Piscine 2020

Lucie Le Briquer

30 mars 2020

Table des matières

1 Shell

1.1 Quelques commandes

- `man -f` : recherche dans tous les registres
- existence de `rmdir` au lieu de `rm -rf`
- `wc` : wildcard, nombre de lignes, mots, bytes
- `env` affiche l'ensemble des variables environnements, une variable créée n'est pas commune à plusieurs shell, il faut l'export si on veut pouvoir l'utiliser
- `more` : pagine pour un long fichier, et on peut utiliser `/` pour rechercher comme dans `vim`
- `head -n` : récupère les `n` premières lignes
- `head -c` : récupère les `n` premiers charactères
- `tail` : fonctionne de la même façon
- `grep` : recherche un motif, `-v` pour afficher les lignes qui ne contiennent pas, `-i` pour ignorer la casse, `-n` pour afficher aussi les numéros lignes contenant le motif
- `sed 's/motif/remplacement/g'` : remplace toutes les occurences de `motif` par `remplacement`
- `chmod` pour changer les droits
- `touch` créé un fichier, `touch -t 07151997 fichier` pour modifier la date
- `for i in ‘seq 0 9‘ ; do ... ; done`
- `ln -h` pour créer un hardlink, `ln -n` pour un lien symbolique
- `ls -t` pour trier par ordre de dernière modification
- `find . \( -type d -o -type f \)` trouver tous les dossiers et fichiers réguliers

1.2 Sorties et écriture de fichiers

- `stdin` est l'entrée standard de la commande lancée, une commande peut écrire sur deux types de sorties `stdout` (sortie standard) et `stderr` (sortie d'erreur)
- | branche la sortie standard sur l'entrée standard
- `>` pour rediriger la sortie standard
- `2>` pour rediriger la sortie d'erreur
- les deux peuvent s'enchaîner
- on peut aussi faire `grep bl < fichier`
- `>>` ajoute au fichier plutôt que de l'écraser
- `<<` écrire jusqu'au mot choisi :

```
cat << FIN
> ...
> ...
> FIN
```

- `2<&1` pour rediriger la sortie d'erreur sur la sortie standard, pour des raisons techniques pour tout rediriger dans un fichier `ls -l > fichier 2>&1`

1.3 Remarques

- `'  '` inhibe tout contrairement à `"  "` pour lequel `"$truc"` affiche `machin`
- fichier magic pour écrire des conditions recherchées sur un fichier

  ```
  offset  type    motif   message
  ```

  compiler ce fichier avec `file -C -m fichier_magique` puis tester le fichier avec `file -m magic_file monfichier`

2 Git

2.1 Commandes

- `git diff` : affiche les différences entre la dernière photo et l'état actuel des fichiers
- `git branch name` : créer une branche
- `git branch -a` : liste les branches
- `git branch -d name` : supprime une branche
- `git checkout name` : va sur une branche donnée
- `git checkout -b name` : créer une branche et va dessus
- `git checkout -- fichier` : remet le fichier à sa valeur lors du dernier commit
- `git commit --amend` : écrase le dernier commit
- `git ls-file` : affiche tous les fichiers suivis par git, `-i` pour les ignorés, `--exclude-standard` pour les exclus standards, `-o` pour les untracked

3 Langage C

3.1 Types

- `char` : 1 octet $(-128, 127)$, par définition signé
- `int` : 4 octets, par définition signé
- `float` : 4 octets mais moins de précision
- `double` : 8 octets
- pour `char` et `int` on peut préciser `unsigned` pour gagner un bit
- `long int` : 8 octets
- `short int` : 2 octets
- définition des tableaux : `int tab[10]`
- `static` pour que la fonction n'existe que dans son fichier
- `const` variables constantes (utile dans le développement à plusieurs)

3.2 Opérateurs

- `/` division entière, `%` modulo
- `i++` et `++i`, différence sur l'ordre de l'opération

3.3 Opérateurs binaires

- `&` = et
- | = ou
- `^=` xor (seulement si un est vrai)
- `~` négation
- `!` si autre que 0 met 0, sinon autre chose (indéfini)
- `<<2` décale de 2 bit et rajoute des 0
- `>>` idem mais si signed et positif `1.......` `>>3` donne `111.....`

3.4 Comparaison

- `==` égalité
- `!=` différent
- `>=` supérieur ou égal etc
- `&&` et, faignant si first partie fausse n'évalue pas la deuxième
- || ou, faignant aussi

3.5 Conditions

- `if` :

```
if  (condition)
{
    commande;
}
else if ...
```

- `switch` :

```
switch(a)
{
    case 1:
        commande1;
        break;
    case 2:
        commande2;
        break;
    ...
}
```

- opérateur ternaire `test ? si oui : si non`

3.6 Commandes spéciales

- `break` : sortie de boucle
- `continue` : repartir en haut de la boucle sans faire la suite des instructions
- `goto` : déclaration d'un label et go à ce label

```
    code
label1 :
    suite du code
    goto label1
```

3.7 Pointeurs

- `int *ptr`, `ptr = &a` (adresse de `a`)
- `*ptr = valeur` : modifie la valeur de la variable à l'adresse pointée par `ptr`
- `ptr + 1` : décalage de 4 sur l'adresse en hexa, pour se décaler d'`int` en `int`
- `*(ptr + 1)` possible
- il est possible de mettre `ptr = 0`, ça signifie qu'il ne pointe sur rien
- `void *ptr` pour créer un pointeur qui pointe vers un objet de type quelconque

3.8 Tableaux et chaînes de caractères

- un tableau est en fait un pointeur sur un `int` qui donne la first case puis `[i]` décale de `i` dans la mémoire. Ainsi, `*tab` donne le premier élément, `*(tab + 1)` le deuxième, etc.
- une chaîne de caractère n'existe pas réellement, c'est une suite de caractères se terminant par le caractère `\0`, `*str` est le premier caractère

3.9 Main

```
int main(int argc, char **argv)
```

```
./a.out arg1 arg2 arg3
```

- `argc` est le nombre d'arguments passé à la fonction lors de l'appel de `a.out`
- `argv` est un *tableau* de chaîne de caractères contenant les arguments
- attention `argv[0]` est le nom de l'exécutable appelé, ici `a.out`

3.10 Malloc

- `#include <stdlib.h>` pour l'utiliser
- permet d'allouer de la mémoire
- prototypage : `void *malloc(size_t size)`
- pour allouer de la place pour 4 `int` il vaut mieux faire `malloc(4*sizeof(int))` pour que cela fonctionne sur n'importe quel processeur
- il faut rendre la mémoire dès que possible avec `free`
- si l'allocation échoue, malloc renvoie le pointeur null

4 Compilation

4.1 Généralités

Il y a 3 phases de compilation :

- préprocess : ajout de plein de codes via les includes, `cpp fichier.c` pour affichier les codes introduits
- compilation : création d'un fichier objet `.o` à partir du `.c` via par exemple `gcc -c main.c`
- link des différents fichiers objets, dans un `.o` il n'y a pas d'erreur si la fonction appelée n'existe pas, c'est lors du link qu'il va la faire correspondre à une fonction dans un autre `.o`

Comprendre cette décomposition permet de recompiler uniquement les fichiers modifiés et donc gagner du temps. Au moment du link seule l'existence est vérifiée mais pas le typage !

4.2 Include

- `#include <...>` : directives de bases du compilateur
- `#include "..."` : fichier spécifique, ou l'inclure avec `gcc -i`, ce genre d'include copie textuellement le fichier à cet endroit là, en général pas une bonne idée les includes de `.c`
- `.h` : fichier contenant les prototypes de fonctions, inclus dans le main et dans le fichier, permet de faire gaffe à la correspondance des types, et propage les changements

4.3 Define et variables

- `#define VAR blbl` : remplace *textuellement* à l'endroit appelé, convention de nommer ces variables en majuscules
- `#define LOL(x) ...` : on peut définir ces variables avec des paramètres, souvent dangereux et illisible
- `gcc -DVAR=90 main.c` : on peut définir les variables directement en appelant gcc

4.4 Préprocesseur de contrôle : if, else

- `#if`, `else` : marche comme en LateX
- `#ifdef`, `#ifndef` : pour vérifier la définition ou non d'une variable, permet de debug mais aussi de protéger les `.h` qui sont souvent inclus plusieurs fois

```
#ifndef __FCT_H__
#define __FCT_H__
code
#endif
```

5 Définition de type et structures

5.1 Définition de type

- `typedef type nom_du_type` : s'utilise comme prévu
- attention `typedef` a un scope

5.2 Structure

Comme les structures dans n'importe quel autre langage :

```
struct  s_point
{
    int     a;
    int     b;
    char    *name;
}
```

- un objet `s` de type `struct s_point` a alors trois *attributs*, on y accède avec `s.a`, `s.b`, `s.name`
- il est plus simple de définir un nom pour la structure, on écrit alors directement :

```
typedef struct
{
    ...
} name_struct;
```

5.3 Énumération

```
enum    e_list
{
    val1 = start,
    val2,
    ...
    val15
};
```

- `enum` créer toutes ces variables en tant que constantes, ce sont des `int`, de base met $0, 1, ..., 14$, mais on peut préciser un start, ou même définir la valeur de n'importe quelle variable
- ces constantes sont globales à tout le scope

6 Makefile

Permet d'organiser la compilation dans un projet et d'éviter les compilations inutiles

- le fichier Makefile contient des règles qui s'exécutent via `make rule_name`
- une règle est une suite de commande shell
- `make` exécute la première règle définie
- la déclaration des variables se fait via `VAR = file1 file2` puis l'appellation via `${VAR}`
- une règle se définit : `rule_name:`
- on peut attribuer des dépendances aux règles, elle dépendent alors des fichiers indiqués

Exemple type de Makefile :

```
SRCS = main.c fct.c
OBJS = ${SRCS.c=.o}

all: ${OBJS}
    code

clean: dependancies
    code

...
```

- avec `${OBJS}`, make comprend tout seul qu'il doit recompiler uniquement les fichiers qui ont été modifiés depuis
- on peut mettre un nom de fichier en nom de règle, si les dépendances ont été modifiées plus récemment que le fichier l'appel de la règle exécute les instructions
- `.c.o` est une règle cachée de Makefile, c'est la compilation des `.c` en `.o`, on peut la redéfinir pour choisir le compilateur, les flags etc de la manière suivante :

  ```
  .c.o:
      ${CC} ${FLAGS} -c $< -o ${<:.c=.o}
  ```

  en définissant les variables `CC` et `FLAGS` avant
- toujours mettre une règle `all`
- les règles pouvant être des noms de fichiers il peut y avoir une ambiguïté, par exemple si le fichier `all` existe déjà, il est alors possible à la fin du Makefile de définir via `.PHONY` les règles qui ne doivent pas être interprétées comme des noms de fichiers
- si on veut dans une variable récupérer tous les fichiers `.c` par exemple, il faut faire `SRCS = $(wildcard *.c)`

7 Bibliothèques

- la création d'une bibliotèque se fait de la manière suivante :

```
ar rc libNomDeLaLibrairie.a file1.o file2.o etc
```

- il faut donc générer les `.o` des `.c` dont on veut ajouter les fonctions à la librairie puis générérer celle-ci via la commande précédente
- les librairies peuvent vite devenir très grandes et leur parcours très long, il est recommandé d'exécuter la commande suivante qui va créer un index de la librairie :

```
ranlib libNomDeLaLibrairie.a
```

- la création d'une librairie permet aussi de partager ses fonctions à d'autres utilisateurs sans dévoiler le code source
- pour utiliser une librairie dans un projet il faut alors le compiler en indiquant l'endroit où se trouve la librairie :

```
gcc ... -LCheminDeLaLibrairie -lNomDeLaLibrairie
```

8 Pour l'examen

- `kinit pseudo`, password demandé
- `examshell` à lancer, un README se trouve sur l'ordi
- bibliothèque utiles : `stdio.h` et `unistd.h`