TP n° 1 : Codage des types primitifs

Connexion sous Linux

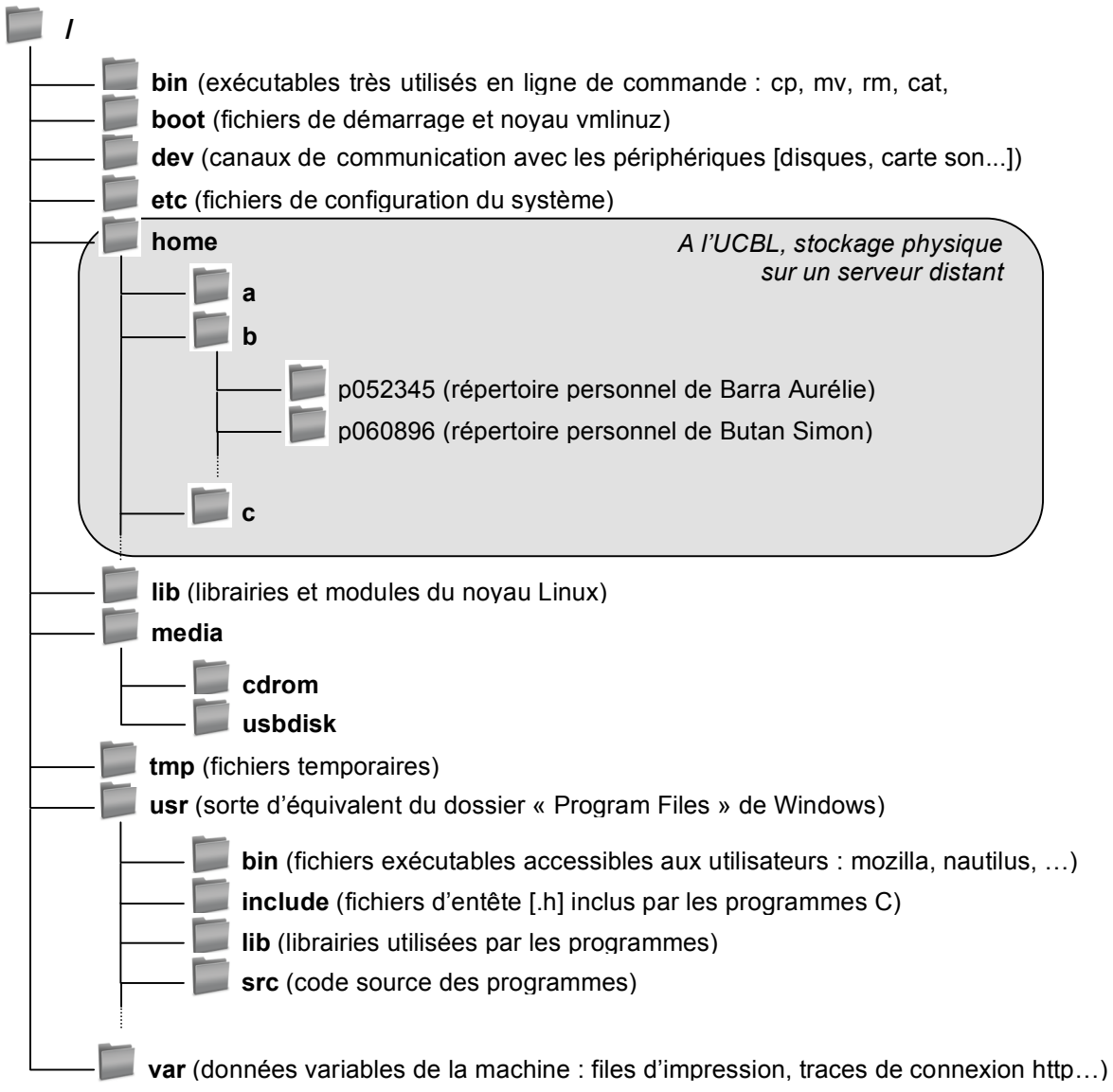
Démarrez l'ordinateur sous Linux (Ubuntu). Connectez-vous avec les mêmes login / mot de passe que sous Windows.

Configuration de Firefox

Lancez le navigateur Firefox (menu Applications / Internet). Configurez la taille de son cache à 2 Mo pour qu'il ne sature pas votre espace disque personnel (menu Outils / Options, icône Avancé, onglet Réseau).

Familiarisation avec le système de fichiers Linux

Pour utiliser au mieux son compte Linux, il est nécessaire de connaître quelques notions basiques sur le système de fichiers Linux. La racine du système de fichier (l'équivalent du « C :\ » d'un Windows non partitionné) est le répertoire « / ». Voici un schéma des principaux répertoires que contient ce dossier racine.



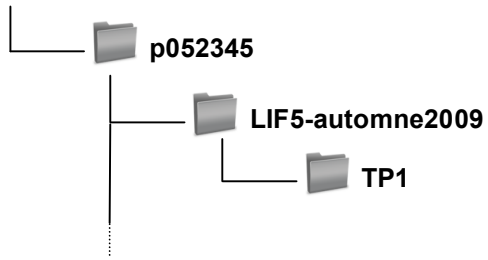
A l'université, les répertoires des utilisateurs ne sont pas stockés physiquement sur les machines des salles de TP, mais sur un serveur auquel les machines de TP accèdent par le réseau (même principe que pour votre lecteur W: sous Windows). Mais cela est transparent pour l'utilisateur, qui accède toujours à son répertoire personnel en allant dans :
`/home/[`*1e lettre du nom de famille*`]/[`*n°d'étudiant*`]`
Cela permet d'avoir accès à ses données même si l'on change de poste de travail.

Regardez le contenu (peut-être vide) de votre compte utilisateur de deux façons :
- avec le gestionnaire de fichiers Nautilus (menu Applications / Dossier personnel)
- en ligne de commande :
  - ouvrez un terminal (menu Applications)
  - vérifiez que vous êtes dans votre répertoire personnel en tapant `pwd` (cela signifie « print working directory »)
  - demandez le listing du contenu du répertoire en tapant `ls -a`

Dans cette UE, nous allons privilégier l'utilisation de la ligne de commande. Voir l'annexe en fin de ce sujet pour les commandes Linux de base et des précisions sur la notion de chemin sous Linux.

Création d'une arborescence LIF5 dans votre répertoire personnel

Vous allez créer l'arborescence suivante dans votre répertoire personnel, en n'utilisant que la ligne de commande.



Pour cela, allez dans le terminal, puis :
- vérifiez que vous êtes dans votre répertoire personnel en tapant `pwd`. Si vous n'y êtes pas, retournez-y en tapant `cd` (cela signifie « change directory », et si l'on ne précise pas de répertoire de destination, on va par défaut dans le répertoire personnel).
- créez le répertoire LIF5-automne2009 en tapant la commande suivante (respectez bien l'espace après mkdir, mais n'en mettez pas dans le nom du répertoire) :
  `mkdir LIF5-automne2009`
- vérifiez que ce nouveau répertoire apparaît dans le répertoire courant, en tapant `ls`
- allez dans le répertoire créé en tapant `cd LIF5-automne2009`
- créez le répertoire TP1 en tapant `mkdir TP1`

Vérifiez que vous retrouvez bien les dossiers créés en explorant votre dossier personnel en mode graphique.

Exercice 1 : Programmer sans environnement de développement

En LIF1, vous avez programmé en C/C++ à l'aide de Dev-C++, qui est un « environnement de développement intégré » regroupant un éditeur de texte, un compilateur, des outils automatiques de fabrication, et un débogueur. Il en existe d'autres, par exemple CodeBlocks. Cependant, il n'est pas indispensable d'utiliser ce genre d'environnement intégré pour programmer : on peut aussi utiliser un éditeur de texte et lancer la compilation et l'exécution en ligne de commande, depuis le terminal. C'est ce que nous allons faire dans cette UE. Cela vous permettra, par la suite, de mieux comprendre ce que fait un environnement de développement lorsque vous cliquez sur « Compiler », par exemple. Cela vous permettra aussi de mieux comprendre les mystérieux fichiers « Makefile » utilisés par ces environnements, car vous en aurez fait vous-même (mais nous verrons cela plus tard…).

Nous allons utiliser l'éditeur de texte « gedit ». Pour cela, allez dans le terminal, puis :

- Vérifiez que vous êtes dans le répertoire TP1 en tapant `pwd`. Si vous n'y êtes pas, retournez-y en tapant `cd ~/LIF5-automne2009/TP1` (~ désigne votre répertoire personnel).
- Lancez gedit en tapant `gedit hello.c &` . Gedit se lance et, comme le fichier hello.c n'existe pas encore dans le répertoire courant, il crée un fichier vide que vous allez pouvoir remplir.
- Tapez le code suivant dans gedit :

```
#include <stdio.h>

int main()
{
  printf("Hello world !\n");
  return 0;
}
```

- Sauvegardez vos modifications.

Nous allons maintenant compiler ce code à l'aide de gcc. Il s'agit du principal compilateur C libre. Pour compiler, retournez dans le terminal (sans fermer gedit) et tapez :
`gcc -Wall -ansi -pedantic -o hello.out hello.c`

Tapez `man gcc` pour comprendre ce que signifient les différents éléments de cette commande et compléter le tableau suivant (aide : une fois la documentation affichée, tapez `/mot` pour rechercher un mot, `n` pour passer à l'occurrence suivante, `q` pour sortir).

gcc	nom du programme que nous voulons exécuter (ici, c'est un compilateur qui va traduire notre code C en langage machine pour en faire un binaire exécutable)
-Wall	option qui demande à gcc d'afficher tous les « warnings », c'est-à-dire les avertissements indiquant des erreurs possibles dans le code, en plus des erreurs avérées
-ansi et –pedantic	ansi et pedantic demandent à gcc de rejeter les codes sources qui ne respectent pas le standard ANSI (ISO C90)
-o hello.out	option qui demande à gcc de nommer l'exécutable « hello.out » (plutôt que « a.out » comme par défaut)
hello.c	c'est l'argument, il s'agit du fichier source que gcc doit traduire en langage machine

Toujours dans le terminal, appuyez plusieurs fois sur la flèche en haut du clavier. Que se passe-t-il ? A quoi cela peut-il servir ?

Rappel des dernières commandes tapées. Très utile lorsqu'on corrige des erreurs de compilation et que l'on doit donc retaper souvent la commande de compilation.

Corrigez les erreurs de compilation éventuelles en modifiant votre code source dans gedit, en sauvegardant et en recompilant. Recommencez jusqu'à ne plus avoir aucune erreur. Vous pouvez ensuite exécuter votre programme en tapant dans le terminal : `./hello.out`

Exercice 2 : Découverte de « printf »

En LIF1, vous avez utilisé la librairie C++ « iostream » pour gérer les saisies au clavier et les affichages à l'écran. Dans cette UE, vous allez découvrir et utiliser une autre librairie, la librairie « stdio », qui est une librairie C. La première fonction à connaître dans cette librairie est la fonction « printf », qui est la fonction d'affichage sur la sortie standard du processus (en général l'écran, pour nous). Exemple :

```
#include <stdio.h>

int main()
{
  double a = 18.159;
  int i = 3;
  printf("Le réel vaut %f et l'entier vaut %d \n", a, i);
  return 0;
}
```

Quelques explications :

- le premier paramètre est une chaîne de caractères appelée « chaîne de format », contenant à la fois des libellés ("Le réel vaut ", " et l'entier vaut ", " \n") et des codes de format (%f et %d ici). Il y a ensuite autant de paramètres supplémentaires que de codes de format.
- printf convertit ses paramètres en caractères affichables selon les codes de format demandés, construit la chaîne à afficher en remplaçant les codes de format par les valeurs converties, et affiche le résultat sur la sortie standard.

Code de format	Conversion en…	Ecriture
%d	int	base 10 signée
%ld	long int	base 10 signée
%u	unsigned int	base 10 non signée
%lu	unsigned long	base 10 non signée
%o	unsigned int	base 8 (octale) non signée
%lo	unsigned long	base 8 (octale) non signée
%x	unsigned int	base 16 (hexadécimale) non signée
%lx	unsigned long	base 16 (hexadécimale) non signée

%f ou %.*n*f	double	décimale virgule fixe (taille auto ou *n* chiffres après la virgule)
%lf ou %.*n*lf	long double	décimale virgule fixe (taille auto ou *n* chiffres après la virgule)
%e	double	décimale notation exponentielle
%le	long double	décimale notation exponentielle
%g	double	décimale, représentation la plus courte parmi %f et %e
%lg	long double	décimale, représentation la plus courte parmi %lf et %le
%c	unsigned char	caractère
%s	char*	chaîne de caractères

- printf retourne ensuite le nombre de caractères écrits, ou un nombre négatif en cas d'erreur (dans l'exemple ci-dessous, cette valeur de retour n'a pas été utilisée).

Dans gedit, ouvrez un nouveau fichier que vous appellerez `exo_printf.c`.Vous avez à présent deux fichiers ouverts dans gedit : hello.c et exo_printf.c. Lorsqu'on travaille sur plusieurs fichiers en même temps, il est nettement préférable d'ouvrir un seul gedit et d'utiliser les onglets, plutôt que d'ouvrir plusieurs fenêtres gedit.

Dans le fichier exo_printf.c, écrire un petit programme qui crée et initialise des variables de différents types (char, short, unsigned short, int, unsigned int, float, double, ...), puis affiche leur contenu en utilisant printf et les codes de format appropriés. Compilez et exécutez votre programme (nommez l'exécutable exo_printf.out), et vérifiez ses résultats. Testez au passage les effets des caractères spéciaux `\n`, `\t`, `\b`, `\a`, `\"` et `\\`.

Modifiez à présent votre programme pour pouvoir répondre aux questions suivantes : que se passe-t-il si…

… on affiche un « unsigned char » initialisé à 'A' avec le code %u au lieu de %c ?	On obtient le code ASCII du caractère 'A'.
… on affiche un signed int avec le code %u : obtient-on sa valeur absolue ? Pourquoi ?	Non, on obtient le décodage en binaire pur d'un nombre prévu pour être décodé en complément à 2 (soit $2^{32} - x$)
… on affiche un « unsigned int » proche de 2^{32} avec le code %d : quelle valeur obtient-on ?	On obtient un nombre négatif : le décodage en complément à 2 d'un nombre prévu pour être codé en binaire pur (soit $x - 2^{32}$)
A-t-on le problème précédent avec un « unsigned char » proche de 2^8 ou un « unsigned short » proche de 2^{16} ? Pourquoi ?	Ce problème ne se pose pas car le code %d implique un cast en signed int, donc vers un type plus large
… on affiche un float avec le code %d : obtient-on un arrondi du réel ?	Non ! Pour afficher un arrondi du réel à l'entier le plus proche, il faut utiliser le code de format `%.0f`
… on affiche un float avec le code %x : obtient-on le codage IEEE du réel ?	Non. En fait, le compilateur effectue d'abord une conversion du float en double avant d'appeler réellement printf, car printf n'accepte pas directement les float. Donc on a déjà un problème de taille : les 8 octets du double ne peuvent pas être ramenés sans perte sur 4 octets (%x = unsigned int).

Exercice 3 : Entiers et caractères

Considérons le programme suivant.

```
#include <stdio.h>

int main ()
{
  unsigned char v;
  for (v = 0; v <= 255; v++)
    printf("%d %c\n", v, v);
  return 0;
}
```

A votre avis, qu'est-il censé faire ?

Afficher la table ASCII étendue (chaque caractère associé à son code).

Tapez ce code dans un nouveau fichier que vous appelerez `exo_entiers.c`. Compilez-le et exécutez-le. Que se passe-t-il ? Comment l'expliquez-vous ?

Le programme tourne en boucle infinie, car c n'atteint jamais la valeur d'arrêt (256), puisque celle-ci n'est pas représentable avec un unsigned char. Lorsqu'on fait 255 + 1, on obtient 0 au lieu de 256, et on repart donc dans la boucle. Il s'agit donc d'un problème de dépassement de capacité (overflow).

Exercice 4 : Réels

Tapez le code suivant dans un nouveau fichier que vous appellerez `exo_reels.c`.

```
#include <stdio.h>

int main()
{
 float pi = 3.14159265;
 if (pi == pi + 5e9 – 5e9)
  {
    printf("pi = pi, tout va bien !\n");
  }
 else
  {
    printf("pi != pi, oups...\n");
  }
 return 0;
}
```

Compilez-le avec la commande suivante (attention à l'option supplémentaire –ffloatstore) :

```
gcc -Wall -ansi -pedantic -ffloat-store -o exo_reels.out exo_reels.c
```

Exécutez-le. Quel affichage obtenez-vous ? Essayez de localiser le ou les problèmes en ajoutant des printf pour voir les résultats des calculs intermédiaires avec au moins 10 chiffres après la virgule (code de format : `%.10f`).

On obtient l'affichage « pi != pi, oups… ».
En ajoutant les printf demandés, on obtient :

```
pi              =           3.1415927410125732421875000000000
```

```
pi + 5e9       =  5000000003.14159297943115234375000000000
pi + 5e9 – 5e9 =            3.14159297943115234375000000000
```

On constate donc 2 problèmes :
1) pi ne vaut pas exactement la valeur que l'on avait spécifié : c'est parce cette valeur ne peut pas être codée exactement en binaire. Ceci dit, ce n'est pas cela qui cause l'échec du ==.
2) l'addition pi+5e9 ne conserve pas toutes les décimales de pi. C'est en fait là que se trouve le problème.

Proposez à présent une explication, en utilisant l'indication suivante : 3,14159265 est codé en interne sous la forme 1,10010010000111111011011 . 2^1, c'est-à-dire avec un exposant qui vaut 1.

Signalons tout d'abord que pi va être converti en double pour pouvoir être ajouté à 5e9, donc 29 zéros vont être ajoutés à la fin de la mantisse de pi (la pseudo-mantisse est codée sur 52 bits en double précision). Ensuite, pi va être dénormalisée pour pouvoir l'additionner avec 5e9, et c'est là que la perte de précision se produit.

En effet, 5e9 est codée sous la forme 1,… . 2^{32}, c'est-à-dire avec un exposant beaucoup plus élevé que celui de pi. On doit donc dénormaliser pi en décalant la virgule de 31 crans vers la gauche, ce qui conduit à la perte des 31 derniers bits de la mantisse, qui sont pour la plupart à 0, sauf les deux derniers. Cela correspond à une perte absolue de $2^{32}.(2^{-53} + 2^{-54})$ sur la valeur de la variable pi.

$$0,00000000000000000000000000000011001001000011111101101\sout{11}.2^{32}$$

Le dernier bit de la pseudo-mantisse est arrondi à 1, car ce nombre machine est plus proche de pi que le nombre obtenu par simple troncature :

$$0,00000000000000000000000000000011001001000011111101111.2^{32}$$

La dénormalisation revient donc au final à enlever $2^{32}.(2^{-53} + 2^{-54})$ et à ajouter $2^{32}.2^{-52}$) .
On va donc ajouter 5e9 à 3,141592979… au lieu de l'ajouter à 3,14159274…

$$\begin{array}{rl} & 0,00000000000000000000000000000011001001000011111101111 \cdot 2^{32} \\ + & 1,00101010000001011111001000000000000000000000000000000 \cdot 2^{32} \\ \hline & 1,00101010000001011111001000000011001001000011111101111 \cdot 2^{32} \end{array}$$

A ce stade, la somme vaut 5000000003,141592979… au lieu de 5000000003,14159274…

Il faut ensuite retrancher 5e9 à ce nombre, mais cette fois les deux nombres sont exprimés dans le même exposant donc on n'aura pas de dénormalisation. Le problème était donc bien localisé dans l'étape pi+5e9 : les deux opérandes étant d'ordres de grandeur différents, il a fallu dénormaliser le plus petit (pi), ce qui a entraîné une perte de précision.

A retenir : les opérations arithmétiques sur les flottants (float mais aussi double) impliquent parfois des opérations de dénormalisation et de renormalisation qui affectent la précision des calculs.

Comment résoudre ce problème ici ? Quelle conclusion en tirez-vous sur l'addition de flottants sur un ordinateur ?

Dans notre cas, il suffit de mettre entre parenthèses (5e9 – 5e9).

Moralité : sur un ordinateur, quand on travaille avec des flottants, l'ordre dans lequel on effectue les opérations a une incidence sur le résultats. Autrement dit, l'addition n'est PAS associative : `(a + b) + c` n'est pas forcément égal à `a + (b + c)` .

** Fin de la partie obligatoire **

Exercice 5 (facultatif, réservé aux plus avancés) : Affichage binaire

Le but de cet exercice est d'écrire un programme qui affiche un entier naturel de type « unsigned int » (initialisé à une valeur valide de votre choix) en base 2, en utilisant les opérateurs de manipulation bit à bit. L'affichage commencera, bien sûr, par les bits de poids fort. Exemples :

```
0   → Le nombre 0 s'écrit en binaire : 00000000000000000000000000000000
9   → Le nombre 9 s'écrit en binaire : 00000000000000000000000000001001
812 → Le nombre 812 s'écrit en binaire : 00000000000000000000001100101100
```

Procédons par étapes.

a) Sur combien de bits est codé un unsigned int sur votre machine ? Pour le vérifier, ouvrez un nouveau fichier nommé affichage_binaire.c dans gedit et tapez-y le code suivant :

```
#include <stdio.h>
#include <limits.h> /* pour CHAR_BIT */

int main()
{
  unsigned int nbbits = sizeof(unsigned int)*CHAR_BIT;
  /* sizeof renvoie la taille en bytes au sens C/C++ (1 byte =
   quantité de mémoire nécessaire pour stocker un caractère). CHAR_BIT
   est une constante definie dans le fichier limits.h, elle donne le
   nombre de bits pour stocker un char. C'est très souvent 8 bits,
   mais cela peut être plus sur certaines architectures. */

  printf("Un unsigned int est code sur %u bits.\n", nbbits);

  return 0;
}
```

Réponse : en principe, 32.

b) Soit a un « unsigned int » intialisé à une valeur valide. Soit valbit un « unsigned int » également. On veut que valbit vaille 0 si le bit de poids faible de a vaut 0, ou 1 si le bit de poids faible de a vaut 1. Quelle instruction faut-il utiliser ? Indication : il faut utiliser l'opérateur & (« et » bit à bit) et une constante.

```
valbit = a & 1 ;

Exemple :    a = 00000000000011110101011101101011
             1 = 00000000000000000000000000000001
         ----------------------------------------
         a & 1 = 00000000000000000000000000000001
```

c) On veut maintenant que valbit prenne la valeur du bit i de a (le bit de poids faible étant le bit 0, et le bit de poids fort étant le bit nbbits-1). Quelle instruction faut-il utiliser ? Indication : il faut utiliser les opérateurs de décalage (<< et >>) en plus de l'opérateur &.

```
valbit = (a & (1 << i)) >> i;

Exemple pour i = 5 :    a = 00000000000011110101011101101011
                      1<<5 = 00000000000000000000000000100000
                      -------------------------------------
                a & (1<<5) = 00000000000000000000000000100000
           (a & (1<<5))>>5 = 00000000000000000000000000000001
```

d) Vous pouvez à présent compléter le « main » de votre fichier pour qu'il affiche un « unsigned int » a en base 2 : il faut effectuer l'instruction précédente pour i allant de nbbits-1 à 0, en affichant la valeur de valbit à chaque fois. Attention de ne pas tomber dans une boucle infinie. Testez votre programme pour différentes valeurs, dont zéro.

```
#include <stdio.h>
#include <limits.h> /* pour CHAR_BIT */

int main()
{
  unsigned int nbbits = sizeof(unsigned int)*CHAR_BIT;
  unsigned int a = 9;
  unsigned int valbit;
  unsigned int i; /* pour ne pas melanger des signed et des unsigned */

  printf("Le nombre %u s'ecrit en binaire : ", a);

  for(i = nbbits-1; i > 0; i--)
    {
      valbit = (a & (1 << i)) >> i;
      printf("%u", valbit);
    }


  /* On traite le bit 0 a part car on ne peut pas utiliser
     i >= 0 comme condition d'arret dans la boucle :
     puisque i est unsigned, il ne peut pas etre negatif */

  valbit = a & 1;
  printf("%u", valbit);

  printf("\n");

  /* printf("en hexa : %x\n", a); */

  return 0;
}
```

Annexe : Commandes Linux usuelles

Action	Commande
Obtenir de l'aide sur une commande	man *commande*
Chercher les commandes relatives à un mot-clé	man –k *motcle*
Obtenir des informations sur l'utilisateur courant	who am i
Lister le contenu du répertoire courant	ls ls –a (affiche aussi les fichiers cachés) ls –al (affichage d'étaillé)
Lister le contenu d'un répertoire autre que le répertoire courant	ls *cheminrepertoire* ls –a *cheminrepertoire* ls –al *cheminrepertoire*
Changer de répertoire	cd *cheminrepertoire*
Aller au répertoire père	cd ..
Aller à la racine de son répertoire personnel	cd cd ~
Afficher le chemin du répertoire courant	pwd (print working directory)
Créer un répertoire	mkdir *cheminrepertoire*
Visualiser le contenu d'un fichier texte	more *cheminfichier* (entree = ligne suivante ; espace = page suivante ; q = quitter)
Editer un fichier texte	gedit *cheminfichier* &
Faire une copie d'un fichier	cp *cheminsource chemincible*
Déplacer ou renommer un fichier	mv *cheminsource chemincible*
Supprimer un fichier	rm *cheminfichier*
Supprimer un répertoire et tout son contenu	rm –r *cheminrepertoire*

Notion de chemin

Le chemin permet de savoir où se trouve un fichier ou un répertoire dans l'arborescence. Deux types de chemins sont utilisés sous Linux :

- les chemins absolus : ils indiquent tout le chemin d'accès à partir de la racine du système (/)
- les chemins relatifs : ils indiquent le chemin à partir du point où l'on est dans l'arborescence (répertoire courant). Le chemin relatif permettant d'accéder au répertoire père du noeud courant est ..

Exemples :

- `/home/b/p012456/LIF5-automne2008/TP1` est le chemin absolu du répertoire TP1.
- `/home/b/p012456/LIF5-automne2008/TP2` est le chemin absolu du répertoire TP2.
- `/home/b/p012456/LIF5-automne2008/TP1/hello.c` est le chemin absolu du fichier hello.c.
- Si on est dans le répertoire LIF5-automne2008, alors le chemin relatif d'accès au répertoire TP1 est simplement `TP1` (ou `./TP1`).
- Si on est dans le répertoire TP1, alors :
  - le chemin relatif d'accès au répertoire LIF5-automne2008 est `..`
  - le chemin relatif d'accès au répertoire TP2 est `../TP2`
  - le chemin relatif d'accès au fichier hello.C est `hello.c` (ou `./hello.c`)

Lorsque vous vous connectez, vous êtes placé dans votre répertoire personnel (par exemple `/home/b/p012456`).