COMPILATION - GÉNÉRALITÉS

LA COMPILATION

En ligne de commande, on **compile** un code source avec le compilateur **gcc** (paquet : *build-essential*) :

```
gcc -c main.c -o main.o
gcc -o executable main.o
```

Le drapeau **-c** de **gcc** ne **linke** pas les fichiers sources **.c**.

> Une liaison ou édition de liens est toute opération qui établit tout ou partie de la chaîne d'accès qui permet de passer du nom d'un objet informatique à sa représentation physique.

liaison

Le **code source C** est **analysé**, grâce à l'option *-c* de *gcc*. C'est la **compilation**, à proprement parler, la **première étape** du processus de compilation, elle vérifie que le code source (*.c*) est correct et produit un fichier texte contenant le code source en **langage assembleur** (*.s*). **L'étape d'assemblage** prend le fichier précédent (*.s*) et génère du **code machine**. C'est la **deuxième étape** de la compilation et ce qui va nous donner un **fichier objet** (*.o*). Si on ne spécifie **pas de cible** avec le drapeau **-o**, alors le **fichier objet** se nommera **a.out** par défaut Le fichier objet contiendra les instructions machine générées pour le processeur. *Souvent, parmi les informations trouvées sur le net sur la compilation, l'étape d'assemblage n'est pas décrite, on passe directement du code source (.c) au code objet (.o). On dit que les fichiers intermédiaires sont gérés de façon transparente par le compilateur. Ces fichiers et formats sont temporaires et ne sont pas particulièrement visibles à l'utilisateur sauf cas spécial.*

Cela ne suffit pas à construire le programme exécutable complet, il manque encore plusieurs choses : grâce aux instructions **#include** (ex : *#include <stdio.h>*), on utilise des fonctions que l'on n'a pas écrites. Ces fonctions appartiennent à des **librairies** dont on utilise rarement le contenu entier. Le rôle de **l'édition de liens** est de déterminer quelles sont les fonctions nécessaires à notre programme, d'**extraire** de leurs librairies respectives **les blocs d'instructions processeurs** correspondants et de **relier ensemble** tous les blocs d'instructions pour former un **programme exécutable** complet. C'est la **troisième étape** de la compilation. **Éditer les liens, c'est produire l'exécutable.**

On **compile** depuis le répertoire où se situe le code source (**cd REPERTOIRE_CODE_SOURCE**, au besoin). Les drapeaux **-Wall**, **-Wextra** et **-Werror** demandent au compilateur d'**afficher** tous les **messages** de prévention et d'**erreurs**. *Si tout se passe bien, le fichier objet est créé*. Le drapeau **-I** permet d'indiquer dans quel répertoire se situent les **headers** (*.h*). En général, on les sépare des codes sources, mais s'ils sont dans le même répertoire que les sources, il est inutile d'y faire référence, *gcc* saura les retrouver.

Le drapeau **-g** permet d'ajouter des **informations** de **débogage** à l'**exécutable**, on pourra les utiliser avec le débogueur **gdb**.

```
# Compilation (l'étape d'assemblage est implicite)
gcc -Wall -Wextra -Werror -o object_file.o -c source_file.c -I headers

# Édition de liens
gcc -o program_name lname_librairy object_file.o
```

Pour **construire l'exécutable**, il faut indiquer à *gcc* quels morceaux doivent être utilisés, au minimum, le **fichier objet** généré par la compilation et éventuellement une ou plusieurs **librairies**. Le drapeau **-o** permet également d'indiquer le **nom** du programme **exécutable** à construire et le drapeau **-l**, le **nom** d'une **librairie**.

Il existe **deux types de librairies** (ou *bibliothèques*) : les **librairies statiques** et les **librairies dynamiques**.

> Librairie statique

COMPILATION - GÉNÉRALITÉS

Une **librairie statique**, généralement d'extension **.a**, est une bibliothèque qui sera **intégrée** à l'**exécutable** lors de la **compilation**. L'avantage, c'est qu'on a toutes les dépendances nécessaires au fonctionnement de l'exécutable, le programme est **autonome**. *La bibliothèque se comporte comme un fichier objet.*
L'inconvénient, c'est que le projet peut atteindre une taille conséquente, surtout si les librairies pèsent lourd. Pour faire une bibliothèque statique, on fera :

```
gcc -c file1.c -o file1.o

ar -q libfile1.a file1.o # Option -q = ajout rapide des fichiers à l'archive

# Ou encore
ar cr libfile1.a file1.0 # L'option c a pour effet de ne pas avertir si la bibliothèque doit être créée et l'option r remplace les fichiers existants ou ajoute les nouveaux à l'archive
ranlib libfile1.a # Crée un index de l'archive pour y accéder plus rapidement, l'option -s de la commande ar a le même fonctionnement

# On lie la bibliothèque comme n'importe quel autre objet, on peut se passer de l'option -l pour une librairie perso, on peut aussi rajouter l'option -static
gcc main.o -llibfile1.a -o executable
```

Librairie dynamique

Les **bibliothèques dynamiques**, d'extension **.so** (*Sharing Object*) ou **.dll** (Dynamic Link Library) sous Windows, sont des bibliothèques qui ne sont **pas intégrées** à l'**exécutable** lors de l'**édition de liens**. **L'exécutable appelle alors la librairie pour exécuter les fonctions.** L'exécutable s'en retrouve plus **léger** ; si la bibliothèque est utilisée par plusieurs programmes, elle n'est chargée qu'une seule fois en mémoire (*donc un seul téléchargement de la bibliothèque suffit*) et on peut la maintenir à jour sans avoir besoin de recompiler tout le projet. Pour faire une librairie dynamique, on fera :

```
gcc -c -fPIC file2.c -o file2.o

# On crée la librairie dynamique avec l'option -shared
gcc -shared -fPIC file2.o -o libfile2.so

gcc -c main.c -o main.0
gcc -fPIC main.o -L. -llibfile2.so -o executable
```

Le drapeau **-L** indique l'endroit **où chercher la librairie** déclarée avec **-l**, si le chemin n'est pas renseigné dans **$LD_LIBRARY_PATH**. *À noter, qu'il n'est pas nécessaire d'indiquer le nom entier de la librairie, si on met juste **-lfile2** avec **-L.**, **gcc** retrouvera sans difficulté la librairie **libfile2.so** dans le **répertoire courant**.*

Le drapeau **-fPIC** (*Position Independent Code*) **compile sans indiquer d'adresse mémoire** dans le code : les adresses pourront être différentes en fonction du programme qui utilisera la bibliothèque. Cela évitera aussi les **conflits** entre bibliothèques.

Le drapeau **-shared** indique que la bibliothèque est **partagée** (ou **dynamique**).

Il faut encore appeler la bibliothèque, on procèdera un peu de la même façon que pour appeler un programme sans **./** : **on doit renseigner son chemin via la constante**.

```
export LD_LIBRARY_PATH=chemin_librairie :$LD_LIBRARY_PATH

# Ou encore, de façon plus définitive
vim ~/.profile
LD_LIBRARY_PATH=chemin_librairie :$LD_LIBRARY_PATH # Trouver l'emplacement approprié dans le fichier .profile (dépend de la structure du fichier)

# Si le fichier ~/.profile n'existe pas :
cp /etc/profile ~/.profile
echo export LD_LIBRARY_PATH=chemin_librairie :$LD_LIBRARY_PATH > ~/.profile # Ajoute l'instruction à la fin du fichier
```

On ne copie surtout pas la bibliothèque dans le répertoire **/lib**, cela fonctionnera mais c'est une **mauvaise pratique**. En effet, c'est la librairie dédiée aux programmes systèmes. Pour les librairies personnelles, on doit utiliser le répertoire **/usr/local/lib**. Le plus simple est d'avoir ses librairies dans un répertoire sur son home et on fera des liens symboliques vers /usr/local/lib/.

Mettre à jour une bibliothèque dynamique :

gcc appelle le linker **ld** pour lier une bibliothèque. *Lire la doc pour plus de précision*. Quand on change la version d'une bibliothèque (pour la mineure comme pour la majeure), il faut l'indiquer à l'éditeur de lien, sans quoi il risque de pointer sur une version caduque.

```
# Commande pour MàJ de bibliothèques dynamiques
ld -soname

# gcc appelle ld pour éditer des liens, avec le flag -Wl, il passe des options à ld, comme le flag -soname. Bien tenir compte des virgules de séparation.
gcc -Wl,-soname, libfichier1.so.1 -o libfichier.so.1.1

# Lire les man concernant ces commandes ci-dessous
ldconfig
ldd
```

Compiler avec des bibliothèques tierces ; (GTK ou SDL par exemple) :

```
pkg-config --libs [librairie].
```

sdl fournit sa propre méthode **sdl-config --cflags** = **pkg-config --cflags sdl**

Exemple avec GTK+

```
gcc main.c $(pkg-config --cflags --libs gtk+-2.0) -o exécutable

# Équivalent à :
gcc main.c $(pkg-config --cflags gtk+-2.0) -o main.o
gcc main.o $(pkg-config --libs gtk+-2.0) -o exécutable
```

pkg-config cherche la bibliothèque **.pc** dans le répertoire **/usr/lib/pkgconfig**

MAKEFILE

Le **Makefile** est un **fichier texte** comportant les **instructions de montage** de notre programme **C** (*ou autre langage à compiler, C++, Java…*) que le programme **make** a besoin pour générer la compilation. C'est au développeur de l'écrire. Il permet d'**automatiser** les étapes de la **construction** de l'**exécutable**. Il a **formatage** très particulier :

```
# Commentaire Makefile
cible: dépendances
    <TAB>commande
    <TAB>commande
```

La **cible** est le plus souvent un **nom d'exécutable** ou de **fichier objet**. Les **dépendances** (ou **pré-requis**) sont les **éléments** ou le **code source** nécessaire pour **créer** une **cible**, les **pré-requis** sont **facultatifs**, on peut les omettre. Les **commandes** (ou **recettes**) sont les **commandes** nécessaires pour **créer** la **cible**. L'ensemble **cible / pré- requis / recettes** est une **règle**. Un **Makefile** est constitué d'une **collection** de **règles**. Par convention, ce fichier doit s'appeler **Makefile** sans extension et être situé dans le **même répertoire** que la **1ere cible** (*en général le nom du programme à reconstituer*).

La **cible** est séparée des **dépendances** par « **:** » et les **commandes** sont toujours **précédées** d'une **tabulation** (*pas d'espaces !*) ou par le caractère indiqué dans la directive **.RECIPEPREFIX** (par défaut : tabulation).

Exemple très simple :

```
# Génération du Makefile, on pourrait passer par vim
echo -e «hello-world:\n\techo \«Hello, world\»» > Makefile

# Que contient ce Makefile ?
cat Makefile
hello-world:
    echo «Hello, world»

# Pour exécuter la commande :
make hello-world
echo «Hello, world»
Hello, world
```

Depuis **vim**, on peut faire :

```
:make
```

Cela lancera le programme exactement comme en ligne de commande.

Si pour une raison quelconque le **makefile** ne s'appelle pas **Makefile** mais **filename** (*par exemple*), alors faire ceci :

```
make -f filename
```

On peut rendre une ligne **silencieuse** en la préfixant du symbole **@**

```
sed -ire «s/echo \«Hello, world\»/@ &/» Makefile

# Que contient ce Makefile ?
cat Makefile
hello-world:
    @ echo «Hello, world»

# En exécutant la commande, on obtient :
make hello-world
Hello, world
```

On **invoque une règle** depuis la ligne de commande en spécifiant le **nom de sa cible** après la commande **make**. Si **rien** n'est précisé (si on tape juste *make*), alors c'est la **première règle trouvée** qui est **exécutée**.

Les **dépendances** sont **indispensables** lorsque l'on veut **construire** un **fichier depuis** un **autre**. **C'est d'ailleurs la principale action d'un processus de compilation** (*voire sa définition*).

make vérifie les **dates** de **modification** des **pré-requis** et les **compare** avec les dates de modifications de la **cible** pour savoir s'il faut reconstruire ou non la cible. **make** ne reconstruit la cible que si les **dépendances** sont **plus récentes** que la **cible**.

On peut avoir une **cible factice** qui ne représente pas un fichier mais qui dépend elle-même dépend de plusieurs fichiers. Dans ces cas-là, on va **vouloir exécuter la règle quoiqu'il arrive**, comme si la **cible** était **tout le temps périmée**. On utilise la cible particulière **.PHONY** (*de l'anglais phoney, bidon, en toc, faux*). On l'ajoute avant la cible factice, voire en **début** de **Makefile**.

```
.PHONY : website, clean

# Ici, on a des dépendances mais pas de commandes
website: index.html apropos.html contact.html

# Ici, on n'a pas de dépendances mais on a une commande
clean:
    rm -rf *.o
```

Les cibles commençant par **.** ne sont pas prises en compte par **make** comme des **cibles**, ainsi placer **.PHONY** en 1ere position ne gène absolument pas **make** pour considérer concrètement la **vraie 1ère cible**, *qui est normalement censée être l'exécutable à produire*.

De même, les **cibles factices** n'ayant **pas** de **pré-requis**, à proprement parler, **make** ne les exécutera jamais de lui-même, il faudra lui préciser, lors de l'appel de **make**, en ligne de commande, qu'on souhaite effectuer l'action correspondante à la cible.

```
make clean

# Ou encore
make install
```

La logique est la suivante : si l'**action** (comme *clean*) n'est **pas** un **pré-requis** (*ou un pré-requis d'un pré-requis d'un pré-requis* etc.) de l'**exécutable** à produire, alors **make** n'évaluera pas la règle de la cible « action ». Comme elle ne sera jamais plus récente que l'exécutable, puisqu'il ne s'agit pas d'un fichier, elle ne sera pas prise en compte, hormis lors d'un appel fait par l'utilisateur. On se **prémunira d'une erreur** d'interprétation, en préfixant la règle correspondant de **-** : cela indique à **make** qu'il ne s'agit pas d'un fichier à évaluer, et dans le cas de **clean** (ou autre) évite de prendre la **cible clean** pour un fichier potentiellement existant dans le répertoire de l'exécutable.

Voici un exemple simple de Makefile :

```
all: executable

# Édition de liens
executable: file1.o file2.0
    gcc -o executable file1.o file2.o

# Assemblage, à ce stade file1.o ne connait pas les adresses des fonctions du code source file1.c
file1.o: file1.c file1.h
    gcc -c file1.c

file2.o: file2.c file2.h
    gcc -c file2.c
```

```
clean:
    -rm file1.o file2.o executable core
```

Dans l'absolu, on passera par des **règles génériques** pour s'éviter d'écrire une ligne Makefile pour chaque fichier source.

Cibles standards – section 16.6 du manuel make pour liste complète

Il existe certaines conventions concernant certaines cibles à utiliser, les **cibles standards** , toutes les règles sont à écrire manuellement par le programmeur:

clean: supprime les fichiers temporaires (objets, voire l'exécutable).

```
clean:
    rm -rf file1.o file2.o
```

mrproper: permet un rebuild complet -> on appelle la règle **clean** comme dépendances et on supprime l'exécutable pour que **make** puisse le reconstruire.

```
mrproper: clean
    rm -rf exécutable
```

all: compile tous les fichiers sources pour créer l'exécutable principal.

```
all: main.o fonctions.o
    gcc main.o fonctions.o -o exécutable
```

install: exécute **all**, et copie l'exécutable, les librairies, les datas et les fichiers d'en-tête dans le répertoire de destination (attention, dans le manuel make, on fait clairement référence à 3 étapes pendant l'installation : le **PRÉ**, la **POST** et l'**installation** elle-même)

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).

info: génère un **fichier info**. La variable **MAKEINFO** doit être définie. Le programme **makeinfo** sera exécuté, c'est une partie de la distribution Texinfo.

```
info: foo.info


foo.info: foo.texi chap1.texi chap2.texi
    $(MAKEINFO) $(srcdir)/foo.texi
```

dvi: génère un fichier **dvi**, fichier d'information. La cible peut aussi être **html**, **ps**, ou **pdf**

dist: crée un fichier tar de distribution.

Les règles implicites – section 10.2 pour liste POSIX

make fonctionne aussi grâce à certaines règles **implicites** : cela permet de réduire la **taille** des **recettes**. Ainsi, il n'est pas nécessaire d'ajouter les **fichiers sources .c** comme **pré-requis**, car **make** a une règle pour mettre à jour de façon implicite les fichiers **.o**. En effet, **make** compile automatiquement avec l'instruction `cc -c main.c -o main.o` si la cible est **main.o**. On peut alors écrire :

```
main.o: mon_header_perso.h
```

make comprendra qu'il faut compiler **main.c** et **mon_header_perso.h** pour obtenir le fichier objet **main.o**.

Également dans le but d'optimiser le rendu du **Makefile**, on peut **regrouper** les **cibles** utilisant les **mêmes dépendances** pour se construire. *Cela évite la duplication de code :*

```
main.o fonctions.o display.o: mon_header_perso.h
utils.o command.o: autre_header_perso.h

# Plutôt que :
main.o: mon_header_perso.h
fonctions.o: mon_header_perso.h
display.o: mon_header_perso.h
utils.o: autre_header_perso.h
command.o: autre_header_perso.h
```

Si on ne veut pas qu'une règle **implicite** s'applique sur une **cible sans recette**, alors il convient de donner à la cible une **recette « vide »** grâce à un point-virgule (semi-colon en anglais).

```
foo: ;
```

Pour **désactiver** toutes les **règles implicites**, on ajoutera la directive suivante, en tête du **Makefile** (cela permet de résoudre les conflits avec des Makefiles plus complexes, *comme compiler un fichier Pascal -> la règle implicite de compilation en C est définie avant celle du Pascal, par conséquent, c'est la règle C qui s'appliquera, donc l'effet attendu ne sera pas obtenu*) :

```
.SUFFIXES:
```

Pour connaître les règles implicites applicables à l'OS sur lequel on évolue, taper en ligne de commande, dans un répertoire **sans Makefile**, la commande `make -p`. La liste des règles implicites s'affichera (sinon, si lancé dans un répertoire avec un **Makefile**, affichera la liste des règle du **Makefile**).

Les variables – sections 10.3 & 16.5

Afin d'obtenir une généricité des règles, un des outils est l'emploi de variables (personnalisées ou non). On essaie de les déclarer au début du Makefile.

```
# Définition et affectation de la variable
NOM=VALEUR

# Appel de la variable
$(NOM)
# Ou
${NOM}

# CC est par convention, la variable servant à définir le compilateur, CFLAGS, la variable contenant les drapeaux à passer au compilateur C.
CFLAGS= -Wall -Werror -Wextra -pedantic -ansi
CC=gcc $(CFLAGS)

# On n'ajoute surtout pas de commentaires à la fin d'une variable, sur la même ligne. Tout ce qui est sur la ligne de déclaration de la variable est considéré comme du texte et interprété littéralement
NOM=VALEUR # Commentaires
$(NOM) donnera VALEUR # Commentaires

# On peut faire la déclaration d'une variable sur plusieurs lignes en utilisant \ en fin de ligne. Le retour chariot sera remplacé par une espace au lancement. Attention, make interprète/substitue (expand en anglais) une première fois la valeur des variables, puis une seconde fois avant de lancer le Makefile. Selon la façon de déclarer une variable (= ou :=), la variable aura sa valeur définitive attribuée au premier (=) ou au second tour( :=), cf 6.2 « The Two Flavors of Variables »
CFLAGS= -Wall \
        -Wextra \
        -Werror
```

Il est recommandé d'utiliser une **casse basse** pour les noms de variables ayant un **usage interne** (comme *objects*) dans le **makefile** et de réserver une **casse haute** pour les variables contrôlant les **règles implicites** (comme *RM* ou *CC*), ou les **paramètres** que l'utilisateur peut **surcharger** (*CFLAGS*). Pour obtenir une **liste** des variables préexistantes dans **make** et utilisées par les règles implicites, voir la section **10.3** du manuel **make**. Pour une liste exhaustive des **variables** concernant les **répertoires** et **destinations** (du style **$(prefix)**=>**/usr/local**), voir la section **16.5** du manuel.

Il existe plusieurs façons de déclarer une variable :

```
NOM=VALEUR

# Ou
NOM:=VALEUR

# Ou
NOM?=VALEUR
```

Outre la question de quand la valeur de la variable est substituée, c'est surtout une question de ce qui se passe quand la variable est substituée qui rend intéressant le concept « *The Two Flavors* » (les deux déclinaisons). L'assignation « **=** » est équivalente à l'instruction **define VARIABLE**. On l'appelle « *recursive expansion* » L'assignation « **?=** » est conditionnelle, l'assignement ne se produit que lorsque la **variable** n'a **pas** déjà été **définie**. La **variable** assignée par « **:=** » ou « **::=** » (POSIX) est appelée « *simply expanded variable* ». La différence majeure d'avec =, c'est qu'il ne contient pas de **références** à d'autres variables.

```
foo=$(bar)
bar=$(baz)
baz=Hello !

all :;echo $(foo)
```

`echo $(foo)` affichera : « `Hello !` »

```
# Attention, on ne peut pas faire ceci, car cela créera une boucle infinie
foo=$(foo)

# Pour ajouter une valeur à $foo, on fera
foo+=VALEUR
```

L'inconvénient, c'est que cette façon de déclarer sera exécutée à chaque fois qu'elle est substituée, si elle existe dans le corps d'une fonction. Cela rend **make** lent et certaines fonctions internes ont des résultats imprédictibles. Pour pallier à ces manquements, on utilisera la **seconde déclinaison** `:=`.

En effet, les variables assignées avec `:=` fonctionnent comme la plupart des variables de la majorité des langages de programmations. On peut les **redéfinir avec leurs propres valeurs** (*alors que ça cause une boucle infinie dans la 1ere déclinaison*), la gestion des espaces en fin de variables est meilleure et les **commentaires** à la suite de la déclaration ne sont **pas interprétés** littéralement comme faisant partie de la valeur de la variable (*D'ailleurs on se sert des commentaires pour ajouter une espace à la fin de la valeur d'une variable*).

```
foo:=$(foo) autre_valeur_ajoutee
```

Un inconvénient de la 2nde **déclinaison** est justement **l'expansion immédiate** : si la variable n'est pas définie, mais qu'elle est appelée dans une recette, alors elle sera remplacée par un vide, tandis que si la variable est assignée avec `=` mais qu'elle est vide alors elle sera remplacée par son appel et non son expansion, cela peut être très utile dans une recette :

```
CFLAGS=$(include_dirs) -O
include_dirs=-Ifoo -bar
```

Si `$(include_dirs)` est une variable récursive alors `$(CFLAGS)` sera interprété comme `$(include_dirs) -O`, si c'est une variable de simple expansion, alors `$(CFLAGS)` sera interprété comme `-Ifoo -bar -O`. Cette expansion peut être un problème lorsque la variable n'est pas définie, il sera alors interprété : `<vide> -O`. Et ce n'est pas ce que l'on recherche. **CFLAGS** étant une variable préexistante, il vaut mieux la définir comme **récursive**, ainsi nous lui laissons l'occasion d'être lue au 2e passage de **make** (il est possible qu'elle soit définie plus bas dans le script, et comme **make** ne re substitue pas une variable substituée au 1er passage… cqfd).

Un autre aspect intéressant de la 2nde **déclinaison** est la **substitution de références** (*cf* `6.3.1` *du manuel make*) : la valeur d'une variable est substituée avec l'altération qu'on aura spécifiée.

```
foo := a.o b.o l.a c.o
bar := $(foo:.o=.c)
```

configurera **bar** avec la valeur **a.c b.c l.a c.c**. La substitution est de forme `$(var:a=b)` remplacera *a* par *b* et où *a* est uniquement à la **fin du mot**, les autres occurrences (début, milieu…) ne seront pas remplacées.

Une autre façon de faire une **substitution**, permettant d'utiliser la pleine puissance de la fonction `patsubst` est d'employer le caractère `%`. La substitution est de la **même forme** que ci-dessus :

```
foo := a.o b.o l.a c.o
bar := $(foo:%.o=%.c)
```

En gros, tout ce qui est **préfixé** par **%** (*stem*) et se terminant par **.o**, dans la variable **foo**, est remplacé par le préfixe **%** et le suffixe **.c**. On retrouve alors dans la variable **bar**, les valeurs **a.c b.c l.a c.c**.

On peut aussi assigner une variable dans la partie pré-requis d'une règle : la notion s'appelle `Target-specific Variable Values`. Cependant, cela répond à des critères bien spécifiques. Consulter la section `6.11` du manuel make.

Static Pattern Rules - Sections 4.12.1 & 10.5

Il s'agit d'utiliser un motif pour retrouver ses plus grandes correspondances et y appliquer les transformations voulues, à la manière d'une **regex**. On utilise principalement les caractères `?` `*` et `%`.

? remplace **une** lettre, ***** **plusieurs** lettres et **%** a le même comportement que *****, sauf qu'on l'emploie uniquement pour les **cibles**. On appelle **%** un **pattern** (motif), **?** et ***** des **jokers**. La **sous-chaîne** correspondant à **%** est appelée **stem** (*tronc*, radical –au sens de *racine lexicale*- en anglais).

Les ***static pattern rules*** sont des règles spécifiant de multiples cibles et construisant les noms des pré-requis basés sur chaque nom cible correspondant.

```
targets … : target-pattern: prereq-patterns…
    recipe
    …

# Exemple
objects :main.o foo.o bar.o

all: $(objects)

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

L'effet obtenu est la compilation des fichiers objets à partir de leurs sources .c.

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

Tous les **fichiers sources .c pré-requis** se verront **compilés** en **fichiers objets**. Ils auront la même racine de noms que leurs fichiers sources.

Variables automatiques section 10.5.3 du manuel make

$@

Correspond au **nom** de la **cible** de la règle. Dans une règle **multi-cibles**, correspond au **nom** de la **cible lançant** la **recette** de la règle.

$%

Si la **cible** est **membre** d'une **archive** (*cf section* **11.1** *du manuel*) donne son nom. Si la cible est **foo.a(bar.o)**, alors **$%** correspond à **bar.o** et **$@** à **foo.a**. **$%** est **vide** s'il ne s'agit pas d'un membre archive.

$<

Donne le **nom** du **1er pré-requis**.

$?

Donne le **nom** de tous les **pré-requis** qui sont **plus récents** que la **cible**, séparés par des espaces. *Si la cible n'existe pas, alors tous les pré-requis sont donnés.* Signification spéciale si membre d'une archive (seulement le nom du membre est utilisé).

$^

Donne le **nom** de tous les **pré-requis**, séparés par des espaces. Si un **pré-requis** est mentionné plusieurs fois dans la partie pré-requis, l n'est **affiché** qu'**une seule fois** via $^. *Ne donne aucun nom de pré-requis « order-only ».*

$+

Comme **$^**, à la différence que si un **pré-requis** est inscrit plus d'une fois dans la liste des pré-requis, il sera **cité autant** de fois qu'il **apparaît** et dans l'ordre dans lequel il est inséré. C'est principalement utile pour les commandes d'édition de liens où il y a un sens à répéter plusieurs fois le nom d'une bibliothèque dans un ordre particulier.

$|

Cf section **4.3** pour la notion de pré-requis « *order-only* ». Donne le **nom** de tous ces **types** de pré-rquis.

$*

Donne la **racine** (stem) qui correspond à **%**, dans une règle **implicite**. Dans une règle **explicite**, donne le **nom** de la cible **moins** son **suffixe** (extension). Ainsi dans *foo.c*, *$** renverra *foo*.

'$(@D)'

Correspond à la partie **répertoire** de la cible dans un nom de **fichier**, / est enlevé.

'$(@F)'

Correspond à la partie **fichier** de la cible dans un nom de **fichier**.

'$(*D)'

'$(*F)'

Correspond respectivement à la partie **répertoire** et à la partie **fichier** d'une **racine** (stem).

'$(%D)'

'$(%F)'

Correspond respectivement à la partie **répertoire** et à la partie **fichier** d'une cible **archive** avec membre.

'$(<D)'

'$(<F)'

Correspond respectivement à la partie **répertoire** et à la partie **fichier** du **1er pré-requis**.

'$(^D)'

'$(^F)'

Correspond respectivement à la partie **répertoire** et à la partie **fichier** de **tous les pré-requis**.

`'$(+D)'`

`'$(+F)'`

Correspond respectivement à la partie **répertoire** et à la partie **fichier** de **tous les pré-requis**, incluant les multiples instances des pré-requis dupliqués.

`'$(?D)'`

`'$(?F)'`

Correspond respectivement à la partie **répertoire** et à la partie **fichier** de **tous les pré-requis** plus **récents** que la cible.

Conditions – Section 7

On peut émettre des **conditions** dans un **Makefile**. La section **7** du manuel évoque les cas de figure et la syntaxe à employer. Il s'agit de directives conditionnelles, sous forme de macros. La syntaxe est la suivante :

```
conditionnal-directive
text-if-true
[else
text-if-false]…
endif
```

La partie **else** est **optionnelle**, on peut répéter cette partie plusieurs fois. 4 invariants sont utilisables : **ifeq**, **ifneq**, **ifdef** et **ifndef**

```
# Si arg1 = arg2 (ifeq) ou non égal (ifneq), texte si vrai
ifeq (arg1, arg2)
text-if-true
endif


# Pour ifdef, cela fonctionne comme les macros C
# Si la variable est définie (ifdef) ou n'est pas définie (ifndef), alors texte si vrai
foo=Hello

ifdef $(foo)
text-if-true
endif
```

Il existe des fonctions conditionnelles **$(if ...)**, **$(and ...)** et **$(or ...)**. *Lire le manuel*.

Fonctions section 8 du manuel make

On peut utiliser des **fonctions prédéfinies** par **make**, ou créer ses **propres fonctions** en utilisant la fonction **call**, on récupèrera les paramètres de la fonction personnalisée avec des variables **positionnelles** : **$(1)**, **$(2)**,….

Un appel de fonction ressemble à une référence de variable.

```
$(function arguments)

# Synopsis fonction call (pas d'espaces entres les arguments)
$(call variable,param,param,…)

# Création d'une fonction personnalisée
reverse= $(2) $(1)

foo=$(call reverse,a,b)
```

La variable **foo** contiendra « **b,a** ».

Exemple d'un Makefile

```
CC=gcc
headers_dir=./HEADERS
HEADERS_CFLAGS=-I $(headers_dir)
CFLAGS=-Wall -Werror -Wextra
CDEBUG=-g
ALL_CFLAGS=$(CDEBUG) $(HEADERS_CFLAGS) $(CFLAGS)
# Variable pour l'édition de liens : -Wl, -soname, -L, -l
LDFLAGS=
RM= -rm -fr
exec=executable
src=$(wildcard *.c)
obj=$(SRC:.c=.o)

all: $(exec)
```

```
# Tous les tutos passe par le vrai nom de l'exécutable et non sa variable, pourtant il est dit en section 6 du manuel make que la valeur d'une variable peut être substituée dans une cible, à vérifier donc
$(exec): $(obj)
    $(CC) -o $@ $^ $(LDFLAGS) ; chmod +x $@

main.o: fonctions.h

%.o :%.c
    $(CC) -o $@ -c $< $(ALL_CFLAGS)

.PHONY: clean mrproper

clean:
    $(RM) *.o

mrproper: clean
    $(RM) $(EXEC)
```

Flags de make section 9.7

Lire le manuel car il y en a toute une flopée (bon pas autant que pour gcc). On notera le flag `-k` qui continue d'exécuter le Makefile, malgré des erreurs, `-f` pour spécifier un makefile dont le nom ne suivrait pas la convention de nommage, `-e` qui donne la priorité aux variables d'environnement sur celles définies par **make** et `-C` qui change le répertoire, comme la commande **cd**. On peut avoir plusieurs flags **-C**. `-d` affiche le **débogage**.

Pour lancer une commande **make** depuis un **Makefile**, on passera par la variable `$(MAKE)`.

Automake

À un niveau supérieur d'expérimentation, on utilisera les outils `Automake` et `Autoconf`. Ce sont des outils de configuration et de création de scripts automatiques. C'est ce qui nous permet de construire un programme en deux étapes avec les instructions `./configure` et **make**. Le **Makefile** se génère automatiquement. Ce n'est pas l'objet de cette fiche et c'est toujours bien de savoir écrire un **Makefile** et de mettre les mains dans le cambouis.

RENDRE EXÉCUTABLE LE PROGRAMME

On n'oublie pas d'attribuer les droits d'exécution au programme généré, sans quoi, on risque d'avoir des petits soucis à l'appel du programme. Dons, depuis le dossier de l'exécutable :

```
# Rend le programme exécutable
chmod +x executable

# Lance le programme
./executable
```

ADRESSES UTILES

https://www.gnu.org/software/make/manual/make.html
https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/
http://sdz.tdct.org/sdz/compilez-sous-gnu-linux.html
https://putaindecode.io/articles/laissez-vous-pousser-la-barbe-apprenez-a-ecrire-des-makefiles/
Automake & Autoconf(http://www-igm.univ-mlv.fr/~dr/XPOSE/Breugnot/)
https://www.april.org/sites/default/files/groupes/doc/make/make.html

MAKEFILE AVEC GESTION DES DÉPENDANCES (HEADERS)

```
CC=gcc
headers_dir=./HEADERS
HEADERS_CFLAGS=-I $(headers_dir)
CFLAGS=-Wall -Werror -Wextra -ansi -Wpedantic
CDEBUG=-g gdb # Ne pas utiliser en prod
ALL_CFLAGS=$(CDEBUG) $(HEADERS_CFLAGS) $(CFLAGS)
#Pour édition de liens : -Wl, -soname, -L laisser libre si aucune library
LDFLAGS= -L./LIBS -llibexec
DEP_CFLAGS=-MM -MD
# ou pour RM=-/bin/rm -rf (valable pour toutes les commandes)
RM=-rm -rf
NAME=nom-du-programme
# Pas sur que wildcard soit autorisé, noter les sources .c une à une
src=$(wildcard *.c)
obj=$(src:.c=.o)
dep=$(src:.c=.d)

.PHONY: all clean fclean re

all: $(NAME)

$(NAME): $(obj)
    $(CC) $(LDFLAGS) $^ -o $@; chmod +x $@

%.o:%.c
    $(CC) -c $< $(ALL_CFLAGS) -o $@

%.d:%.c
    $(CC) $< $(DEP_CFLAGS) $(ALL_CFLAGS) -o $@

clean:
    $(RM) $(obj) $(dep)

fclean: clean
    $(RM) $(NAME)

re: fclean all

-include $(dep)
```