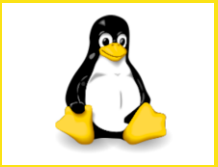
CESI_
ÉCOLE D'INGÉNIEURS

A3 FISA S3E – Langage C

TP 6 – Programmation sous Linux



LINUX - DEFINITION

Introduction

Linux est un système d'exploitation complet et libre, qui peut être utilisé en lieu et place de systèmes d'exploitation commercialisés, tels que Windows, de Microsoft. Il est accompagné de nombreux logiciels libres complémentaires, offrant un système complet aux utilisateurs.
Le système peut être utilisé sur des serveurs (LAN ou serveurs web), sur des PC ou encore sur des smartphones.
Linux est disponible en plusieurs versions, téléchargeables gratuitement sur le net, et nommées "**distributions**", telles que **Debian** ou **Ubuntu (version 20.04)**. (**f**https://ubuntu.com/download/desktop)

Les commandes de base en console Linux

1. man

- Équivalent Windows : help
- Signification : ***manual***
- Affiche les pages du manuel système. Chaque argument donné à man est généralement le nom d'un programme, d'un utilitaire, d'une fonction ou d'un fichier spécial.
- Exemples d'utilisation : **man man :** affiche les informations pour l'utilisation de man
- 'q' pour quitter.

2. ls

- Équivalent Windows : dir
- Signification : *list*
- Permet de lister un répertoire
- **cd**
- Équivalent Windows : cd
- Signification : *change directory*
- Permet de se promener dans les répertoires
- Exemples d'utilisation :
- **cd :** permet de revenir au répertoire /home/utilisateur (identique à cd ~)

- **cd - :** permet de revenir au répertoire précédent

3. mv

- Équivalent Windows : move / ren
- Signification : *move*
- Permet de déplacer ou renommer des fichiers et des répertoires

4. mkdir

- Équivalent Windows : mkdir / md
- Signification : *make directory*
- Crée un répertoire vide
- Exemples d'utilisation :

mkdir photos : Crée le répertoire *photos*

5. cp

- Équivalent Windows : copy / xcopy
- Signification : *copy*
- Permet de copier des fichiers ou des répertoires

6. cat

- Équivalent Windows : type
- Signification : *concatenate*
- Affiche le contenu d'un fichier

COMMANDE GCC

1. Introduction

GCC signifiait, au départ, **G**NU **C** **C**ompiler, mais au fur et à mesure de son évolution, il compile de plus en plus de langage (C++, objective C, fortran, ada, java), donc maintenant cet acronyme veut dire **G**NU **C**ompiler **C**ollection.

GCC est portable, il fonctionne donc aussi bien sur UNIX/Linux que sur Windows ou sur MacOS.

2. Du code source à l'exécutable

Le passage depuis un code source en C vers un programme exécutable par la machine est un processus à plusieurs étapes.

i. Compilation

Pour compiler un fichier source C, on utilise l'option –c du compilateur gcc. Si la compilation réussira, un fichier objet est généré ayant l'extension « .o ».

```
$ gcc -c main.c
$ gcc -c carre.c
```

ii. Edition de liens

La phase d'édition de liens permet de combiner plusieurs codes objets afin de générer le fichier exécutable. Pour invoquer l'éditeur de liens, il suffit d'appeler GCC avec les fichiers objet en paramètre.

```
$ gcc carre.o main.o
```

Le fichier exécutable généré portera par **défaut** le nom **« a.out** ». L'option -o permet de spécifier le nom du fichier exécutable à générer à l'issue de l'étape d'édition de liens.

```
$ gcc -o carre carre.o main.o
```

Dans ce cas, le fichier exécutable portera le nom « carre » au lieu de « a.out ».

iii. Exécution

L'exécution du programme se fait en précisant le chemin d'accès à l'exécutable comme suit :

```
$ ./carre
```

« ./ » se réfère au répertoire courant, donc « ./carre » permet de charger et exécuter le fichier exécutable « carre » se trouvant dans le répertoire courant.

3. Options d'avertissements

GCC affiche parfois des messages d'avertissements, marqués « **warning**: ». Contrairement aux erreurs, ces avertissements ne sont pas fatals et n'empêchent pas la génération du code objet. Toutefois, ils attirent l'attention du programmeur sur les structures de programmation risquées ou potentiellement fausses. En éliminant de telles constructions, vous réduirez les risques de bugs. Par exemple, le compilateur affichera un avertissement sur *un type de retour incorrect* pour la fonction *main* ou pour une fonction *non void* qui ne dispose pas d'instruction *return*.

L'invocation de l'option –*Wall* du compilateur gcc permet d'activer tous les messages d'avertissement.

```
$ gcc -Wall -c carre.c
$ gcc -Wall -c main.c
```

Pour illustrer l'impact de l'ignorance de ces avertissements, prenons le cas où on spécifie, dans le fichier *main.c*, le format du deuxième paramètre à afficher par l'instruction *printf* comme étant un *réel* au lieu d'être un *entier*.

```
/**********************    main.c    **************************/
          ...
          printf("Le carré de %d est égal à %f\n", i, resultat);
          ...
```

Bien que GCC le compile sans rien dire, le résultat affiché est erroné.

```
$ gcc -c main.c
$ gcc -o carre main.o carre.o
$ ./carre
Le carré de 1 est égal à 0.000000
Le carré de 2 est égal à 0.000000
Le carré de 3 est égal à 0.000000
Le carré de 4 est égal à 0.000000
Le carré de 5 est égal à 0.000000
```

Maintenant, si on compile ce code en utilisant l'option –Wall, le compilateur affichera l'avertissement suivant :

```
$ gcc -Wall -c main.c
main.c: In function 'main':
main.c:9: attention : format '%f' expects type 'double', but
argument 3 has type 'int'
```

Cet avertissement indique que le format *« %f »* s'attend à un paramètre de type réel, cependant le paramètre est de type entier.

PREMIER PROGRAMME EN C POUR TESTER LE COMPILATEUR GCC

1. **Editons un nouveau fichier texte dans gedit nommé toto.c :**

```
gedit toto.c
```

Voici le code source en C de ce premier programme qui se contente d'afficher "Bonjour" sur la sortie standard :

```
#include <stdio.h>
int main()
{
printf("Bonjour\n");
}
```

2. **Compilons le programme source toto.c en utilisant le compilateur gcc en ligne de commande:**

```
gcc toto.c
```

Remarques :

- Le fichier exécutable résultant de la compilation est nommé par défaut **a.out** .
- Si la commande **gcc n'a pas été trouvée.** Vous pouvez l'installer avec :

```
Sudo apt install gcc
```

3. **Exécutons le programme** a.out **:**

```
./a.out
```

EXERCICES

1. Ecrire un programme qui donne la valeur résultante de la mise en série ou en parallèle de trois résistances.

2. Ecrire un programme qui convertit des degrés Celsius en degrés Fahrenheit et en Kelvin.

   Rappel :

   Le degré Fahrenheit est égal à la 180ème partie de la différence entre la température de fusion de la glace et la température d'ébullition de l'eau à la pression atmosphérique.

   0 Kelvin correspond au zéro absolu soit –273°C.

3. Ecrire un programme qui calcule la somme des **N** premiers termes de la série harmonique :

   Série harmonique :

$$1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$$

4. Ecrire un programme qui calcule $n_1 + 2n_2 - n_3^3$, pour trois valeurs entières positives n_1, n_2, n_3, saisies au clavier. Une exécution de ce programme donnera à l'écran ce qui suit :

   Entrez trois entiers positifs : 3 1 5

   Pour n1=3, n2=1, n3=5, on calcule -120

MAKE FILE

1. Introduction

Lorsqu'on réalise un projet en le découpant en **plusieurs modules**, on constate qu'il est difficile de se rappeler des fichiers qu'il faut recompiler après une modification. L'utilitaire **make** de Linux permet **d'automatiser la compilation de gros logiciels** sous Linux.

La commande **make** permet de maintenir un programme automatiquement. Pour cela, elle utilisera un **fichier de description** nommé en général **makefile** ou **Makefile** qui contient des **dépendances** et les **actions** (commandes) à mettre en œuvre pour maintenir le programme.

À la différence d'un simple script **Shell**, **make** exécute les commandes seulement si elles sont nécessaires. Le but est d'arriver à un logiciel compilé ou installé sans nécessairement refaire toutes les étapes. En d'autres termes, au lieu de compiler la totalité du code source, **make** ne construit que le code source qui a subi des changements.

2. Syntaxe

La syntaxe de la commande **make** est la suivante :

```
$ make [-f nom fichier] [-arg optionnels] [ref..]
```

L'option **-f** indique le **nom du fichier de description**. En général, la commande est lancée **sans argument**. Dans ce cas, elle recherche un fichier de description nommé **makefile** ou **Makefile** sur le répertoire courant.

3. Contenu d'un fichier Makefile

Un fichier **Makefile** contient :

- Un ensemble de **règles (Rules)** ayant la syntaxe suivante :

```
cible: dépendances ...
<tab>  commandes
       ...
```

- La **cible (Target)** est le fichier à construire. Elle est le plus souvent un nom d'exécutable ou de fichier objet.
- Les **dépendances (Dependencies)** sont les éléments ou les fichiers sources nécessaires pour créer une cible.

- Les **commandes (Commands)** sont les actions à lancer afin de produire la cible à partir des sources. Ces actions sont lancées suivant la modification d'un de ces fichiers sources.

Donc, une règle spécifie à **Make** comment exécuter une série de commandes dans le but de construire une cible à partir des fichiers sources desquels elle dépendait.

- des **variables**, dites aussi des **macros**, qui permettent de fixer des chemins, des noms d'exécutable, …etc. Par convention, un nom de variable est écrit en majuscules (**Uppercases**).

Une variable est **définie** par son nom et sa **valeur**.

```
VAR=Valeur
```

Et, elle est référencée (utilisée) dans les cibles, les dépendances, les commandes et les autres parties du makefile comme suit :

```
$(VAR)
```

Il existe certaines variables qui sont prédéfinies et qu'on puisse les modifiées, entre autres :

- La variable CC qui contient le nom du compilateur utilisé.
- La variable CFLAGS qui contient la liste des options de compilation.
- La variable LDFLAGS qui contient la liste des options d'édition de liens.

On écrit un fichier Makefile en plaçant d'abord **les variables générales**. Ensuite, **la règle de construction de l'exécutable**. Et, on termine enfin le fichier en déclarant **toutes les autres règles.**

4. Exemple

Ci-après le fichier Makefile permettant de construire une application **power :**

```
#*****************************    Makefile    ****************************#
      OBJS=main.o carre.o cube.o
      EXEC=power
      CC=gcc
      CFLAGS=-Wall                                  Variables générales

power: $(OBJS)
      $(CC) $(OBJS) -o $(EXEC)                      La règle de construction de
                                                    l'exécutable

carre.o: carre.c carre.h
      $(CC) $(CFLAGS) -c carre.c
cube.o: cube.c cube.h
      $(CC) $(CFLAGS) -c cube.c                     Les règles de construction
main.o: main.c carre.h cube.h                       des fichiers objets
      $(CC) $(CFLAGS) -c main.c

clean:
      rm $(OBJS) $(EXEC)                            Autres règles
```

Un ensemble de **variables générales** ont été définies pour qu'elles soient utilisées

par la suite par les règles définies. La commande **make** substituera ces variables par leurs valeurs lorsque les actions sont exécutées :

- **OBJS** est une variable contenant tous les fichiers objets nécessaires à la construction de l'exécutable. Dans cet exemple, ce sont les fichiers **main.o, carre.o, et cube.o.**
- **EXEC** est une variable contenant le nom de l'exécutable à générer. Dans notre cas, cet exécutable sera nommé **power**.
- **CFLAGS** est une variable de make. Cette variable contient les options de compilation. Pour notre application, on veut que tous les avertissements soient affichés, donc il suffit d'initialiser CFLAGS à **–Wall**.

La déclaration des variables générales est suivie par une liste des **dépendances** et des **actions** associées. Le ou les fichiers qui dépendent d'autres fichiers sont précédés du caractère « : ».

- Il est clair que **power** dépend de **main.o**, **carre.o** et **cube.o** ; car on ne peut passer à l'étape d'édition de liens qu'après avoir compilé chacun des fichiers objets.
- Les fichiers objets doivent être recompilés à chaque fois que le fichier source correspondant est modifié. Ce qui fait que le fichier source doit apparaître dans les dépendances du fichier objet correspondant.
- Une modification de **carre.h** doit entraîner la recompilation des deux fichiers objets **carre.o** et **main.o** car les deux fichiers source incluent ce fichier d'entête. De même, une modification de **cube.h** doit entraîner la recompilation des deux fichiers objets **cube.o** et **main.o**.

La règle **clean** ne possède aucune dépendance. Seules des actions sont définies. En exécutant les **fichiers objets** et l'**exécutable**, produits lors de la construction de l'application avec la commande make, seront **supprimés**.

```
$ make clean
rm *.o power
$
```

5. Exécution d'un Makefile

Par **convention**, le fichier exécuté par make est **nommé makefile** ou **Makefile**. Dans ce cas, l'exécution de la commande :

```
$ make
gcc -Wall -c main.c
gcc -Wall -c carre.c
gcc -Wall -c cube.c
gcc main.o carre.o cube.o -o power
$
```

Permet de produire la **première** cible trouvée dans le fichier Makefile. Si on veut **construire** une **cible particulière**, il suffit de lancer make avec le nom de la cible. Par exemple, la commande :

```
$ make carre.o
```

Permet de construire la cible **carre.o** du fichier Makefile.

Si on donne au fichier exécuté par make un **nom différent** de Makefile, il suffit d'utiliser l'option **–f** de cette commande afin de spécifier le nom du fichier. Par exemple, si notre fichier s'appelle **projet**, alors la commande à exécuter pour donner l'ordre à make de traiter ce fichier est :

```
$ make -f projet
```

6. Compiler avec Makefile sous Ubuntu

i. Créer un Makefile :

Touch Makefile

ii. Créez un fichier d'en-tête **sub.h** et entrez le code :

Float x2x(int a, int b) ;

iii. Entrez le code dans le Makefile :

```
main: main.o sub.o
	gcc -o main main.o sub.o
main.o:main.c sub.h
	gcc -c main.c
sub.o:sub.c sub.h
	gcc -c sub.c
clean:
	rm *.o main
```

iv. Utilisez la commande make pour afficher le résultat :

./main

EXERCICES

1. 1.1 Ecrire une fonction « ***Carré*** » chargée de calculer le carré d'un nombre réel
   1.2 Utiliser cette fonction dans un programme Main qui saisit un nombre, appelle la fonction pour l'élever au carré et affiche le résultat.
   1.3 Changer votre programme pour calculer aussi le cube d'un nombre réel.
   Une exécution de ce programme donnera à l'écran ce qui suit :

   Entrez un nombre réel :3
   Le carré de 3 vaut : 9
   Le cube de 3 vaut : 27

2. Instructions de contrôle :
   2.1 Ecrire une fonction **void *testCaractere*()** qui teste si un caractère saisi au clavier est un chiffre, un caractère en minuscule ou un caractère en majuscule.
   2.2 Ecrire une fonction **void *tableMultiplication*()** qui demande "Quelle table de multiplication voulez-vous, tapez 0 (z_ero) pour sortir ?". Si le caractère saisi est compris entre '1' et '9' alors on fera afficher la table correspondante puis on réitèrera

le processus, sinon on affichera "ce n'est pas dans les possibilités du programme, recommencez !" et on réitérera le processus.

3. Les tableaux en C
   3.1 Dans un programme principal, déclarer un tableau de 10 entiers.
   3.2 Initialiser le tableau avec des valeurs aléatoires comprises entre 0 et 20.
   3.3 Implémenter et tester à chaque fois les fonctions suivantes :
   - Fonction ***afficher*** qui permet d'afficher les éléments du tableau.
   - Fonction ***calculerMoyenne*** qui permet de calculer la moyenne des éléments du tableau.
   - Fonction ***trouverMin*** qui permet de trouver la valeur minimum du tableau.
   - Fonction ***inverserTableau*** qui inverse les éléments du tableau (penser à une variante qui n'utilise pas de tableau supplémentaire).