Informatique 1ère année Enseirb

J.-L. Bienvenu

Ecole Nationale Supérieure d'Electronique, Informatique et Radiocommunications de Bordeaux

Partie 1

Unix et le shell `bash`

Introduction

Unix est un système d'exploitation (en anglais O.S.=operating system) apparu vers 1970 (voir historique)
A quoi sert un système d'exploitation ?

Un ordinateur est construit à partir de composants électroniques et mécaniques (microprocesseur, mémoire, disque dur...). Cependant, il est nécessaire d'intercaler entre l'ordinateur et ses utilisateurs une couche logicielle, appelée système d'exploitation, dont le rôle est multiple :

- Masquer la complexité

  Par exemple, un disque dur ne permet que d'accéder à des blocs de données contigus et de taille fixe. Pour l'utilisateur, il est pratique de cacher ceci derrière une organisation hiérarchique en fichiers et en répertoires. C'est au système d'exploitation de relier chaque fichier aux blocs qu'il occupe sur le disque dur. Plus généralement, le système d'exploitation permet de cacher le fonctionnement réel des composants de l'ordinateur derrière une interface plus simple.

- Permettre le partage des ressources

  Le microprocesseur ne peut faire qu'une chose à la fois. Cependant, dans le cas de l'utilisation de programmes interactifs, le microprocesseur passe plus de 99 % de son temps à attendre que l'utilisateur humain lui indique quoi faire. Sur un ordinateur utilisé par plusieurs personnes à la fois, ces temps d'attente peuvent être efficacement utilisés pour permettre l'exécution concurrente de plusieurs programmes.

- Protéger l'accès aux ressources

  Puisqu'il permet à plusieurs programmes d'utiliser le microprocesseur en concurrence, le système d'exploitation doit protéger chaque programme des autres. En particulier, il gère la protection de la mémoire et des fichiers, de telle sorte qu'aucune application ne puisse modifier les données d'une autre application ou du système d'exploitation lui-même.

Pratiquement, le système d'exploitation est le premier logiciel lancé au démarrage de l'ordinateur. Il initialise les composants matériels et en verrouille l'accès. Tous les autres programmes devront passer par l'interface qu'offre le système d'exploitation pour accéder au matériel.

Cette abstraction des accès permet de simplifier la conception des programmes utilisateurs, qui n'ont pas à se soucier du fonctionnement réel des composants de l'ordinateur, avec qui seul le système d'exploitation dialogue. (modèle dit "en couches")

1.1 Commandes

Pour utiliser votre machine, vous pouvez

- soit utiliser la souris, les icones, les menus.. (usage qui ne sera pas décrit ici)
- soit utiliser l'interprète de commandes, qui est un programme qui exécute (si possible) les commandes que vous tapez au clavier.

Il est souvent tentant d'utiliser la première méthode, car elle constitue une solution de facilité. Elle est cependant beaucoup moins puissante dans de nombreuses situations, et l'aprentissage des commandes sera rentable si on utilise fréquemment Unix.

Les commandes sont analysées par un programme : l'interprète de commandes (souvent appelé "shell"). Il en existe plusieurs (sh, ch, tcsh, bash, ksh). Les différences entre ces différents interprètes n'apparaît pas pour les usages les plus simples.
Le shell *bash* est l'un des plus puissants et le plus répandu: c'est le shell par défaut de Linux et Mac OS X, et il est disponible dans la distribution Cygnus pour Windows. (c'est aussi shell par défaut de l'Enseirb).

Pour exécuter une commande, il suffit de taper son nom, suivi éventuellement d'autres informations.

La syntaxe d'une commande est la suivante :

<commande> [-<options>] [<argument> ....]

Le sens de cette "grammaire" primitive est le suivant

- *commande* est l'equivalent d'un ordre (verbe à l'infinitif)
- *option* indique comment effectuer l'action (complément circonstanciel)
- *argument* indique à quoi s'applique la commande (complément d'objet direct)

conventions

- les mots encadrés par < > sont à remplacer par un mot de votre choix.
- les crochets *[ ]* indiquent que le contenu est facultatif.

- les points de suspension ... indiquent que l'item précédent peut être répété.

Quelques exemples (utiles)

- `clear` (effacer l'écran)
- man <commande>
  indique que la commande `man` (affichant les informations sur une commande) nécessite un argument.
- lp -d<imprimante> <fichier>...
  indique que la commande `lp` (permettant d'imprimer les fichiers au format "postscript") necessite une option (-d) et au moins un argument.
  Ex. : `lp -dmigou planning.ps infos.ps` imprime sur l'imprimante *migou* les fichiers `planning.ps` et `infos.ps`
  par contre les commandes `lp, lp infos.ps lp -d, lp -d infos.ps` sont incorrectes.

Remarques :

- Les options sont généralement cumulables (`ls -lFa` équivaut à `ls -l -F -a`)
  En cas de doute, la commande **man** <*commande*> affiche une information sur <commande>

- Pour écrire plusieurs commandes sur la même ligne, il faut les séparer par un point-virgule.

Lorsqu'une commande se termine, elle renvoie au shell un entier (en général 0 si tout s'est bien passé et une autre valeur sinon). Cette valeur n'est pas affichée, mais on peut la consulter avec : `echo $?`. On peut aussi la tester (voir programmation)

Complétion et édition

L'interprète `bash` permet de compléter les noms des fichiers avec la touche *Tab*. Si le terminal emet un bip, cela signifie qu'il n'y a pas de complétion possible ou qu'il y a plusieurs complétions. Il suffit de taper 2 fois *Tabulation*, si plusieurs possibilités existent, elles seront affichées.
Pour re-taper une commande déjà utilisée, il suffit d'utiliser la flèche vers le haut pour remonter dans l'historique des commandes. Si la commande est loin dans l'historique, on peut la rechercher avec *Ctrl-R* et taper une suite de caractères de la commande recherchée.
Pour se déplacer dans une ligne de commande, il est possible d'utiliser les flèches gauche et droite, ou *Ctrl-A* pour aller au début, ou *Ctrl-E* pour aller à la fin. (notez que ces commandes servent également dans l'éditeur *emacs*, il est donc rentable de les mémoriser).

1.2 Processus

Un processus est un programme en cours d'exécution. Unix étant multi-tâches, un même programme peut donner lieu à plusieurs processus s'exécutant simultanément.

La commande `ps` permet d'afficher les processus.

```
~ > ps
  PID TTY      TIME CMD
18614 pts/34   0:01 bash
18648 pts/34   0:28 emacs
 1779 pts/34   0:07 firefox
 4108 pts/34   0:00 ps
```

Unix attribue à chaque processus un numéro unique : le PID, qui apparaît dans la première colonne. Pour avoir plus d'informations, on utilise l'option `-f`

```
~ > ps -f
    UID   PID  PPID  C    STIME TTY      TIME CMD
   toto 18614 18612  0 09:39:31 pts/34   0:01 -bash
   toto  1779 18614  0 13:55:29 pts/34   0:07 firefox
   toto   604 18614  0 14:31:18 pts/34   0:00 ps -f
   toto 18648 18614  0 09:40:39 pts/34   0:26 emacs -nw unix.tex
```

La deuxième colonne (PPID) donne le numéro du processus parent, c'est à dire le processus qui l'a créé.
En fait, la commande `ps` n'affiche ici que les processus descendant du processus 18614. Pour voir tous les processus de l'utilsateur, il faut utiliser l'option `-u` suivie du nom de cet utilisateur.

Lorsqu'une commande est exécutée, `bash` attend qu'elle se termine pour réafficher l'invite, ce qui signifie que vous pouvez taper une nouvelle commande. Si une tâche nécessite beaucoup de temps, ou si elle ne se termine pas seule (ce qui est le cas des processus qui ne se termine qu'à la demande de l'utilisateur, comme thunderbird ou firefox), il faut terminer la commande par le symbole `&`

Il arrive qu'un processus reste bloqué. Pour mettre fin à un processus, il suffit d'obtenir son PID avec `ps` et de taper `kill -kill <PID>`

1.3 Répertoires

La mémoire permanente est organisée de façon hiérarchique en répertoires (ou dossiers).
Le répertoire principal est noté : /
Celui-ci contient des sous-répertoires, par exemple : *bin, etc, home, tmp, usr* qui existent en général sur tous les systèmes Unix.
Chaque utilisateur possède son propre répertoire (qui sera appelé par la suite répertoire personnel ou répertoire d'accueil). Il a le droit (c'est même fortement conseillé) de créer des sous-répertoires pour classer de façon organisée ses données personnelles.
A l'Enseirb, par exemple, les répertoires des élèves de 1ère année sont dans un répertoire *e1* qui est lui-même un sous-répertoire de *net*

A un instant donné, chaque utilisateur est "situé" dans un répertoire appelé répertoire courant

En dehors de votre répertoire personnel, vous pouvez vous placer dans un répertoire quelconque (à condition que vous en ayez l'autorisation), mais vous ne pourrez pas y écrire de données. (exception : /tmp)

Chemins

Un répertoire est désigné de façon unique par la suite des noms des répertoires à parcourir pour y accéder.

- le répertoire de plus haut niveau (appelé racine) est noté /
- un chemin est dit absolu s'il commence à la racine
  exemple : /net/e1/toto/Programmation/Projet
- un chemin est dit relatif s'il commence au répertoire courant
  exemple : (si le répertoire courant est /net/e1/toto) Programmation/Projet
  Un chemin relatif ne commence donc pas par /.

Notations

- le répertoire hiérarchiquement supérieur au répertoire courant est noté ..
- le répertoire courant est noté .

  Le shell permet en plus les notations suivantes :
- Le chemin absolu du répertoire personnel peut être abrégé en : ~
- Le chemin absolu du répertoire personnel de l'utilisateur toto peut être abrégé en : ~toto

Commandes agissant sur les répertoires

- **cd** syntaxe : cd *[- | <répertoire>]*
  cd *[<répertoire>]* change le répertoire courant.
  cd - ramène au précedent répertoire visité.
  cd ramène au répertoire d'accueil.

- **pushd, popd** pushd fonctionne comme cd, en empilant le répertoire qui vient d'être quitté. On peut revenir en tapant popd

- **mkdir** syntaxe : mkdir *<répertoire> ....* : crée les répertoires spécifiés.

- **rmdir**
  syntaxe : rmdir *<répertoire> ....* : détruit les répertoires spécifiés. (à condition qu'ils soient vides.)

Nommage des répertoires

Les noms des répertoires utilisent, de préf'erence, des majuscules, des minuscules, des chiffres et quelques symboles (tels que . + = ou - (qu'il vaut mieux éviter de mettre en tête)).
La plupart des autres symboles sont déconseillés. Néanmoins, on peut les utiliser en les préfixant par \ ou en les encadrant par des apostrophes.
Exemple : `mkdir Mes\ Documents` ou `mkdir 'Mes Documents'`

1.4 Fichiers sur disque

Les données et les programmes sont stockés de façon permanente sur disque. Un fichier peut être vu comme étant constitué d'un nom et d'un bloc de données.
Un même bloc de donnés peut être référencé par plusieurs noms. La commande `ls -l` affiche dans la $2^{\text{ème}}$ colonne le nombre de liens, c'est à dire le nombre de fois où la bloc du fichier listé est référencé

Le nom suit les mêmes règles que les noms des répertoires, et la notion de chemin est aussi la même.

Unix n'effectue aucun contrôle sur le contenu des fichiers, il vous appartient donc de leur donner un nom adapté à leur contenu. Par convention, chaque type de fichier possède une extension (ou suffixe), c'est à dire un point et une suite de lettres caractéristique.
Exemples de suffixes

- `.ps` ou `.eps` : fichier Postscript
- `.pdf` (portable document format) : document destiné à acrobat
- `.c` : fichier source du langage C
- `.mws` : fichier Maple

Remarques :

1. Les fichiers exécutables n'ont pas besoin de suffixe.
2. La commande `file` donne le type du fichier (ou essaie de le faire en examinant son contenu)

Commandes agissant sur les fichiers

- `ls [<fichier ou répertoire> ...]` affiche les répertoires et fichiers passés en argument. Sans argument, ls équivaut à `ls .` (affichage du répertoire courant)
- `rm <fichier> ...` : efface les fichiers passés en argument. Pour être exact, rm décrémente le nombre de liens et détruit le bloc de données si ce nombre est 0. Un confirmation est demandée uniquement pour les fichiers n'offrant pas le droit d'écriture.
  Options :

  `-i` demande une confirmation pour chaque fichier

  `-f` ne demande jamais de confirmation

  `-R` permet d'effacer des répertoires et leur contenu Attention : les fichiers effacés ne sont pas récupérables (sauf par l'administrateur qui peut le retrouver dans un archivage si ces fichiers ne sont pas trop récents).

- `cp <fichier> ... <destination>` : copie les fichiers sur la destination, qui doit être un répertoire si le nombre des fichiers est supérieur à 1.
  Exemples :

  `cp fichier1 fichier2` : crée une copie de fichier1 sous le nom fichier2

  `cp fichier1 fichier2 Archive` crée une copie de fichier1 et fichier 2 dans le répertoire Archive.
  Options :

  `-i` demande une confirmation

  `-R` permet de copier des répertoires et leur contenu

- `ln <fichier> ... <destination>` : comme cp, mais le bloc de chaque fichier n'est pas dupliqué. Il s'agit de la création d'un lien.
  Options :

  `-s` crée un lien symbolique ,c'est à dire un raccourci. C'est la seule méthode pour créer un lien sur un répertoire.

- `mv <objet> ... <destination>` (ou `mv <<objet1> <objet2>` où <objet> désigne un fichier ou un répertoire) déplace dans le répertoire destination; s'il n'y a que 2 arguments, le 1erest renommé en le 2ème s'il n'existe pas.

Jokers (dans le shell)

Il est possible d'utiliser des caractères spéciaux pour désigner des fichiers (ou des répertoires)

- le caractère * remplace toute suite de caractères (même vide)
- le caractère ? remplace tout caractère
- la séquence $[motif]$ remplace tout caractère correspondant au motif

exemple: (avec la commande ls)
Si un répertoire contient les fichiers *liste.c liste.o liste.h file.c file.h lire.s lire.o lire.aux fileAttente.c fileAttente assemble make Makefile listeFichiers*

```
 ~ > ls *.c
liste.c  file.c fileAttente.c

 ~ > ls *e.?
liste.c liste.o liste.h file.c file.h lire.s lire.o

~ > ls *fi*
file.c file.h fileAttente.c fileAttente listefichiers

~ > ls ????.?
```

```
file.c file.h lire.s lire.o

~ > ls [afs]*
file.c file.h fileAttente.c fileAttente assemble

~ > ls *[A-Z]*
fileAttente.c fileAttente  Makefile listeFichiers
```

1.5 Droits d'accès

1.5.1 Droits standard

Chaque fichier et répertoire possède des droits d'accès. Ces droits existent pour 3 catégories d'utilisateurs : le propriétaire, le groupe du propriétaire et le reste des utilisateurs. Il existe 3 droits : r, w et x.
Le tabeau donne la signification de ces droits.

	droits correspondant à r	droits correspondant à w	droits correspondant à x
fichier	consulter, copier	modifier, supprimer	exécuter
répertoire	lister	supprimer, créer un fichier	accéder

l'option `-l` de la commande `ls` affiche dans la première colonne les droits de chaque fichier ou répertoire.

```
 ~ > ls -l
total 202
-rw-r--r--   1 bozo     e1          2174 sept 26  2008 aide-memoire.txt
drwx---r-x   2 bozo     e1           512 oct   9  2008 Desktop
drwxr-xr-x   5 bozo     e1           512 fevr 12 10:17 Documents
drwx------   2 bozo     e1           512 dec  15 15:48 Downloads
-rw-r--r--   1 bozo     e1         46972 dec   2 12:16 bozo.jpg
drwx------   2 bozo     e1           512 avr   3 13:33 imap
drwx---r-x   6 bozo     e1           512 dec  17  1999 office97
drwx---r-x   3 bozo     e1           512 avr   1 15:07 personnel
drwxr-xr-x   1 bozo     e1           512 mars 17 10:56 Programmation
-rw-r--r--   1 bozo     e1         39819 janv 16 10:49 promo.jpg
drwxr-xr-x   4 bozo     e1           512 nov  18 13:29 public_html
drwx---r-x   2 bozo     e1           512 mars 16 18:29 tmp
drwxr-xr-x   3 bozo     e1           512 janv 16 10:12 TpMatlab
drwx---r-x   5 bozo     e1           512 oct   9  2008 windows
```

Le 1ercaractère indique le type de l'entrée (- pour un fichier, d pour un répertoire, l pour un lien sybolique (raccourci)). les 3 suivants donnent les droits du propriétaire, puis viennent les droits du groupe et ceux des autres. Pour des raisons pratiques, vous ne devez retirer les droits pour le groupe et les autres que si vous avez une bonne raison. Les comptes de l'Enseirb ne sont pas là pour héberger vos données confidentielles (même s'il ne vous est pas interdit d'en stocker quelques unes pour lesquelles vous conserverez un accés privé). Les enseignants doivent pouvoir accéder à votre travail ...

Modification des droits

la commande `chmod` modifie les droits des fichiers ou répertoires passés en argument.
La syntaxe est *chmod [-R] <droits> <objet> ...*

où les droits sont définis par un nombre octal de 3 chiffres obtenus additivement à partir de r=4, w=2, x=1.
L'option -R permet d'appliquer ces droits à tout le contenu des répertoires.

exemple

`chmod 754 memo` attribue les droits rwx (7=4+2+1) à l'utilisateur, rx (5=4+0+1) au groupe et r (4=4+0+0) aux autres sur memo.

1.6 Redirections et tubes

1.6.1 Notion de fichier dans Unix

Pour Unix, un fichier est un dispositif quelconque capable d'émettre ou de recevoir des caractères (ou les deux). Un fichier sur disque est donc un fichier, mais les périphériques sont aussi considéres comme des fichiers. Lorsque un programme, comme ls, produit des caractères, ceux-ci apparaissent sur le terminal, mais le programme n'est pas conçu pour afficher spécifiquement sur ce terminal. En fait, la majorité des programmes (en tout cas ceux qui s'exécutent à l'intérieur d'un terminal) écrivent leur résultats sur un "tuyau" appelé sortie standard. (et lisent sur l'entrée standard) Ces entrées et sorties sont, par défaut, associées au clavier (en entrée) et à l'écran (en sortie), mais sont susceptibles d'être redirigées.

1.6.2 Redirections

Si vous taper la commande `ls /net/e1` , vous verrez la liste des utilisateurs de e1. Si vous tapez `ls /net/e1 > liste` , rien ne s'affiche. Les caractères ont été détournés vers le fichier liste.
Si on exécute la commande `cat`, rien ne se passe. En fait, cat lit l'entrée standard, et attend donc que l'utilisateur entre des données. A chaque nouvelle ligne entrée, cat se contente de la réafficher sur la sortie standard,, ce qui ne présente aucun intérêt .... Par contre, si on redirige l'entrée standard à partir du fichier *liste* : `cat < liste`, celà permet d'afficher le contenu de *liste*. On peut combiner les 2 redirections; par exemple : `sort -r <liste >listeInversée` .
Si on effectue une redirection vers un fichier existant, le contenu est détruit. Pour ajouter à ce qui existe, il faut rediriger avec `>>`. Exemple:

```
~ > ls /net/e1 >liste
~ > ls /net/e2 >>liste
~ > sort <liste
```

Remarques:

1. les commandes qui lisent les données sur leur entrée standard et écrivent sur leur sortie standard sont appelés *filtres*

2. la commande `cat < fichier` a exactement le même effet que `cat fichier`, mais dans le $2^{\text{ème}}$ cas, ce n'est pas l'entrée standard qui est lue. Le processus cat ouvre lui-même un autre "tuyau" pour lire les données du fichier

3. Quelques fichiers spéciaux :

   /dev/tty est spécifiquement associé au terminal (pas de redirection possible)

   /dev/null reçoit des caractères, mais ne les restitue pas. On l'utilise pour éliminer une sortie indésirable.

   /dev/urandom fournit des octets pseudo-aléatoires.

Quelques filtres:

- **cat**
- **more, less** affiche son entrée standard en arrêtant l'affichage à chaque page.
- **head** affiche les $n$premières lignes de son entrée standard. item **tail** comme head, mais pour les n dernières lignes.
- **tr** remplace sur l'entrée standard chaque occurence d'un caractère par un autre.
- **uniq** supprime les lignes successives identiques.
- **sort** trie son entrée standard.
- **grep** selectionne les lignes qui contiennent une expression.
- **sed** remplace sur chaque ligne une expression par une autre.
- **awk** Filtre aux multiples possibilités; on l'utilisera pour sélectionner des mots sur chaque ligne.

Les processus ont une $2^{\text{ème}}$ sortie appelée sortie d'erreur standard. Pour la rediriger, on utilise la notation `2>`. Pour rediriger simultanément les 2 sorties, il faut utiliser `>&`. Enfin, pour rediriger la sortie standard vers la sortie d'erreur standard, il faut utiliser `1>&2` (et pour la réciproque : `2>&1`)
Exemple : `echo "fichier inconnu" 1>&2`

1.6.3 Tubes

Lorsque on exécute les commandes `ls /net/e1 >liste; sort -r <liste`, le fichier `liste` ne sert que d'intermédiaire entre les deux commandes. Il existe une méthode pour transmettre directement la sortie de `ls` à l'entrée de `sort`. Il suffit d'exécuter : `ls /net/e1 | sort -r` (et on peut relier de cette façon plusieurs commandes)

1.7 Expansion et substitution de commande

1.7.1 Expansion des accolades

Bash remplace une expression de la forme {mot1,mot2, ... motN} par mot1, puis mot2, ..., puis motN. (attention,il ne doit pas y avoir d'espaces autour des virgules)
Exemples
{bas,atten,mo,ac}tion est expansé en : bastion attention motion action.
{,re,de}pos{a,ai,as,a} est expansé en : posai posas posa reposai reposas reposa deposai deposas deposa
{{a,e}{i,o{{r,t},u}}} est expansé en : ali alor alot alou avi avor avot avou eli elor elot elou evi evor evot evou

Ce mécanisme d'expansion n'est pas lié aux noms des fichiers, mais il peut être utilisé pour les opérations sur les fichiers:

```
~ > cp ~toto/Programmation/Piles/{test,}pile.{h,cpp} .
```

est plus rapide que :

```
~ > cp ~toto/Programmation/Piles/testpile.h ~toto/Programmation/Piles/testpile.cpp
~toto/Programmation/Piles/pile.h ~toto/Programmation/Piles/pile.cpp .
```

et plus précis que

```
~ > cp ~toto/Programmation/Piles/*pile.* .
```

qui copie bien les fichiers concernés, mais aussi par exemple pile.o ou horripile.jpg

```
~ > mkdir -p Prog/{Piles,Files,Arbres}/{Source,Include,Lib}
```

est plus rapide que :

```
~ > mkdir -p Prog/Piles/Source Prog/Piles/Include Prog/Piles/Lib  Prog/Files/Source
Prog/Files/Include Prog/Files/Lib Prog/Arbres/Source Prog/Arbres/Include Prog/Arbres/Lib
```

A pertir de la version 3 de bash, on peut aussi expanser des intervalles de caractères (ex: {a..d} en a b c d)

1.7.2 substitution de commande

La substitution de commande transforme en une chaîne de caractères la sortie standard d'une commande. Cette opération ne peut être confondue avec une redirection ou un tube car le résultat n'est transmis ni à un fichier, ni à l'entrée standard d'un processus.
la syntaxe est : *$(<commande>)* ou *‘<commande>‘* Exemple 1:
La commande `who |cut -c -8|sort|uniq` affiche les noms des utilisateurs connectés à la machine. Supposons que l'on veuille écrire un courrier à ces utilisateurs, il suffit de taper: `mail $(who |cut -c -8|sort|uniq)`

Exemple 2:
On peut le trouver dans le fichier `.xsession`. La commande `uname` donnant le nom du système d'exploitation est utilisée pour choisir le système graphique à exécuter.

```
if [ ‘uname‘ = "Linux" ] ;then

     exec /usr/bin/gnome-session
else
   case ‘uname -a | cut -f 3 -d’ ’‘ in
5.9) exec gnome-session  > $HOME/.gnome-errors 2>&1;;
5.8) exec twm > $HOME/.gnome-errors 2>&1;;
5.8) if [ -e /usr/bin/gnome-session ]; then
                exec gnome-session > $HOME/.gnome-errors 2>&1
             else
                 exec twm  > $HOME/.gnome-errors 2>&1
             fi;;
5.7) exec twm  > $HOME/.twm-errors 2>&1;;
*) exec twm  > $HOME/.twm-errors 2>&1;;
   esac
fi
```

On peut en trouver un autre exemple dans le fichier `.bash_export`. La commande `arch` est utilisée pour déterminer le type de machine (architecture) et décider de la valeur de certaines variables.

1.8 Personnalisation

1.8.1 Fichiers de démarrage

Lors du démarrage d'une session, des fichiers sont exécutés.

`.xsession` : démarre le gestionnaire graphique et lance des applications, en particulier deux fenêtres `xterm`. Vous pouvez y ajouter d'autres commandes que vous voulez exécuter au démarrage.

`.bash_profile` est exécuté à chaque ouverture d'une session `bash` (par exemple au lancement de `xterm`). Il exécute les tâches qui doivent être exćutées une seule fois, par exemple la définition des variables d'environnement (cf. plus loin)
Remarque : les définitions de variables sont faites dans `.bash_export`, qui est invoqué par `.bash_profile`

`.bashrc` exécute les tâches qui doivent être exécutées à chaque lancement de `bash` , comme les définitions de fonctions et d'alias.
Remarque : ces définitions sont faites dans `.bash_alias`, qui est invoqué par `.bashrc`

1.8.2 Variables

L'assignation d'une variable se fait par :

$$<variable>=<valeur>$$

(sans espaces de part et d'autre du =), et son évaluation par:

$$\$\{<valeur>\} \text{ ou } \$<valeur>$$

Pour désassigner une variable :

$$<variable>= \text{ ou } unset <variable>$$

Il existe des variables définies par l'administrateur. On peut les lister avec la commande `set`. On y trouve des variables comme : DISPLAY (nom du terminal), EUID (numéro de l'utilisateur) , HOME (répertoire de l'utilisateur) , HOSTNAME (nom de la machine), USER, LOGNAME (nom de l'utilisateur). Ces variables ne doivent pas être modifiées, à l'exception de PS1 (voir Annexe : Changer le prompt)

La variable PATH a une importance particulière. Elle contient la liste des chemins dans lesquels une commande peut être trouvée. Vous pouvez ajouter des répertoires, changer l'ordre s'il ne vous convient pas, mais il est déconseillé d'en supprimer.

Pour éviter de détruire une variable importante, il est conseillé de nommer ses propres variables en minuscules.

protection
Si vous voulez assigner une liste de mots à une variable, il faut encadrer cette liste par des guillemets (protection faible) ou de apostrophes (protection forte). La différence principale se trouve dans l'évaluation des variables : elles sont évaluées avec la protection faible et pas avec la protection forte.
Exemple :

```
 ~ > nom=toto
 ~ > echo "bonjour $nom"
bonjour toto
 ~ > echo 'bonjour $nom'
bonjour $nom
```

Visibilité :
Une variable n'est définie que dans le shell où elle a été définie. Pour transmettre cette variable aux processus fils, il faut déclarer :

<variable>=<valeur>; export <variable>
ou *export <variable>=<valeur>*

1.8.3 Alias

Lorsqu'on utilise souvent un commande (et surtout si elle est longue), il est possible de lui donner un nom appelé alias. On déclare un alias par:

alias <nom>=<commande>

exemple : `alias progC='cd  /Informatique/Programmation/C'`

Comme les variables, les alias n'existent que dans le shell où ils sont définis. En revanche, il n'existe aucun mécanisme analogue à `export`. Il faut donc les évaluer à chaque fois (donc les placer dans `.bashrc` ou (mieux) dans `.bash_alias`).
Sans argument, `alias` affiche tous les alias définis. Pour supprimer un alias, utiliser `unalias`

1.8.4 Scripts

Dans le cas où les commandes sont plus complexes, il est possible de les placer dans un fichier. (il est conseillé d'écrire en première ligne `#!/bin/bash` pour être sûr qu'il sera bien traité par bash.
Pour l'exécuter, on peut

-a- soit taper `source <fichier>` ou `. <fichier>`

-b- soit lui donner le droit x et l'exécuter en tapant son nom.

Les scripts ont quelques inconvénients.

il faut en général donner le nom complet de la commande, car le répertoire où elle se trouve n'est pas spécifié dans la variable PATH.

le comportement n'est pas le même suivant la méthode d'exécution. Dans le cas de la méthode -b-, les commandes sonr exécutées dans un sous-shell et il est possibles que les commandes exécutées n'aient aucun effet (ex : définir une variable, changer de répertoire).
Si par exemple, votre script contient une commande `exit`, la méthode -b- ne posera pas de problème, mais la méthode -a- provoquera la fermeture de votre terminal.

1.8.5 Fonctions

Les fonctions sont une bonne alternative aux scripts. Une fois évaluée, une fonction est mémorisée par l'interprète et ne nécessite pas de localisation. Une fonction f se définit par

f() { ... } ou *function f* { ... }

Dans une fonction

- on peut accéder aux arguments avec les notations *$1,$2 ... $10* .
- la liste des arguments est notée *$@* .
- la commande `shift` décale le arguments d'un rang vers la gauche.
- on peut retourner une valeur au shell avec la commande `return`
- la récursivité est possible

Comme les alias, les fonctions doivent être définies à chaque exécution de bash.

Pour supprimer une fonction, utiliser `unset -f`

1.8.6 Priorité

lorsqu'on tape une commande, `bash` recherche et exécute dans l'ordre suivant :

1. alias
2. fonction
3. commande interne (comme echo, alias ,cd ...)
4. les commandes sur disque dans l'ordre défini par la variable PATH.

Remarque : lorsqu'une commande est précédée de son chemin, elle est exécutée sans que le shell fasse la recherche.

1.9 Programmation en bash

1.9.1 Calcul numérique

On peut évaluer des expressions numériques entières à l'intérieur de `$(( ))`. Si la valeur de la variable n'est pas un entier, le résultat vaut 0. Les opérateurs utilisables sont ceux du langage C, plus `**` pour le calcul de puissance.
Exemple : `x=$(( x + 1 + $1 ))`

1.9.2 Le choix "binaire"

```
if <condition>
  then
    <instructions>

  elif <condition>
    then  <instructions>
    ..............
    else
      <instructions>
fi
```

Types de conditions utilisables :

a- résultat de commande
on teste le code retour d'une commande avec la convention : 0 vaut vrai, toute autre valeur vaut faux

exemple :

```
if gcc prog.c
  then ./a.out
fi
```

On peut relier les commandes avec les connecteurs logiques : && (et) , || (ou) , ! (non). Un équivalent de ce qui précède est : `gcc prog.c && ./a.out`

Dans les autres types de test, les conditions doivent être entourées de crochets. et les connecteurs logiques sont : -a (et) , -o (ou), ! (non)

b- tests sur les chaînes (la condition doit etre entourée de [ ])

opérateurs définis :
= , != , -z (-z ¡chaine¿ vaut vrai si ¡chaine¿ est vide), -n (-n ¡chaine¿ vaut vrai si ¡chaine¿ est non vide)

exemple :

```
if [ -z "$1" ]
  then echo "Usage : $0 <argument>"
fi
```

Notez l'usage des guillemets pour délimiter `$1` ; s'ils sont absents et qu'il n'y a pas de paramètre, bash ne voit que `if [-z ]`, ce qui est une erreur de syntaxe.

c- tests sur les fichiers
opérateurs définis : -e (existe), -s (existe et est non vide), -f (est un fichier), -d (est un repertoire)

exemple :

```
if [ -s pense_bete.txt ]
  then
    cat pense_bete.txt
fi
```

d- tests sur les entiers

opérateurs définis : -eq (=), -ne (!=), -gt (¿), -ge (¿=), -lt (¡), -le (¡=)

exemple :

```
if [ $n -ne 0   -a  $X -gt $Y ]
    .....
```

Remarque : toute variable ne contenant pas une valeur correspondant à un nombre est considérée comme nulle.

1.9.3 Le choix multiple

```
case  <expression> in
  <modele> )
    <instructions>;;
  <modele> )
    <instructions>;;

  ............

esac
```

Remarque : cette commande ne permet de distinguer que des formes de chaînes de caractères, elle n'est donc pas équivalente au `switch` du langage C.

exemple :

```
compile()
{
    case $fichier in
     *.c)     gcc $fichier;;
     *.c++)   g++ $fichier;;
     *)       echo "type inconnu";;
    esac
}
```

exemple 2 :
une fonction `f` attend un argument numérique

```
f()
{
  case "\$1" in
   ""      ) nombre=100 #valeur par defaut
   *[!0-9]*) echo "usage : $0 nombre";return;;
   *       ) nombre=$1
  esac
}
```

1.9.4 Le parcours de liste

```
for <variable> [in <liste>]
  do
   <instructions>
   ....
 done
```

Dans une fonction, si aucune liste n'est précisée, la valeur par defaut est : in $@
Remarque : comme en C, on peut interrompre le traitement avec `break`, ou le reprendre en passant à l'élement suivant avec `continue`

1.9.5 La répétition

```
while  <condition>                    until  <condition>
   do                                      do
     <instructions>                          <instructions>
     ...                                     ...
   done                                    done
```

où <condition> a été décrite ci-dessus.

1.9.6 Menu

Il est possible d'effectuer une saisie avec un choix multiple (qui est proposé en boucle) Cette saisie est en général associée à `case` pour traiter les différentes options.

Exemple :

```
menu ()
{
   select commande in courrier internet   emacs  quitter
     do
       case  $commande in
          courrier) thunderbird  \&;;
          internet) firefox \& ;;
          emacs   ) emacs  \&;;
          quitter ) break;;
       esac
    done
}
```

1.10 Principales commandes

Le résumé suivant ne donne que les options les plus utiles. Pour une information complète sur une commande, utiliser

```
man <commande>
```

1.10.1 Commandes les plus utiles

- **a2ps** lit son entré standard ou le(s) fichier(s) en argumentet transforme le teste en Postscript. L'utilité pricipale du Postscript étant l'impression, on précise alors l'option `-P<imprimante>`
  exemple

  ```
  ~ > a2ps hello.c -Ptethys
  ```

  Quelques options:

  -1 : impression en mode portrait

  -nH : pas d'en-tête

  -nn : pas de numérotation

  -ns : pas de cadre

- **awk, nawk** applique à chaque ligne le traitement spécifié entre les accolades. Les mots de chaque ligne sont accessibles par la notation $1, $2 etc.. et chaque ligne est acessible par $0. Ce traitement peut être précédé d'une phase préliminaire (BEGIN) et une phase postérieure (END)
  exemples

  ```
  ~ > ls -l |awk '{print $1,$9}'
  ~ > ls -l |awk 'BEGIN{t=0}{t+=$5;printf "%8s %s\n", $5,$9}END{printf "total %d\n",t}'
  ```

  awk permet d'utiliser des variables et de faire des calculs exemple

  ```
  ~ > ls -l |awk 'BEGIN{total=0}{printf "%8s %s\n", $5,$9}END{printf "\n%s octests\n", total}
  ```

  Quelques options:

  -v (seulement nawk et /usr/xpf4/bin/awk) permet de transmettre des valeurs

  -F permet de définir le séparateur de mots (espace par défaut) exemple

  ```
  ~ > nawk -F: -v total=0 -v chemins=$PATH '{  .... }'
  ```

- **cat** lit sur l'entree standard ou sur les fichiers passés en argument et écrit sur la sortie standard.
  exemple

```
~ > cat < ~/.bash_export
~ > cat demo.h demo.c
```

- **chmod** change les droits des répertoires et fichiers passés en argument

  exemple

```
~ >  chmod 755  Public/projet.pdf photo.jpg
```

- **echo** affiche ses arguments sur la sortie standard.

  Options:

  -n empêche le passage à la ligne

  -e accepte les séquences Ansi (comme \n)

- **find** recherche des fichiers à partir du répertoire spécifié.

  exemple

```
~ > find ~/Programmation -name liste.c
~ > find ~ -name core -exec rm '{} \;'
~ > find ~/Prive -name '*.jpg'
```

- **grep** Cherche un motif (mot ou expression régulière dans un fichier ou un ensemble de fichiers. Le premier argument est le motif, les suivants les fichiers de recherche. Si aucun fichier n'est spécifié, grep lit l'entrée standard.

- **head** affiche les n premières lignes de son entrée standard.

- **less** comme more, en mieux

- **ls** liste les fichiers et les répertoires passés en argument. Options:

  -a -A : affiche les noms commençant par . (sauf . et .. pour -A)

  -d : si l'argument est un répertoire, affiche son nom au lieu du contenu

  -l -o : affiche des informations en plus du nom

  -R : affiche le contenu de chaque répertoire en explorant les sous-répertoires.

  -t : trie par date de modification (par défaut, ls trie par nom)

  -- color : utilise des couleurs différentes pour distinguer les répertoires, les exécutables et les liens symboliques.

- **mkdir** crée les répertoires passés en argument

- **mail, mailx** permet d'envoyer un courriel (passe en mode interactif s'il n'y apas d'arguments). exemple

```
~ > echo "rendez-vous a 14h"|mailx -s RV vincent francois paul
~ > mailx -s "derniere version du projet" toto < projet.c
```

- **more** comme cat, mais l'affichage est arrêté à chaque page et quelques commandes permettent un fonctionnement interactif

  entrée : ligne suivante

  espace : page suivante

  Ctrl-b : page precédente

  / : recherche une expression

  q : quitter

- **passwd** permet de changer son mot de passe.

- **printf** affichage formatté inspiré de la fonction C
  exemple

```
~ > printf "%12s %6d\n" $USER $EUID
```

- **read** lit une ligne sur l'entrée standard et la stocke dans la variable passée en argument (`REPLY` par défaut). L'option `-a` permet la lecture d'un tableau.

- **rm** efface les fichiers passés en argument.
  Options

  `-R` permet d'effacer complètement un réprtoire.

  `-i` (interactif) : demande une confirmation.

  `-f` efface, y compris les fichiers ne possédant pas le droit `w` sans confirmation.

- **rmdir** efface les répertoires passés en argument à condition qu'ils soient vides.

- **sed** Effectue de la substitition de texte sur l'entrée standard ou les fichiers passés en argument. Le 1erargument décrit la substitution avec le format suivant : 's<sep><motif1><sep><motif2><sep>[g]'
  où <sep> désigne un caractère de séparation (au choix de l'utilisateur), <motif1> la chaîne à rechercher et <motif2> la chaîne à de remplacement. Si le g est absent, le remplacement ne concerne que la 1ère occurence de chaque ligne, toutes les occurences sinon.

  exemple

```
~ > pwd |sed 's|/| |g'
~ > cat monfichier |sed 's/^ *$//'
~ > echo decoupage |sed 's#.# &#g'
```

- **sort** trie par ligne son l'entrée standard ou les fichiers passés en argument

  Options:

  -r inverse l'ordre du tri

  -kn trie selon la colonne numéro n

- **source .** exécute des commandes du shell contenues dans le fichier spécifié en argument.

- **su** permet à un autre utilsateur de se connecter pendant la session d'un autre.
  exemple

```
~ > su - toto
```

- tail affiche les n dernières lignes de son entrée standard.
- **tr** remplace sur l'entrée standard chaque occurence d'un caractère par un autre.
- **type** : `type <commande>` indique ce qui sera exécuté si on tape cette commande (alias, fonction, commande interne ou commande externe) l'option `-all` permet de connaître toutes les possibilités.

1.10.2 Autres commandes

- **cal** Donne le calendrier du mois passé en argument

  exemple

```
~ > cal 10
~ > cal 10 2024
```

- **crypt** encrypte/décrypte l'entrée en utilisant une clé de cryptage (choisie librement par l'utilisateur). Le décryptage se fait avec la même clé.

  exemple

```
~ > crypt fw4g8- < prpjet.c >projet.c.enigma
~ > crypt fw4g8- <projet.c.enigma
```

- **cut** selectionne sur l'entré standard ou les fichiers passés en argument, des zones de texte repérés par des indices de début et/ou de fin.

  exemple

```
~ > ls -l| cut -c 1-9,32,42,54-
```

- **date** affiche la date et l'heure.
- **gzip/gunzip** compresse/decompresse les fichiers passés en argument
- **dirname** affiche le nom du répertoire contenant le fichier ou répertoire dont le chemin est passé en argument exemple

```
~ > basename /net/e1/toto/Projet/demo.c
/net/e1/toto/Projet/
```

- **head** `head -<n>` lit les `<n>` premières lignes de son entrée standard
- **nl** comme *cat*, avec numérotation des lignes

- **od** affiche le contenu d'un fichier considéré comme un fichier binaire.

  Options:

  `-N<items>` limite la lecture à N items

  `-a` considère les items comme des caractères.

  `-t <format><taille>` considère les items comme des nombres

  `<format>` peut être d(d'écimal) s(décimal signé) o (octal) x (hexa) f (float)

  `<taille>` est le nombre d'octets de chaque item : 1,2,4,8 pour u,d,o et x; 4, 8 ou 16 pour f

- **pwd** Affiche la chemin courant.

- **script** provoque l'enregistrement de toutes les commandes et leur résultat dans un fichier (dont le nom par défaut est typescript) L'enregistrement se termine par *exit* . L'option -a permet d'ajouter à un fichier existant.

- **tail** `tail -<n>` lit les `<n>` dernières lignes de son entrée standard

- **tar** : Crée un fichier d'archive à partir d'un ensemble de fichiers (ou reciproquement selon la commande) Les commandes sont : c=créer, f=écrire le résultat dans un fichier (conseillé) v=afficher la liste des fichiers traités x=extraire z=mode compressé

  exemple

```
 ~ > tar cvzf *.c
 ~ > tar cvf projet.tar  RepertoireProjet
 ~ > tar xvf projet.tar
```

- **touch** : Positionne sur l'heure actuelle la date des fichiers passés en argument (s'ils n'existent pas, ils sont créés avec une taille nulle.

- **wc** compte les caractères, les mots et les lignes sur l'entrée standard ou les fichiers passés en argument.

- **which** : which <commande> affiche le chemin absolu de la 1ère commande trouvée dans l'ordre des répertoires de la variable PATH. (ne traite ni alias, ni fonctions, ni commandes internes)

  exemple

```
 ~ > which ls
ls is /opt/jumble/bin/ls
```

1.11 On ne sait jamais, ça peut toujours servir

- Protection
  On peut empêcher la modification d'une variable en la déclarant *readonly*. Exemple :`readonly home='/net/e1/toto'`

- RANDOM
  la "variable" RANDOM donne une évaluation à chaque fois différente qui est un nombre pseudo-aléatoire entre 0 et 65535.

- Evaluation conditionnelle
  l'expression `$var:-valeur` vaut `$var` si var est définie et `valeur` sinon

- Assignation conditionnelle
  l'expression `$var:=valeur` a la même valeur que précedemment, mais la variable `var` se voit assigner cette valeur si elle n'est pas définie.

- Tableaux
  Il est possible d'utiliser des tableaux sans qu'il soit nécessaire de les déclarer ou de les dimensionner. La syntaxe d'assignation est identique à celle du C; pour l'évaluation, il est nécessaire d'utiliser les accolades.
  Exemple:

```
 ~ > t[0]=10;  t[1]=20;  t[3]=50
 ~ > echo ${t[0]}
10
 ~
```

  Il n'est pas obligatoire de remplir un tableau de façon continue. L'ensemble du tableau peut être obtenu par `${t[*]}` ou `${t[@]}`. Dans le ^{er}cas, il est considéré comme une chaîne unique, dans le $2^{ème}$, comme une suite de chaînes. Le nombre d'élements définis est donné indifféremment par `${#t[*]}` ou `${#t[*]}`.

- Découpage de chaînes

  `${<var>#<motif>}` retourne la valeur de la variable privée du motif s'il figure au début (et sa valeur initiale sinon).

  `${<var>##<motif>}` retourne la valeur de la variable privée de toutes les premières occurences motif s'il figure au début (et sa valeur initiale sinon).

  `${<var>%<motif>}` retourne la valeur de la variable privée du motif s'il figure à la fin (et sa valeur initiale sinon).

  `${<var>##<motif>}` retourne la valeur de la variable privée de toutes les dernières occurences motif s'il figure à la fin (et sa valeur initiale sinon).

  Exemple : si `chemin` vaut `/net/e1/toto/windows` et

  `${chemin#/}` vaut `net/e1/toto/windows`

  `${chemin#/*/}` vaut `e1/toto/windows`

  `${chemin##*/}` vaut `windows`

  `${chemin#/*/}` vaut `windows`

  `${chemin%/}` vaut `/net/e1/toto/windows`

  `${chemin%/*}` vaut `/net/e1/toto`

  `${chemin%%/*}` est vide

- Substitution de processus
  Alors que la substitution de commande permet de considérer la sortie d'un processus comme une chaîne de caractères, la substitution de processus permet de la considérer comme un fichier. La notation utilisée est `<(commande)` ou `>(commande)`

  Par exemple, pour trier deux fichiers $f1$ et $f2$, on utilise `sort f1 f2`. Si on veut trier les résultats des commandes `la /net/e1` et `/net/e2`,il suffit d'utiliser : `sort <(/bin/ls /net/e1) <(/bin/ls /net/e2)`. Si on dispose d'une commande `cmd` qui écrit des nombres en binaire dans un fichier dont le nom est donné en argument,on peut visualiser le résultat directement avec `od` en tapant `cmd >(od)`

- Traitement des arguments dans une fonction
  Le shell ne dispose pas de fonctions permettant d'examiner individuellement les caractères d'un chaîne, il est donc difficile de reconnaitre les options (commençant par un tiret. La commande `getopts` permet cette reconnaissance (qui est simple si les options sont en premier)
  La syntaxe est : *getopts <chaine> <var>*
  la chaine de caractères contient chaque lettre correspondant à une option. si l'argument est une option faisant partie de `chaine`, `getopts` assigne la lettre reconnue à `<var>` et retourne 0 ; si l'argument n'est pas une option, `getopts` retourne 1 ; si l'argument ne fait pas partie de `chaine`, un message d'erreur est émis. Exemple :

```
....
while getopts "ab" option
 do
   case $option
     a) .... ;;
     b) .... ;;
  esac
done
....
```

  On peut spécifier les options avec `-a -b` ou `-ab`. Certaines options peuvent avoir des compléments (ex: `-n 123` ou `-n123`. Il faut alors, dans la liste des caractères d'option, faire suivre la lettre correspondante de *:*. le complément d'option est accessible par `$OPTARG` (pour une option utilisant plusieurs lettres (ex: `-color` il faut vérifier que l'option est `-c` et que `$OPTARG` vaut `olor` !).
  `getopts` utilise (et modifie) également une variable `OPTIND` indiquant la position de l'argument sur la ligne de commande. Une fois les options traitées, il faut utiliser `OPTIND` pour décaler les arguments pour continuer le traitement.

```
....
unset OPTIND
while getopts "abn:c:" option
 do
   case $option
     a) .... ;;
     b) .... ;;
     n) val=$OPTARG;;
     c) if [ "$OPTARG" = "olor" ]
         then  ...
        ;;
```

```
  esac
done
shift $((OPTIND - 1))
....
```

Partie 2

Outils

2.1 Commandes emacs

Déplacements

- **Ctl f** Un caractère à droite
- **Ctl b** Un caractère à gauche
- **Esc f** Un mot à droite
- **Esc b** Un mot à gauche
- **Ctl a** Début de ligne
- **Ctl e** Fin de ligne
- **Ctl n** Une ligne vers le bas
- **Ctl p** Une ligne vers le haut
- **Ctl v** Une fenêtre vers le bas
- **Esc v** Une fenêtre vers le haut
- **Esc** $\prec$ Début du texte
- **Esc** $\succ$ Fin du texte

Destruction, copie

- **Del** Détruit le caractère précédent
- **Ctl d** Détruit le caractère courant
- **Esc d** Détruit la fin du mot courant
- **Ctl k** Coupe la fin de la ligne
- **Ctl espace** Marque le début d'une zone
- **Ctl w** Coupe la zone se terminant à la position du curseur
- **Esc w** Copie la zone se terminant à la position du curseur
- **Esc h** Marque le paragraphe courant
- **Ctl x h** Marque le texte entier
- **Ctl y** Insère la dernière zone coupée ou copiée
- **Ctl x Ctl o** Supprime les lignes vides

Buffers

- **Ctl x b** Sélectionne un autre buffer
- **Ctl x Ctl b** Donne la liste des buffers
- **Ctl x k** Ferme un buffer

Recherche, remplacement

- **Ctl s** Recherche incrémentale vers le bas
- **Ctl r** Recherche incrémentale vers le haut
  Esc interrompt la recherche
  Ctl s ou **Ctl r** continue la recherche
- **Esc %** Remplacement
  Esc interrompt le remplacement
  espace remplace l'occurence
  . remplace l'occurence et termine
  Del ne remplace pas l'occurence
  ! remplace toutes les occurences

Fenêtres multiples

- **Ctl x 2** Partage horizontal
- **Ctl x 3** Partage vertical
- **Ctl x 1** Ferme les autres fenêtres
- **Ctl x 0** Ferme la fenêtre courante
- **Ctl x 0** Ferme la fenêtre courante
- **Ctl x o** Passe à la fenêtre suivante
- **Ctl x 4 b** Selectionne un buffer dans une nouvelle fenêtre
- **Ctl x 4 f** Charge un fichier dans une nouvelle fenêtre

Accés au disque

- **Ctl x Ctl s** Sauvegarde le fichier sur disque
- **Ctl x Ctl f** Charge un fichier
- **Ctl x i** Insère un fichier dans la fenêtre courante
- **Ctl x Ctl w** Ecrit dans un fichier la fenêtre courante

Divers

- **Ctl x Ctl d** Liste le répertoire dans un buffer
- **Esc x !** Execute une commande du Shell
- **Esc x shell** Execute un Shell
- **Ctl x Ctl c** Quitte **emacs**
- **Ctl z** Suspend **emacs**
- **Ctl g** Interrompt la dernière commande
- **Ctl x u** Annule la dernière modification
- **Ctl l** Rafraîchit l'affichage

2.2 La compilation

Le compilateur utilisé est `gcc`. Il s'agit en fait d'un ensemble de p rogrammes enchaînant prétraitement, compilation proprement dite, assemblage et édition de liens (ou link). Par abus de langage, on appellera *compilation* l'ensemble de ces opérations et *compilation seule* la suite des 3 premières (obtenue avec l'option `-c`).

Dans les premiers exercices, le programme sera contenu dans un seul fichier (`premier.c` par exemple). Pour le compiler, tapez `gcc premier.c`. Si la compilation se passe sans erreur, recompilez avec (au moins) l'option `-Wall` pour voir les éventuels avertissements. Ne pensez pas que vous pouvez ignorer ces avertissements: ils révèlent en général des erreurs de conception. L'erreur la plus fréquente est : `parse error before ..` . Cela signifie seulement que ce qui est écrit ne respecte pas la syntaxe du langage C. C'est donc à vous de rechercher quelle faute de syntaxe a été commise.

Considérons le programme `betisier.c` suivant :

```
 1
 2  int carre(int x){
 3    printf("%d,x*x)
 4  }
 5
 6  factorielle( x)
 7    return x*factorielle(x-1);
 8  }
 9
10  int puissance(int n, int k){
11    for (puiss=1, k>=0, k--);
12      puiss=puiss*n;
13  }
14
15  main(){
16    int n,p;
17    printf("n=? ");Scanf(%d,n);
18    carre(n);
19    printf("%d %d\n",factorielle(n), puissance(n,p));
20  }
21
```

On peut classer les défauts qu'il contient en plusieurs catégories :

- 1 - Erreurs

  La compilation produit :

```
bug.c:3:10: missing terminating " character
bug.c: In function ‘carre’:
bug.c:4: error: parse error before ’}’ token
bug.c: At top level:
bug.c:7: error: parse error before "return"
bug.c: In function ‘puissance’:
bug.c:11: error: ‘puiss’ undeclared (first use in this function)
bug.c:11: error: (Each undeclared identifier is reported only once
bug.c:11: error: for each function it appears in.)
bug.c:11: error: parse error before ’)’ token
bug.c: In function ‘main’:
bug.c:17: error: parse error before ’%’ token
```

L'erreur de la ligne 3 est explicite: il manque les guillemets fermants.

A la ligne 4, le compilateur s'est arrêté sur l'accolade qui termine la fonction (qui commence bien par une accolade), ce n'est donc pas celle-ci la source de l'erreur. Il faut alors revenir en arrière et découvrir que le point-virgule a été oublié a la fin de la ligne précédente.

A la ligne 7, il faut aussi revenir en arrière et s'apercevoir que l'accolade ouvrante a été oubliée.

Ligne 11, le compilateur signale explicitement que la variable `puiss` n'a pas été déclarée.

Ligne 11, à nouveau, le compilateur s'arrête sur la parenthèse fermante. Il fallait utiliser les points-virgules au lieu des virgules pour séparer les trois clauses du `for`.

Ligne 17, les guillemets ont été oubliés autour du `%d`

Une fois ces erreurs rectifiées, la compilation produit le message:

```
Undefined first referenced
 symbol        in file
Scanf                                /var/tmp//ccw6jD2X.o
ld: fatal: Symbol referencing errors. No output written to a.out
collect2: ld returned 1 exit status
```

Notez que ce message ne ressemble pas aux autres: il n'indique aucun numéro de ligne et est écrit sur 2 colonnes (il faut lire : `Undefined symbol Scanf first referenced in file /var/tmp//ccw6jD2X.o`
En fait c'est l'éditeur de liens qui signale une erreur. Il lui est impossible de créer un exécutable car il ne trouve pas de fonction `Scanf` ni dans le programme, ni dans la bibliothèque C. Il s'agit en fait de la fonction `scanf`
Une fois cette dernière rectification faite, gcc n'affiche plus d'erreur. On exécute alors a.out.

```
 ~ > ./a.out
n=? 5
Segmentation Fault (core dumped)
```

. . . ce qui prouve qu'il reste des problèmes.

- 2 - Erreurs provoquant un avertissement

On compile alors en demandant l'affichage des avertissements : `gcc -Wall -std=c99 betisier.c`. On obtient :

```
bug.rep1.c: In function ‘carre’:
bug.rep1.c:3: warning: implicit declaration of function ‘printf’
bug.rep1.c:4: warning: control reaches end of non-void function
bug.rep1.c: At top level:
bug.rep1.c:6: warning: return type defaults to ‘int’
bug.rep1.c: In function ‘factorielle’:
bug.rep1.c:6: warning: type of ‘x’ defaults to ‘int’
bug.rep1.c: In function ‘puissance’:
bug.rep1.c:13: warning: control reaches end of non-void function
bug.rep1.c: At top level:
bug.rep1.c:15: warning: return type defaults to ‘int’
bug.rep1.c: In function ‘main’:
bug.rep1.c:17: warning: implicit declaration of function ‘scanf’
bug.rep1.c:17: warning: format argument is not a pointer (arg 2)
```

Les avertissements des lignes 3 et 17 indiquent que le compilateur manque d'informations pour savoir si on a correctement utilisé les fonctions `printf` et `scanf` (il ne les connaît pas car les fonctions ne font pas partie de la définition du langage C. Il faut donc inclure le fichier contenant les déclarations (également appelées en-têtes ou prototypes) des fonctions scanf et printf. Il faut ajouter au début : `#include <stdio.h>`

l'avertissement `.. type defaults to int` indique qu'en l'absence d'information sur le type, le compilateur suppose qu'il s'agit du type `int`, ce qui est le cas pour le paramètre `x` de la fonction `factorielle` et la valeur retournée par les fonctions `factorielle` et `main`, mais il est fortement conseillé de déclarer explicitement le type.

Le message `control reaches end of non-void function` indique qu'une fonction dont le type retourné n'est pas `void` ne retourne rien. En fait, on pourrait modifier `carre` pour qu'elle retourne `void`, mais il est indispensable d'ajouter `return puiss;` à la fin de `puissance` (sinon, le résultat est imprévisible). Le cas de `main` est un peu particulier : elle ne peut que retourner un entier (qui est le code retour transmis au shell) ; on peut donc ajouter `return 0;` à la fin, mais l'absence de cette instruction ne pose aucun problème.

Le dernier avertissement indique un problème plus grave. Un entier (non initialisé) a été utilisé comme un pointeur. Cela provoque une erreur d'adressage en mémoire qui peut être à l'origine du message : `Segmentation Fault`

- 3 - Erreurs de programmation

Après avoir corrigé les causes des avertissements, on re-compile et ré-exécute; le problème persiste. Il faut donc étudier le code pour en comprendre l'origine. On s'aperçoit que:

la fonction `f` n'a pas de cas terminal, son exécution donne lieu à une récursion "infinie".

il y a un point-virgule à la fin de la ligne 11, ce qui termine l'instruction `for` et la ligne 12 ne fait pas partie de la répétition.

la variable `p` de `main` est bien déclarée, mais pas initialisée, la fonction `puissance` effectue donc le calcul avec un exposant imprévisible.

- 4 - Erreurs d'algorithme

le nombre d'itérations a été mal calculé. entre k et 0, $k+1$ itérations sont effectuées

- 5 - Défauts de conception

  la fonction `carre` calcule effectivement le carré de son argument mais ne retourne aucune valeur. On ne peut pas l'utiliser pour calculer `carre(2)+carre(3)`. Il faut séparer les calculs des entrées et des sorties.

Une fois tous les problèmes réglés, compiler avec l'option `-std=c99` pour vérifier que le code satisfait à la norme du langage C.

2.3 Make

2.3.1 Compilation séparée

Supposons qu'un programme écrit en C soit composé de 3 fichiers : `prog.c, chaines.c` et `util.c` (ces deux derniers incluent leurs fichiers d'en-tête respectifs `chaines.h` et `util.h`). Il est possible de compiler avec :
`gcc -Wall prog.c chaines.c util.c`.
Cette façon de faire entraîne souvent des compilations inutiles dans la mesure où on ne modifie souvent qu'un seul fichier. Il est préférable de taper sucessivemment :

```
gcc -c -Wall prog.c
gcc -c -Wall chaines.c
gcc -c -Wall utill.c
gcc prog.o chaines.o util.o
```

Lorsqu'un seul fichier est modifié (`prog.c` par exemple), il suffit de ne refaire que ce qui est nécessaire (ici les étapes 1 et 4).
Inconvénient : c'est plus fastidieux, et on court le risque d'oublier de refaire une des compilations.

2.3.2 Make

L'utilitaire `make` permet de pallier ces inconvénients. Il suffit de lui préciser de quoi dépend chaque fichier et ce qu'il faut faire pour le mettre à jour. En se basant sur la date de modification des fichiers, `make` n'exécute que les actions nécessaires.
Pour cela, il faut écrire ces informations dans un fichier (qui s'appellera de préférence `makefile` ou `Makefile`).

Le format du fichier est le suivant :

```
cible1: dependances1
     action1

cible2: dependances2
     action2

etc ...
```

où `cible` est le fichier à produire, `dependance` la liste des fichiers dont il dépend et `action` la commande à exécuter pour le générer.
IMPORTANT: la ligne indiquant l'action doit obligatoirement débuter par une tabulation.
Dans l'exemple, le fichier `makefile` peut s'écrire ainsi:

```
a.out : prog.o util.o chaines.o
     gcc prog.o util.o chaines.o

prog.o : prog.c util.h chaines.h
     gcc -c -Wall -std=c99 prog.c

util.o : util.c util.h
     gcc -c -Wall -std=c99 util.c

chaines.o : chaines.c chaines.h
     gcc -c -Wall -std=c99 chaines.c
```

Une fois que ce fichier est créé, il suffit de taper `make` pour mettre à jour la compilation.

2.4 Utilisation du debogueur gdb

gdb permet de contrôler de déroulement d'un programme et de visuliser l'état des variables (On supposera ici qu'il s'agit d'un programme écrit en langage C). Pour pouvoir utiliser gdb, il faut d'abord compiler avec l'option -ggdb.

```
~ > gcc -Wall -ggdb monprog.c
~ > gdb a.out
```

Si les exécutions précedentes se sont terminés sur le message "core dumped", il faut à nouveau l'exécuter pour provoquer l'erreur et ajouter l'argument `core` à la commande

```
~ > gcc -Wall -ggdb monprog.c
~ > a.out
Segmentation Fault (core dumped)
~ > gdb a.out core
```

Il est également possible d'utiliser gdb depuis Emacs avec la commande *Esc x gdb*

2.4.1 Déroulement

La première chose à faire pour examiner le déroulement du programme par gdb est de le démarrer avec la commande `r (run)`. il est possible de préciser les arguments de la ligne de commande ex: `r donnees.txt 5`
Dans ce cas, le programme s'exécute d'une traite. Pour l'exécuter pas à pas, il faut poser des points d'areêt (breakpoints) avec la commande `b (break)` en précisant le nom de la fonction ou le numéro de la ligne. Dans le cas d'un programme multifichiers, il faut eventuellement préciser en plus le nom du fichier avec la sytaxe `b <fichier>:<fonction>` ou `b <fichier>:<ligne>`

Lorsque le programme démarre, il s'exécute jusqu'au premier point d'arrêt. Il est alors possible d'exécuter les instructions une à une en tapant `n (next)`. Dans ce cas, un appel de fonction est compté comme une instruction. Pour entrer dans la fonction, il faut taper `s (step)`.

Un numéro est attribué à chaque point d'arrêt. Le point d'arrêt n peut être désactivé avec `dis n (disable)` et réactivé avec `ena n (enable)`. Pour voir la liste des points d'arrêt, taper `info breakpoints` enfin, pour supprimer un point d'arrêt, utiliser `cl (clear)` en précisant le nom de la fonction ou le numéro de la ligne et pour tous les supprimer, taper `del (delete)`.

Pour reprendre l'exécution jusqu'au prochain point d'arrêt, taper `c (continue)`.

Il est également possible de sauter à la fin d'une boucle avec `u (until)` ou la fin d'une fonction avec `f (finish)`

La commande `bt (backtrace)` permet d'afficher la pile d'appel.

On peut définir une suite de commandes <C1>, <C2>, .. <Ck> associée au point d'arret <n> de la façon suivante :

```
commands <n>
<C1>
<C2>
..
<Ck>
end
```

Pour quitter gdb, taper `q (quit)`

2.4.2 affichage des variables

Pour vérifier que les variables prennent des valeurs conformes à ce qui est atendu, il est possible de les afficher avec la commande `p (print)`. Il suffit de spécifier le nom de la variable et gdb l'affichera sans qu'il soit besoin de préciser un format. Il est également possible d'imposer un format en le préfixant par /. Ce format peut être : d (décimal), x (hexadécimal), t (binaire), c (caractère), a (adresse), f (float). Par exemple :`p/t valeur` affiche la variable valeur en base 2.

Si on veut connaître à tout moment la valeur d'une variable, on peut utiliser `disp (display)` à la place de `print`. Dans ce cas, elle sera affichée à chaque arrêt de l'exécution (et donc à chaque instruction si on utilise step ou next). Comme pour les points d'arrêt, on peut desactiver, réactiver ou annuler la commande display avec les commandes `disable display, enable display` et `undisplay`.

On peut afficher les variables locales d'une fonction avec la commande `info locals` ainsi que ses paramètres avec la commande `info args`.

Pour modifier une variable, il suffit de taper set variable <var>=<valeur>

Appendice A

Appendice

A.1 Considérations pratiques

A.1.1 Changer le prompt

Il est conseillé d'avoir en permanence l'affichage du répertoire courant pour éviter d'utiliser sans cesse `pwd`. Pour cela, il faut changer la variable `PS1` en lui assignant la valeur `\w`.
Commencez par supprimer les définitions de `PS1` présentes dans `.bash_profile` et `.bashrc` et ajoutez à la fin du fichier `.bash_export` la commande `export PS1='\w > '`

A.1.2 Se connecter à partir de la session d'un autre utilisateur

Si vous êtes l'utilisateur toto, il suffit de taper `su - toto` pour ouvrir une session. Vous ne disposez que d'un terminal en mode texte seulement. Si vous avez besoin de ressources graphiques, vous devrez préalablement taper `xhost +`
Pour terminer cette session, tapez simplement `exit`.

A.1.3 Se connecter à une autre machine

Pour vous connecter sur la machine *hal9000*, tapez simplement `ssh hal9000`

A.1.4 Utilisation à distance

Vous pouvez utiliser les machines de l'Enseirb depuis une machine située à l'extérieur (même s'il s'agit d'un PC/Windows.)

- Si votre machine est munie d'Unix (Linux, MacOSX), il suffit d'xexécute *ssh -X ssh.enseirb-matmeca.fr* pour travailler directement sur ces machines.

- Pour utiliser le shell à distance avec windows, vous devez disposer d'un émulateur de terminal (le plus connu est *putty* (gratuit)). Lors de la connexion, vous devez préciser le nom de la machine (ssh.enseirb-matmeca.fr) et le port (22).
  Vous disposez alors d'une fenêtre reproduisant le comportement d'un terminal Unix. En revanche, vous ne disposez d'aucune ressource graphique; vous ne pourrez donc pas exécuter de programme nécessitant ces ressources.

  Cas particulier d'emacs : Il est possible d'utiliser emacs en mode "terminal". Pour cela, utiliser la commande `emacs -nw <fichier>` ou `command emacs -nw <fichier>` dans le cas où emacs serait une fonction ou un alias. Vous serez obligés d'utiliser les commandes au clavier, car ni la souris, ni les menus ne sont disponibles. (Pour accéder au shell sans quitter emacs, taper `Ctrl z`; pour revenir à emacs, tapez `fg`. (en fait, il est plus simple de lancer un deuxième terminal pour utiliser le shell)

- Pour transférer des fichiers d'un site à l'autre, utilisez *filezilla* (gratuit). Les réglages sont les mêmes que pour putty.
  S'il y a peu de données à transférer, vous pouvez aussi vous adresser un courriel auquel vous avez joint vos fichiers.

A.1.5 Imprimer un fichier

Si le fichier à imprimer est un fichier texte, il faut utiliser la commande `a2ps` pour le convertir en Postscript et l'envoyer à l'imprimante. La syntaxe est : `a2ps -P<imprimante> <fichier> ...`

Lorsqu'il s'agit déjà de Postscript (ce qui est le cas lorsque vous imprimez à partir d'un logiciel), il faut utiliser `lp` en précisant le nom de l'imprimante avec l'option `-d<imprimante>`

A.1.6 Quelques conseils pour utiliser Thunderbird

Thunderbird est suffisamment simple à utiliser et ne nécessite pas de manuel d'utilisation. Vous devez simplement vous organiser pour gérer les dizaines de courriels que vous recevez quotidiennement : Une bonne partie d'entre-eux sont écrits à tous les élèves et ne sont pas d'un intérêt primordial.
Une bonne méthode est de créer des dossiers pour aiguiller le courrier entrant. Pour cela créez de nouveaux dossiers (par un clic droit sur le nom de votre boîte) : Indesirables, Profs , MailAll
Lorsque c'est fait, faites un click gauche sur le nom de votre compte et, parmi les options proposées, choisissez `Voir les paramètres pour ce compte`. Dans la colonne de gauche, selectionnez `Paramètres pour les indésirables`, puis cochez la case `Déplacer les indésirables vers` et choisissez votre dossier `Indésirables` comme destination. A chaque fois que vous identifiez un spam, marquez-le comme indésirable avant de l'effacer. Thunderbird apprendra ainsi à reconnaître les spams. Pensez à supprimer vos courriers quand vous les avez lus, ou à les placer dena le dossier Archive si vous voulez les conserver. Pour les courriers provenant des enseignants ou de l'administration, créez un filtre (menu Messages/créer un filtre à partir du message) qui envoie ces courriers dans le répertoire

Profs.
Pensez de temps en temps à effacer le contenu du dossier Indésirables (après l'avoir parcouru rapidement pour vérifier qu'un courrier important n'a pas été écarté) et à vider la corbeille.
Si vous utilisez chez vous un ordinateur, faites les mêmes réglages.

A.2 Historique

- En 1969, Ken Thompson (laboratoires de Bell) écrit la première version de ce qui devait par la suite s'appeler Unix. Le nom "Unix" était à l'origine un calembour sur le système d'exploitation Multics (Multiplexed Information and Computing Service) et a été écrit "Unics" en premier lieu (UNiplexed Information and Computing System).
  Cette première mouture a fonctionné sur un DEC PDP-7. En 1970, Ken Thompson et Dennis Ritchie l'adaptent au DEC PDP-11/20. Il résulta de cette expérience le premier compilateur C. Le langage C a été conçu spécifiquement pour un OS portable.
  Dès 1974, le code source d'Unix est distribué librement aux universités. En conséquence, UNIX a gagné la faveur de la communauté scientifique et universitaire. Il a été ainsi à la base des systèmes d'exploitation des principales universités.

- 1978

  UNIX, 7ème édition. Cette version a été développée expressément pour être portée sur diverses architectures matérielles. En outre, avec la version 7, AT&T annonce qu'ils font payer des licences pour accéder aux sources du système. En conséquence, la version 7 forme la base de toutes les versions d'Unix actuellement disponibles. Les écrits de Brian Kernighan et Rob Pike (deux chercheurs des laboratoires Bell fortement impliqués dans le développement d'Unix) présentent la philosophie de conception d'Unix.

- 1979

  L'annonce d'AT&T de son intention de commercialiser UNIX a incité l'université de Californie à Berkeley pour créer sa propre variante : BSD UNIX.

- 1983

  AT&T met en vente la version commerciale du système V.

- 1987

  Diffusion de X Window, une interface client/serveur graphique développée au MIT (Massachussetts Institute of Technology).
  C'est cette année là qu'AT&T et Sun ont choisi conjointement d'unifier le Système V et BSD.

- A partir de 1990

  La version 4 du système V d'AT&T comporte de nouveaux standard d'unification d'UNIX. C'est le résultat de la coopération entre Sun et AT&T. Sun développe son OS, Solaris, un dérivé de la version 4 du système V.

  Les clones d'Unix comme Linux et FreeBSD commencent à émerger.

  Sun développe son OS, Solaris, un dérivé de la version 4 du système V.

  Linux a été écrit par Linus Torvalds et a été amélioré par de nombreux developpeurs bénévoles du monde entier. C'est un sosie du système d'exploitation Unix réécrit de A à Z. Linux a été protégé par le droit d'auteur conformément à GNU General Public License (GPL). C'est une licence créée par la Free Software Foundation (FSF) qui est conçue pour empêcher les limitations de distribution du logiciel.

Table des matières