variables de *make*. En fait, il faut plutôt considérer les variables comme des macro-commandes (`#define` en C).

4.1 Déclaration

La déclaration se fait tout simplement avec la syntaxe ci-dessous:

```
NOM = VALEUR
```

Les espaces insérés ici ne sont pas obligatoires, mais facilitent la lisibilité du `Makefile`. La valeur affectée à la variable peut comme pour les macro-commandes du C comporter n'importe quels caractères, elle peut aussi être une autre variable.

4.2 Appel

La syntaxe de l'appel de la macro-commande est la suivante:

```
$(NOM)
```

4.3 Exemple

```
prefix = /usr/local
bindir = $(prefix)/bin
```

5 Un Makefile un peu plus complexe et commenté

```
# $(BIN) est la nom du binaire généré
BIN = foo

# $(OBJECTS) sont les objets qui seront générés après la compilation
OBJECTS = main.o foo.o

# $(CC) est le compilateur utilisé
CC = gcc

# all est la première règle à être exécutée car elle est la première
# dans le fichier Makefile. Notons que les dépendances peuvent être
# remplacées par une variable, ainsi que n'importe quel chaine de
# caractères des commandes
all: $(OBJECTS)
	$(CC) $(OBJECTS) -o $(BIN)

# ensuite les autres règles
main.o: main.c main.h
	$(CC) -c mainc
foo.o: foo.c foo.h main.h
	$(CC) -c foo.c
```

6 Caractères jokers

Les caractères jokers s'utilisent comme sous *shell*. Les caractères valides sont `* ? [...]`. Par exemple, `toto?.c` représente tous les fichiers commençant par `toto`, finissant par `.c`, avec une lettre entre ces deux chaînes.

Comme sous shell, le caractère '\' permet d'inhiber l'action des caractères jokers. Par exemple `'sr.c'` fait référence à `st*r.c` et non pas à tous les fichiers commençant par `st` et finissant par `r.c`.

Exemples:

```
clean:
	# ici le caractère joker * est géré par le shell et non pas
	# par make
	rm -f *.o

print: *.c
	# le $? est une variable automatique (Voir chapitre VII
	# [Patterns et variables automatiques])
	lpr -Php $?
```


7 Patterns et variables automatiques

7.1 Pattern

Un pattern s'utilise un peu comme un caractère joker, mais uniquement dans le cas des cibles. Le caractère faisant office de joker est le caractère pourcent ('%').

L'intérêt est d'avoir plusieurs cibles. Les patterns permettent de filtrer les cibles pour savoir pour lesquelles d'entre elles les commandes qui suivent seront exécutées.

7.2 Exemple idiot

Pourquoi cet exemple est-il idiot ? Tout simplement parcequ'on met pèle-mêle tous les fichiers à compiler dans la même variable `OBJECTS`. Le plus simple aurait été d'utiliser plusieurs variables et de faire des règles différentes pour chaque type de compilation (normale, x11, athena).

```
# Notons que les objets sont à la fois des fichiers objets à lier et
# des exécutables à compiler
OBJECTS = text/main.o text/foo.o x11/main x11/bar athena/main
# Le compilateur est bien sûr gcc
CC = gcc

# Les dépendances sont tous les objets
all: $(OBJECTS)
	# On va quand même faire quelque chose dans cette règle
	echo DONE

# Ici les règles pour tous les fichiers .o se trouvant dans le
# répertoire 'text'
text/%.o: text/%.c
	$(CC) -c text/$*.c

# Ici les règles pour tous les fichiers se trouvant dans le répertoire
# 'x11'
x11/%: x11/%.c
	$(CC) -lX11 x11/$*.c

# Ici les règles pour les fichiers se trouvant dans le répertoire
# 'athena'
athena/%: athena/%.c
	$(CC) -lXaw -lXext -lXmu -lXt -lX11 athena/$*.c
```

7.3 Liste des variables automatiques

Les variables automatiques sont des variables qui sont actualisées au moment de l'exécution de chaque règle, en fonction de la cible et des dépendances.

- `$@`: nom de la cible
- `$<`: première dépendance de la liste des dépendances

- `$?`: les dépendances plus récentes que la cible
- `$^`: toutes les dépendances
- `$*`: dans le cas de l'utilisation des patterns, la chaine correspondant au %

8 Conventions d'appellation

8.1 Noms d'exécutables et d'arguments

Entre parenthèses les valeurs par défaut

- AR: programme de maintenance d'archive (*ar*)
- CC: compilateur C (*cc*)
- CXX: compilateur C++ (*c++*)
- RM: commande pour effacer un fichier (*rm*)
- TEX: programme pour créer un fichier TeX dvi à partir d'un source TeX (*tex*)
- ARFLAGS: paramètres à passer au programme de maintenance d'archives ()
- CFLAGS: paramètres à passer au compilateur C ()
- CXXFLAGS: paramètres à passer au compilateur C++ ()

8.2 Noms de répertoires de destination

Entre parenthèses les valeurs usuelles

- prefix: racine du répertoire d'installation (`/usr/local`)
- exec_prefix: racine pour les binaires (`$(prefix)`)
- bindir: répertoire d'installation des binaires (`$(exec_prefix)/bin`)
- libdir: répertoire d'installation des librairies (`$(exec_prefix)/lib`)
- datadir: répertoire d'installation des données statiques pour le programme (`$(exec_prefix)/lib`)
- statedir: répertoire d'installation des données modifiables par le programme (`$(prefix)/lib`)
- includedir: répertoire d'installation des en-têtes (`$(prefix)/include`)
- mandir: répertoire d'installation des fichiers de manuel (`$(prefix)/man`)
- manxdir: répertoire d'installation des fichiers de la section *x* du manuel (`$(prefix)/manx`)
- infodir: répertoire d'installation des fichiers info (`$(prefix)/info`)
- srcdir: répertoire d'installation des fichiers source (`$(prefix)/src`)

8.3 Noms de cibles

Un utilisateur de *make* peut donner à ses cibles le nom qu'il désire. Mais pour des raisons de lisibilité, on donne toujours un nom standard à ses cibles selon leur comportement.

Quelques exemples de cibles standard:

- `all`: compile tous les fichiers source pour créer l'exécutable principal
- `install`: exécute *all*, et copie l'exécutable, les librairies, les datas, et les fichiers en-tête s'il y en a dans les répertoires de destination
- `uninstall`: détruit les fichiers créés lors de l'installation, mais pas les fichiers du répertoire d'installation (où se trouvent les fichiers source et le `Makefile`)
- `clean`: détruit tout les fichiers créés par *all*
- `info`: génère un fichier *info*
- `dvi`: génère un fichier *dvi*
- `dist`: crée un fichier *tar* de distribution

9 Copyright

XEmacs Reference Card

(for version 20.5+)

Starting Emacs

To enter XEmacs, just type its name: `xemacs`

To read in a file to edit, see Files, below.

Leaving Emacs

suspend Emacs (or iconify frame under X)	`C-z`
exit Emacs permanently	`C-x C-c`

Files

read a file into Emacs	`C-x C-f`
save a file back to disk	`C-x C-s`
save all files	`C-x s`
insert contents of another file into this buffer	`C-x i`
replace this file with the file you really want	`C-x C-v`
write buffer to a specified file	`C-x C-w`

Getting Help

The Help system is simple. Type `C-h` and follow the directions. If you are a first-time user, type `C-h t` for a **tutorial**.

quit Help window	`q`
scroll Help window	`space`
apropos: show commands matching a string	`C-h a`
show the function a key runs	`C-h c`
describe a function	`C-h f`
get mode-specific information	`C-h m`

Error Recovery

abort partially typed or executing command	`C-g`
recover a file lost by a system crash	`M-x recover-file`
recover files from a previous Emacs session	`M-x recover-session`
undo an unwanted change	`C-x u` or `C-_`
restore a buffer to its original contents	`M-x revert-buffer`
redraw garbaged screen	`C-l`

Incremental Search

search forward	`C-s`
search backward	`C-r`
regular expression search	`C-M-s`
reverse regular expression search	`C-M-r`
select previous search string	`M-p`
select next later search string	`M-n`
exit incremental search	`RET`
undo effect of last character	`DEL`
abort current search	`C-g`

Use `C-s` or `C-r` again to repeat the search in either direction. If Emacs is still searching, `C-g` cancels only the part not done.

Motion

entity to move over	backward	forward
character	`C-b`	`C-f`
word	`M-b`	`M-f`
line	`C-p`	`C-n`
go to line beginning (or end)	`C-a`	`C-e`
sentence	`M-a`	`M-e`
paragraph	`M-{`	`M-}`
page	`C-x [`	`C-x ]`
sexp	`C-M-b`	`C-M-f`
function	`C-M-a`	`C-M-e`
go to buffer beginning (or end)	`M-<`	`M->`

scroll to next screen	`C-v`
scroll to previous screen	`M-v`
scroll left	`C-x <`
scroll right	`C-x >`
scroll current line to center of screen	`C-u C-l`

Killing and Deleting

entity to kill	backward	forward
character (delete, not kill)	`DEL`	`C-d`
word	`M-DEL`	`M-d`
line (to end of)	`M-0 C-k`	`C-k`
sentence	`C-x DEL`	`M-k`
sexp	`M-- C-M-k`	`C-M-k`

kill region	`C-w`
copy region to kill ring	`M-w`
kill through next occurrence of *char*	`M-z` *char*
yank back last thing killed	`C-y`
replace last yank with previous kill	`M-y`

Marking

set mark here	`C-@` or `C-SPC`
exchange point and mark	`C-x C-x`
set mark *arg* words away	`M-@`
mark paragraph	`M-h`
mark page	`C-x C-p`
mark sexp	`C-M-@`
mark function	`C-M-h`
mark entire buffer	`C-x h`

Query Replace

interactively replace a text string	`M-%`
using regular expressions	`M-x query-replace-regexp`

Valid responses in query-replace mode are

replace this one, go on to next	SPC or `y`
replace this one, don't move	`,`
skip to next without replacing	DEL or `n`
replace all remaining matches	`!`
back up to the previous match	`^`
exit query-replace	ESC
enter recursive edit (`C-M-c` to exit)	`C-r`
delete match and enter recursive edit	`C-w`

Multiple Windows

delete all other windows	`C-x 1`
delete this window	`C-x 0`
split window in two vertically	`C-x 2`
split window in two horizontally	`C-x 3`
scroll other window	`C-M-v`
switch cursor to another window	`C-x o`
shrink window shorter	`M-x shrink-window`
grow window taller	`C-x ^`
shrink window narrower	`C-x {`
grow window wider	`C-x }`
select buffer in other window	`C-x 4 b`
display buffer in other window	`C-x 4 C-o`
find file in other window	`C-x 4 f`
find file read-only in other window	`C-x 4 r`
run Dired in other window	`C-x 4 d`
find tag in other window	`C-x 4 .`

Formatting

indent current line (mode-dependent)	`TAB`
indent region (mode-dependent)	`C-M-\`
indent sexp (mode-dependent)	`C-M-q`
indent region rigidly *arg* columns	`C-x TAB`
insert newline after point	`C-o`
move rest of line vertically down	`C-M-o`
delete blank lines around point	`C-x C-o`
join line with previous (with arg, next)	`M-^`
delete all white space around point	`M-\`
put exactly one space at point	`M-SPC`
fill paragraph	`M-q`
set fill column	`C-x f`
set prefix each line starts with	`C-x .`

Case Change

uppercase word	`M-u`
lowercase word	`M-l`
capitalize word	`M-c`
uppercase region	`C-x C-u`
lowercase region	`C-x C-l`
capitalize region	`M-x capitalize-region`

The Minibuffer

The following keys are defined in the minibuffer.

complete as much as possible	`TAB`
complete up to one word	`SPC`
complete and execute	`RET`
show possible completions	`?`
fetch previous minibuffer input	`M-p`
fetch next later minibuffer input	`M-n`
regexp search backward through history	`M-r`
regexp search forward through history	`M-s`
abort command	`C-g`

Type `C-x ESC ESC` to edit and repeat the last command that used the minibuffer. The following keys are then defined.

previous minibuffer command	`M-p`
next minibuffer command	`M-n`

XEmacs Reference Card

Buffers

select another buffer	C-x b
list all buffers	C-x C-b
kill a buffer	C-x k

Transposing

transpose **characters**	C-t
transpose **words**	M-t
transpose **lines**	C-x C-t
transpose **sexps**	C-M-t

Spelling Check

check spelling of current word	M-$
check spelling of all words in region	M-x ispell-region
check spelling of entire buffer	M-x ispell-buffer

Tags

find a tag (a definition)	M-.
find next occurrence of tag	C-u M-.
specify a new tags file	M-x visit-tags-table
regexp search on all files in tags table	M-x tags-search
run query-replace on all the files	M-x tags-query-replace
continue last tags search or query-replace	M-,

Shells

execute a shell command	M-!
run a shell command on the region	M-|
filter region through a shell command	C-u M-|
start a shell in window *shell*	M-x shell

Rectangles

copy rectangle to register	C-x r r
kill rectangle	C-x r k
yank rectangle	C-x r y
open rectangle, shifting text right	C-x r o
blank out rectangle	M-x clear-rectangle
prefix each line with a string	M-x string-rectangle
select rectangle with mouse	M-button1

Abbrevs

add global abbrev	C-x a g
add mode-local abbrev	C-x a l
add global expansion for this abbrev	C-x a i g
add mode-local expansion for this abbrev	C-x a i l
explicitly expand abbrev	C-x a e
expand previous word dynamically	M-/

Regular Expressions

any single character except a newline	. (dot)
zero or more repeats	*
one or more repeats	+
zero or one repeat	?
any character in the set	[...]
any character not in the set	[^ ...]
beginning of line	^
end of line	$
quote a special character *c*	\c
alternative ("or")	|
grouping	\(... \)
*n*th group	\n
beginning of buffer	\`
end of buffer	\'
word break	\b
not beginning or end of word	\B
beginning of word	\<
end of word	\>
any word-syntax character	\w
any non-word-syntax character	\W
character with syntax *c*	\sc
character with syntax not *c*	\Sc

Registers

save region in register	C-x r s
insert register contents into buffer	C-x r i
save value of point in register	C-x r SPC
jump to point saved in register	C-x r j

Info

enter the Info documentation reader	C-h i
Moving within a node:	
scroll forward	SPC
scroll reverse	DEL
beginning of node	. (dot)
Moving between nodes:	
next node	n
previous node	p
move up	u
select menu item by name	m
select *n*th menu item by number (1–5)	*n*
follow cross reference (return with l)	f
return to last node you saw	l
return to directory node	d
go to any node by name	g
Other:	
run Info **tutorial**	h
list Info commands	?
quit Info	q
search nodes for regexp	s

Keyboard Macros

start defining a keyboard macro	C-x (
end keyboard macro definition	C-x)
execute last-defined keyboard macro	C-x e
edit keyboard macro	C-x C-k
append to last keyboard macro	C-u C-x (
name last keyboard macro	M-x name-last-kbd-macro
insert Lisp definition in buffer	M-x insert-kbd-macro

Commands Dealing with Emacs Lisp

eval **sexp** before point	C-x C-e
eval current **defun**	C-M-x
eval region	M-x eval-region
eval entire **buffer**	M-x eval-current-buffer
read and eval minibuffer	M-ESC
re-execute last minibuffer command	C-x ESC ESC
read and eval Emacs Lisp file	M-x load-file
load from standard system directory	M-x load-library

Simple Customization

Here are some examples of binding global keys in Emacs Lisp.

```
(global-set-key [(control c) g] 'goto-line)
(global-set-key [(control x) (control k)] 'kill-region)
(global-set-key [(meta #)] 'query-replace-regexp)
```

An example of setting a variable in Emacs Lisp:

```
(setq backup-by-copying-when-linked t)
```

Writing Commands

```
(defun command-name (args)
  "documentation"
  (interactive "template")
  body)
```

An example:

```
(defun this-line-to-top-of-window (line)
  "Reposition line point is on to top of window.
With ARG, put point on line ARG.
Negative counts from bottom."
  (interactive "P")
  (recenter (if (null line)
                0
              (prefix-numeric-value line))))
```

The argument to `interactive` is a string specifying how to get the arguments when the function is called interactively. Type `C-h f interactive` for more information.