Laboratoire

Introduction

mba, mwa, nvs

septembre 2021

Table des matières

1 Makefile simple **2**
1.1 Intro - LaboIntro 01-01 - un Makefile 2

2 Un Makefile à améliorer **5**
2.1 Intro - LaboIntro 01-02 - Compilation de plusieurs sources 5

3 Automatiser la démonstration **8**
3.1 Intro - LaboIntro 01-03 - script Demo 8

4 Un premier exercice : découvrir le shell **10**
4.1 Intro - LaboIntro 01-04 - Fonctionnement d'un shell, premiers pas 10

5 Makefile, règles implicites **13**
5.1 Intro - LaboIntro 01-05 - sans Makefile 13

Chapitre 1

Makefile simple

1.1 Intro - LaboIntro 01-01 - un Makefile

Titre :	Intro - LaboIntro 01-01 - un Makefile
Support :	OS 43.2 Leap
Date :	08/2018

1.1.1 Make

L'utilitaire *make* permet de déterminer quelles parties d'un programme complexe nécessitent une compilation et exécute les commandes qui lui auront été renseignées pour compiler. Nous allons l'utiliser dans ce cadre.
Un Makefile est un fichier contenant des règles répondant au format suivant :

cible : pré-requis ...
<tab> commande

make sait ainsi que pour produire *cible* il aura besoin de *pré-requis* et devra exécuter *commande*

1.1.2 Énoncé

Écrivez un Makefile contenant plusieurs règles.
La première (qui est la règle par défaut) exécute votre programme *TestMake*. Ce petit exécutable ne fait rien de spécial, il affiche trois représentations d'une même valeur.
Le fichier exécutable TestMake doit être généré si il n'existe pas ou n'est pas à jour. Vous allez avoir besoin d'une règle pour décrire cela. Comme contrainte, le code TestMake.c utilise un fichier d'include *MonInclude.h*.
Vous écrirez également la règle ou cible *clean* qui n'a aucun pré-requis et sert juste à supprimer l'exécutable *TestMake* pour revenir à une situation de départ ou "faire du propre", comme son nom l'indique.
Exécutez ensuite manuellement la commande `make clean` suivie de deux fois la commande `make`.
Observez et justifiez la différence de comportement entre ces deux dernières exécutions.
Modifiez le fichier .c ou/et le fichier .h (ou modifiez leur date par la commande *touch <fichier>*).
Quel comportement génère la commande `make` maintenant ?

1.1.3 Une solution

Voir dans LaboIntro0101/SOURCES

```
#NOM    : Makefile
#CLASSE : Intro - LaboIntro 01-01
#OBJET  : autour du Makefile
#HOWTO   : make; make clean
#AUTEUR  : mba 08/2016

# La première règle est résolue par la commande "make Test"
# c'est aussi la règle appelée par défaut car c'est la première
# elle est donc également résolue par le commande "make" sans paramètres
# Cette règle exécute "TestMake" par la commande ./TestMake
# précédée ici par le caractère @ qui empêche l'affichage de la ligne de commande exécutée.
# Si l'exécutable n'existe pas ou n'est pas à jour, l'outil make se charge de le générer

Test:TestMake
    @./TestMake

         # la commande est obligatoirement précédée d'une tabulation
         # cela diffère d'une série d'espaces
         # @ pour ne pas afficher la comande même

# La deuxième règle est invoquée par "make TestMake"
# elle est également invoquée quand "TestMake" est manquant et est le prérequis d'une autre règle
# Elle sert à regénérer le fichier exécutable
# notamment, si le fichier source ou le fichier include ont été modifiés
# La commande de compilation est : gcc TestMake.c -o TestMake
# $@ notation qui désigne la cible de la règle : TestMake
# $< notation qui désigne le premier prérequis de la règle : TestMake.c

TestMake : TestMake.c  MonInclude.h
   gcc -std=c99 $< -o $@

# La troisième règle est invoquée par "make clean"
# c'est d'ailleurs la seule manière d'invoquer cette règle car la cible "clean"
# n'apparaît nulle part comme prérequis d'une autre règle
# Elle n'a aucun prérequis non plus.

clean:
   rm -f TestMake *~
```

```
/*
 * NOM - monInclude.h
 */
#define MAX 10
```

```
/*
NOM      : TestMake.c
CLASSE   : intro - LaboIntro  01-01
#OBJET  : réservé au makefile
AUTEUR   : mba 01/2016
*/
#include <stdlib.h>
#include <stdio.h>
#include "MonInclude.h"
int main ( int argc, char * argv[] )
{
   printf("%d, %o, %X\n",MAX,MAX,MAX);
   // afficher MAX en base 10, 8 et 16
      // man format pour les spécifications de conversion de printf
   exit(0);
}
```

1.1.4 Commentaires

— nous choisissons le nom Makefile pour le fichier makefile c'est un des noms de fichier que l'outil `make` emploie par défaut.
— un programme ne doit pas être recompilé à chaque exécution.

— le programme nécessite d'être recompilé si le fichier .c ou le fichier .h dont il dépend, ont une date plus récente que l'exécutable. Ceci est exprimé par une règle et sera interprété par l'outil `make`.
— un pré-requis dans une règle de Makefile n'est pas toujours un nom de fichier il peut s'agir d'un nom de règle.

1.1.5 Voir

Pour une description complète de l'outil `make`
GNU Make : (Stallman, McGrath, Smith)

Chapitre 2

Un Makefile à améliorer

2.1 Intro - LaboIntro 01-02 - Compilation de plusieurs sources

Titre :	Intro - LaboIntro 01-02 - Compilation de plusieurs sources
Support :	0S 42.3 Leap
Date :	08/2016

2.1.1 Énoncé

Écrivez un Makefile qui réalise la compilation et l'édition de liens de plusieurs sources en C.

```
#NOM    : Makefile
#CLASSE : Intro - LaboIntro 01-02
#OBJET  : autour du Makefile
#HOWTO   : make
#AUTEUR  : mba 01/2016
#

Test:TestMake2
   @# Comment exécuter le programme TestMake2
   @echo
   @echo "vous pouvez maintenant exécuter votre programme TestMake2 avec un paramètre numérique. "
   @echo
   @echo "par exemple : ./TestMake2 3"
   @echo
   @echo


TestMake2 : TestMake2.c  MonInclude.h boucle.c boucle.h

   @# compiler le programme
   gcc -std=c99  -Wall -pedantic TestMake2.c -c -o TestMake2.o

   @# compile TestMake2.c (compilation séparée produit le fichier objet)
   @# compilation séparée de boucle.c
   @# (pas de fonction main) produit l'objet
   gcc -std=c99 -Wall  -pedantic boucle.c -c -o boucle.o

   @# édition des liens TestMake2.o et boucle.o
   @# produit l'exécutable TestMake2
   gcc TestMake2.o boucle.o -o TestMake2


clean:
   @# supprime les fichiers objet, l'exécutable
   @# et les fichiers temporaires
   @rm -f TestMake2 *.o  *~
```

```
/*
 * NOM - MonInclude.h
```

```
 */
#define MAX 10
```

```
/*
NOM      : TestMake2.c
CLASSE   : intro - LaboIntro  01-02
#OBJET  : réservé au makefile
AUTEUR   : mba 01/2016
 */
#include <stdlib.h>
#include <stdio.h>
#include "MonInclude.h"
#include "boucle.h"

int main ( int argc, char * argv[] )
{  int i;
   if (argc < 2)  { // argv[0] est le nom du programme
      printf ("\n\nusage %s <nombre>\n\n",argv[0]);
      exit(1);
   }

   i = atoi (argv[1]); // une chaine numérique qui devient un entier
   if (i>MAX) {
      printf ("MAX = %d !\n\n", MAX);
      i=MAX;
   }

   printf("%d\n",i);

   onBoucle ("Salut ! ", i);
   exit(0);
}
```

```
/*
 NOM  : boucle.h
 CLASSE : intro - LaboIntro  01-02
 AUTEUR : MBA
 DATE : 01/2016
*/

void onBoucle (char *texte, int compt);
```

```
/*
 NOM  : boucle.c
 CLASSE : intro - LaboIntro  01-02
 AUTEUR : MBA
 DATE : 01/2016
*/
#include <stdlib.h>
#include <stdio.h>

void onBoucle (char *texte, int compt){
   for (int i = 1; i<= compt; i++)
      printf ("%s", texte);
   printf ("\n");
}
```

2.1.2 Commentaires

Le fichier Makefile de cet exercice n'est pas écrit de manière optimale.
En effet la moindre modification de fichier source c ou de fichier d'include amène à recompiler chaque fichier source et refaire l'édition de liens des objets générés pour produire le fichier exécutable.
Or, lorsque un fichier source est modifié, seul ce dernier nécessite d'être recompilé avant de refaire l'édition de liens.
La compilation de chaque source n'est donc nécessaire que si le fichier source même ou un des fichiers d'include qu'il utilise ont été modifiés.
Par ailleurs, une édition de liens pour générer un exécutable est nécessaire dès qu'un fichier objet qui le construit

a été généré par une compilation.
Nous venons d'établir des dépendances entre fichiers : un exécutable dépend de fichiers objet qui à leur tour dépendent de fichiers sources et include. Ceci va nous aider à améliorer notre Makefile.

- Il est nécessaire de recompiler une unité de compilation (et donc générer un fichier objet .o) si et seulement si le fichier .o a été supprimé ou le fichier source ou un des include files a été modifié.
- Il est nécessaire de faire une édition de liens si et seulement si le fichier exécutable a été supprimé ou un des fichiers .o (objets) est plus récent que l'exécutable. Ces derniers nécessitent d'ailleurs peut-être d'être rafraîchis en premier lieu. Les pré-requis sont examinés en cascade dans le cas de dépendances.

2.1.3 En Roue Libre

Identifiez les dépendances réelles entre les différents fichiers et réécrivez les règles en respectant ces dernières de manière à ce que le minimum de compilations indispensables soit réalisé (une des deux compilations et où l'édition de liens).
Une compilation séparée est obtenue via l'option -c du compilateur.

Réécrivez un Makefile amélioré et vérifiez les point suivants :
la commande `make` ne réalise dès à présent l'édition de liens que lorsque elle est précédée par la commande `touch boucle.o`.
la commande `make` ne recompile pas le fichier boucle.c lorsqu'elle est précédée par la commande `touch MonInclude.h`.

Chapitre 3

Automatiser la démonstration

3.1 Intro - LaboIntro 01-03 - script Demo

Titre :	Intro - LaboIntro 01-03 - script Demo
Support :	0S 42.3 Leap
Date :	08/2016

3.1.1 Énoncé

Il est temps de s'attarder sur l'écriture d'un script de démonstration bash, on le nommera *Demo*.
Soit un Makefile solution de l'exercice précédent que l'on renommera ici MonMakefile, essayons de montrer de manière automatisée que le comportement de MonMakefile correspond à notre attente.
Vous veillerez à utiliser la commande `chmod` pour donner le droit d'exécution au script Demo.
Vous pourrez dès lors lancer son exécution par la commande `./Demo`
La première ligne du script est `#!/bin/bash`. Lorsque on essaye d'exécuter ce script via la commande ./Demo dans le shell, cette ligne indique au shell la nature de ce fichier : un fichier de commandes destiné à l'interpréteur `/bin/bash` et non un exécutable.

3.1.2 Une Solution

```
#!/bin/bash
# le couple #! s'appelle "shebang"
# ailleurs, le caractère # débute un commentaire

#NOM    : Demo
#CLASSE : Intro LaboIntro 01-03
#OBJET  : réservé au makefile
#AUTEUR  : MBA 01/2016

# définitions de couleurs pour le terminal
C='\033[44m'
E='\033[32m\033[1m'
N='\033[0m'

echo "Tester l'écriture du Makefile"
echo "---------------------------------"
#La commande make a ici en paramètre le nom du makefile MonMakefile
echo -e "Exécution de ${E}make -f MonMakefile clean${N} pour commencer en de bonnes conditions :"
make -f MonMakefile clean
echo -e "${C}             --> Enter pour continuer${N}"
read # on attend un caractère au clavier (évite le défilement)
echo "---------------------------------"
echo "construction de l'exécutable pour exécuter"
echo -e "Exécution de ${E}make -f MonMakefile${N} :"
```

```
make -f MonMakefile
echo -e "${C}             --> Enter pour continuer${N}"
read # on attend un caractère au clavier (évite le défilement)
echo "---------------------------------"
echo "exécution sans construction de l'exécutable"
echo -e "Exécution de ${E}make -f MonMakefile${N} une nouvelle fois:"
make -f MonMakefile
echo -e "${C}             --> Enter pour continuer${N}"
read # on attend un caractère au clavier (évite le défilement)
echo -------------------------------------------------
echo "reconstruction et exécution après "mise à jour" de boucle.h"
echo -e "Exécution de ${E}touch boucle.h; make -f MonMakefile${N} :"
# le ; entre deux commandes indique que celles-ci seront exécutées l'une après l'autre.
touch boucle.h; make  -f MonMakefile
echo -e "${C}             --> Enter pour continuer${N}"
read # on attend un caractère au clavier (évite le défilement)
echo -------------------------------------------------
echo "reconstruction et exécution après "mise à jour" de boucle.c"
echo -e "Exécution de ${E}touch boucle.c; make -f MonMakefile${N} :"
touch boucle.c; make  -f MonMakefile
echo -e "${C}             --> Enter pour continuer${N}"
read # on attend un caractère au clavier (évite le défilement)
echo -------------------------------------------------
echo "reconstruction et exécution après "mise à jour" de boucle.o"
echo -e "Exécution de ${E}touch boucle.o; make -f MonMakefile${N} :"
touch boucle.o; make  -f MonMakefile
echo -e "${C}             --> Enter pour continuer${N}"
```

3.1.3 Commentaires

- Le script créé ici a pour seul but de montrer de manière automatisée le comportement de notre fichier MonMakefile.
- L'écriture de scripts est très intéressante dans le cas de tâches administratives récurrentes.
- Lors de nos laboratoires de SYStème, nous allons travailler un peu différemment : le script sera la démonstration de l'exercice. L'exercice qui suit celui-ci est une illustration de cette manière de travailler.
- Tous nos exercices de laboratoire utiliseront ces deux outils.

3.1.4 Voir

man bash Scripts sous linux - C. Blaess

Chapitre 4

Un premier exercice : découvrir le shell

4.1 Intro - LaboIntro 01-04 - Fonctionnement d'un shell, premiers pas

Titre :	Intro - LaboIntro 01-04 - Fonctionnement d'un shell, premiers pas
Support :	OS 42.3 Leap
Date :	08/2016

4.1.1 Énoncé

Nous allons nous servir de la commande `echo` pour comprendre comment le shell traite une ligne de commande et les wildcards.
`echo` est une commande qui affiche sur la sortie standard ce qu'elle a reçu en paramètre. Observez le comportement de la commande `echo` dans les cas suivants : `echo coucou ; echo une phrase complète > fichier ; echo coucou >> fichier ; echo * ; echo coucou | wc -c`
Dans le sous-répertoire SOURCES, Écrivez un code source en c et baptisez le Mecho.c. Il affichera sur la sortie standard ce qu'il a reçu en paramètre.
Cette commande se comporte-t-elle comme la commande `echo` dans les cas cités précédemment ?

4.1.2 Une solution

Pour résoudre cet exercice il faut écrire le code source Mecho.c, le script Demo qui exécute les différents tests demandés et le Makefile qui génère l'exécutable et lance la démonstration. Ce dernier aura également une règle *clean* appropriée.
Ainsi `cd SOURCES ; make` montre l'exercice résolu.
Les fichiers suivants se trouvent dans le répertoire SOURCES :

```
/*
NOM      : Mecho.c
CLASSE   : intro - LaboIntro 01-04
#OBJET   : réservé au makefile
AUTEUR   : J.C. Jaumain, 07/2011
*/
/*
 * ce petit code affiche les paramètres qu'on lui passe en ligne de commande, i
 * c'est d'ailleurs ce que fait la commande echo
 * */

#include <stdlib.h>
#include <stdio.h>
main ( int argc, char * argv[] )
{  int i;
   for (i=1; i<argc-1; i++){
   printf("%s ",argv[i]);  // arguments séparés d'un espace
   }
```

```
    printf("%s\n",argv[argc-1]); // et retour à la ligne
    exit(0);
}
```

```
#!/bin/bash
#NOM    : Demo
#CLASSE : Intro - LaboIntro 01-04
#OBJET  : réservé au makefile
#AUTEUR  : J.C. Jaumain, 07/2011

#codes couleur
C='\033[44m'
E='\033[32m\033[1m'
N='\033[0m'

echo "Commande echo écrite en c : Mecho"
echo "--------------------------------"
echo -e "Éxecution de ${E}echo coucou${N} :"
echo coucou
echo -e "Éxecution de ${E}./Mecho coucou${N} :"
./Mecho coucou
echo -e "${C}             --> Enter pour continuer${N}"
read
echo -e "Éxecution de ${E}echo un deux trois${N} :"
echo un deux trois
echo -e "Éxecution de ${E}./Mecho un deux trois${N} :"
./Mecho un deux trois
echo -e "${C}             --> Enter pour continuer${N}"
read
```

```
#NOM    : Makefile
#CLASSE : Intro - LaboIntro 01-04
#OBJET  : demo du programme Mecho.c
#HOWTO   : make; make clean
#AUTEUR  : MBA 08/2016

demo: Mecho Demo
    @clear
    @./Demo

Mecho: Mecho.c
    gcc -std=c99  -pedantic $< -o $@

clean:
    rm -f Mecho fichier *~
```

4.1.3 En roue libre : complétons Demo

— est-ce que, pour les cas suivants, le programme Mecho se comportera comme `echo` ?
  — `echo une phrase complète > fichier`
  — `echo coucou >> fichier`
  — `echo coucou | wc -c`
  — `echo *`
— complétez votre script Demo pour vérifier ces derniers cas

4.1.4 Commentaires

— argv[0] n'est pas affiché. En effet, en C, le premier argument est toujours le nom du programme, dans ce cas Mecho.
— Remarquons que lorsque on utilise les wildcards (`*, ?, ...`), les redirections (`>,>>`) et les pipes (`|`), ceux-ci ne sont pas affichés par Mecho. Mecho tout comme `echo` ne reçoit pas ces symboles en paramètre. Ils sont en effet filtrés et traités par le Shell.
— En écrivant notre commande Mecho il apparaît clairement que c'est le shell qui interprète ces caractères spéciaux et réalise les opérations qu'ils cachent, avant d'exécuter votre programme. De la même manière, dans la commande `ls > f` ce n'est pas la commande `ls` qui gère la redirection.

— Quand vous écrirez votre propre shell (bientôt) vous aurez appris à gérer les redirections les pipes et les wildcards tout comme le shell le fait ...
— Le but de cet exercice était de découvrir une partie de travail que le shell prend en charge lorsque une commande est appelée. Il est donc très important de mentionner cela dans ce commentaire ;-)
— Vous travaillerez de cette manière pour tous les exercices du laboratoire.

4.1.5 Voir

`man bash` pour le fonctionnement du shell bash

Chapitre 5

Makefile, règles implicites

5.1 Intro - LaboIntro 01-05 - sans Makefile

Titre :	Intro - LaboIntro 01-05 - sans Makefile
Support :	OS 42.3 Leap
Date :	01/2016

5.1.1 Énoncé

Reprenez les fichiers source de la solution de l'exercice 0101, on renommera TestMake.c TestMake2.c. Ajoutez un deuxième affichage d'une constante définie dans un deuxième fichier d'include de nom TestMake2.h (même préfixe que le code c)
Exécutez la commande `make TestMake2` sans avoir créé de fichier Makefile.
Quel comportement a la commande dans ce cas ?
Répétez la commande après avoir exécuté la commande `touch TestMake2.c`
Répétez la commande après avoir modifié la valeur de la constante MAX dans le fichier *MonInclude.h*
Répétez la commande après avoir modifié la valeur de la constante MIN dans le fichier *TestMake2.h*

```
/*
 * NOM - monInclude.h
 */
#define MAX 20
```

```
/*
  NOM    : TestMake2.h
  AUTEUR : MBA
  DATE   : 01/2016
*/

#define MIN 5
```

```
/*
NOM      : TestMake2.c
CLASSE   : Intro - LaboIntro  01-01
#OBJET  : réservé au makefile
AUTEUR   : mba 01/2016
*/
#include <stdlib.h>
#include <stdio.h>
#include "TestMake2.h"
#include "monInclude.h"
int main (){
   printf("MAX=%d\n",MAX);
   printf("MIN=%d\n",MIN);
   exit(0);
}
```

5.1.2 Commentaires

— `make` utilise ici des règles par défaut pour les compilations en c.
— `make` considère à défaut d'information qu'une cible dépendra de fichiers de même préfixe. Dans ce cas on doit générer TestMake2 et le fichier TestMake2.c est présent.
— Dans ce cas `make` fait appel au compilateur cc
— Pour l'exemple nous avons ajouté un fichier d'include de même préfixe que le fichier exécutable *TestMake2*.
— Toutefois, les liens avec des éventuels fichiers d'include ne sont pas établis par défaut, même si ceux-ci ont le même préfixe. Une modification de *TestMake2.h* ne génère pas de nouvelle compilation.
— En présence d'un Makefile, `make` essayera de générer de cette même manière, une cible pour laquelle aucune règle explicite n'est fournie.
— Nous préférerons l'écriture de règles explicites