2. Bases Linux, premier programme C, compilation, exécution

1 Bases de *Linux* pour l'utilisateur

1.1 Commandes, ligne de commande

Pour utiliser un ordinateur fonctionnant avec *Linux* il faut d'abord fournir un « identifiant » (c'est à dire un nom d'utilisateur) et un mot de passe[1]. Dans la suite on prend l'exemple d'un utilisateur nommé « averell » et d'un ordinateur nommé « vega » .
Ensuite des commandes peuvent être transmises de deux façons :

1. par des icônes, des menus et des raccourcis clavier (comme dans *Microsoft Windows*)
2. par l'écriture de ces commandes dans une fenêtre particulière nommée « Terminal » . Cette écriture se fait à l'aide du clavier mais aussi de la souris et de quelques procédés qui permettent de réduire au minimum les caractères à taper.

C'est la seconde méthode qui est décrite ici.
Le système indique qu'il est prêt à recevoir des commandes dans la fenêtre Terminal en inscrivant à l'écran, dans cette fenêtre, pour l'exemple envisagé ici, la suite de caractères :

averell@vega $\sim$ *$*

qui s'appelle l'« invite » (ou « prompt »). Pour abréger on l'écrira parfois simplement $ dans la suite. L'espace vierge situé à la suite de cette invite, sur la même ligne, s'appelle la « ligne de commande » , c'est là que l'utilisateur écrit ses commandes.

Remarque :

Dans l'écriture d'une commande *Linux*, minuscules et majuscules n'ont pas la même signification.

1.2 Fichiers, répertoires

Les fichiers sont des ensembles d'informations écrites sous forme de 0 et de 1 sur des supports tels que mémoire vive, disques durs, clés USB, bandes magnétiques, cartes mémoire, CD, DVD, etc. A chaque utilisateur est attribuée une zone personnelle sur le disque dur de l'ordinateur, dans laquelle il crée, modifie et conserve ses fichiers.
Ces fichiers peuvent avoir des contenus très divers :

- texte brut directement lisible (fichier dit « texte ») qui peut représenter par exemple :
  — un programme écrit en C ou en tout autre langage
  — des données sous forme de chiffres ou de caractères alphanumériques
  — un texte ordinaire non mis en forme
- informations écrites dans un codage particulier, lisibles seulement par l'intermédiaire d'un logiciel et représentant par exemple :
  — un programme C (ou autre langage) sous forme compilée
  — un texte mis en forme par un traitement de texte
  — des figures, images, sons etc.

Les fichiers textes peuvent être créés et modifiés par l'utilisateur, soit directement à l'aide d'un « éditeur » (ensemble de commandes permettant d'écrire, depuis le clavier, des caractères dans une fenêtre de l'écran[2] et de sauvegarder ce qui est écrit dans cette fenêtre dans un fichier), soit indirectement par l'intermédiaire d'un processus quelconque effectué par la machine (par exemple fichier contenant les résultats d'un calcul fait par un programme C).

Conseil pratique :

Avoir un fichier (unique) dans lequel on note toutes les informations utiles que l'on écrit habituellement sur de petits bouts de papier.

1.2.1 Répertoires, arborescence

De même que, dans la vie courante, il est plus pratique de ranger ses papiers dans plusieurs dossiers, plutôt que dans un seul où tout se mélange, les fichiers sont classés dans des ensembles nommés *répertoires*[3].

1. En général plusieurs utilisateurs sont enregistrés sur un ordinateur donné et sont donc susceptibles de l'utiliser.
2. Différente de la fenêtre Terminal.
3. Éventuellement imbriqués les uns dans les autres.

L'utilisateur peut créer à sa guise, une structure en arbre constituée de répertoires et de fichiers auxquels il attribue des noms de son choix. La figure suivante donne un exemple d'une telle structure :

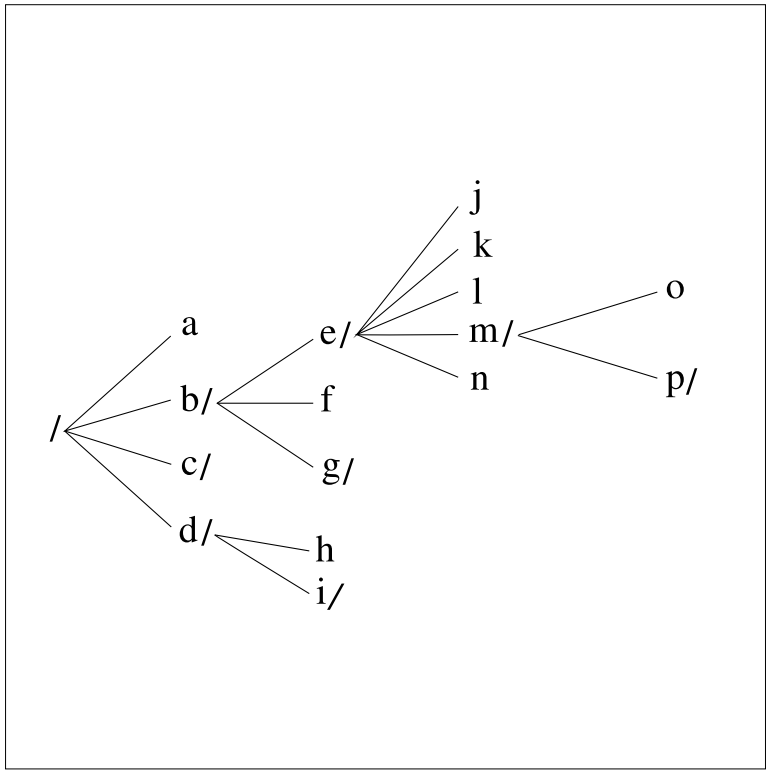

FIGURE 1 – Exemple de structure arborescente de répertoires et de fichiers

Dans cet exemple on a nommé les répertoires et les fichiers à l'aide de simples lettres, pour simplifier la description. En général on leur donne des noms plus explicites pour mieux s'y retrouver (voir exemple ci-dessous). Sur la figure les répertoires se distinguent des fichiers par le fait que leur nom est suivi d'un « slash » : « / » . Par exemple *b* est un répertoire contenant deux répertoires (*e* et *g*) et un fichier (*f*). *e*, à son tour, contient un répertoire et quatre fichiers.

1.2.2 Répertoires particuliers

« répertoire initial »
: répertoire personnel de l'utilisateur dans lequel il se trouve initialement placé après s'être identifié sur l'ordinateur. Il porte le nom de l'utilisateur (par exemple *averell*) et il peut être désigné de façon générique par un « tilde » : « ∼ » . C'est dans son répertoire initial qu'un utilisateur crée tous ses répertoires et ses fichiers.

« répertoire courant »
: répertoire dans lequel se trouve l'utilisateur au moment où il écrit une commande. Son nom est inscrit à la fin de l'invite, il peut être désigné de façon générique par un point : « . » .

« répertoire père »
: répertoire immédiatement ascendant du répertoire courant, il peut être désigné de façon générique par deux points : « .. » .

« répertoire racine »
: répertoire le plus en amont, le père de tous, il est désigné par un slash : « / »

1.2.3 Chemin d'accès

Le « chemin d'accès » d'un répertoire ou d'un fichier est la suite des répertoires qui permettent d'arriver jusqu'à lui depuis le répertoire racine inclus. Dans l'exemple de la figure 1, le chemin d'accès du répertoire *p* est */b/e/m*.

Le nom complet d'un répertoire ou d'un fichier est son chemin d'accès suivi de son nom local. Le nom complet du fichier o est $/b/e/m/o$.
Pour désigner un fichier à partir d'un répertoire quelconque de l'arbre il faut, en principe, donner son nom complet, en particulier parce que plusieurs répertoires ou fichiers de pères différents peuvent porter le même nom (déconseillé), mais des simplifications apparaissent souvent :

— si le répertoire ou fichier que l'on veut désigner est un descendant du répertoire courant il suffit de donner le chemin à partir du fils du répertoire courant. Par exemple, pour désigner o :

quand on est dans e : m/o
quand on est dans b : $e/m/o$

— si le répertoire ou fichier que l'on veut désigner n'est pas un descendant du répertoire courant on peut quand même simplifier en donnant le nom complet sous forme relative par rapport au répertoire courant. En effet, comme « .. » désigne de façon abrégée le répertoire père du répertoire courant, pour désigner m en étant dans g on peut écrire $../g/e/m$.

La désignation des fichiers et des répertoires est également facilitée par l'utilisation des caractères génériques et le mécanisme de « complétion » des noms de fichiers (voir polycopié *Linux*), ainsi que par l'utilisation du copier-coller avec la souris (voir TD). Il n'est jamais nécessaire d'écrire complètement le nom d'un fichier pour le désigner, sauf, évidemment, au moment de sa création.

1.2.4 Quelques commandes *Linux* pour manipuler les fichiers et les répertoires

pwd : affichage du nom de répertoire courant

Ainsi, si le répertoire courant est e, et en notant l'invite par $\$$:

$\$\ pwd$

donne le résultat :

$/b/e$

cd : déplacement dans les répertoires

Si l'utilisateur se trouve dans le répertoire b et écrit la commande :

$\$\ cd\ e$

il passe de b à e. Le répertoire courant qui était b devient e. Si l'utilisateur se trouve dans le répertoire g et écrit la commande :

$\$\ cd\ ../e/m$

le répertoire courant qui était g devient m.
La figure suivante illustre quelques cas possibles.

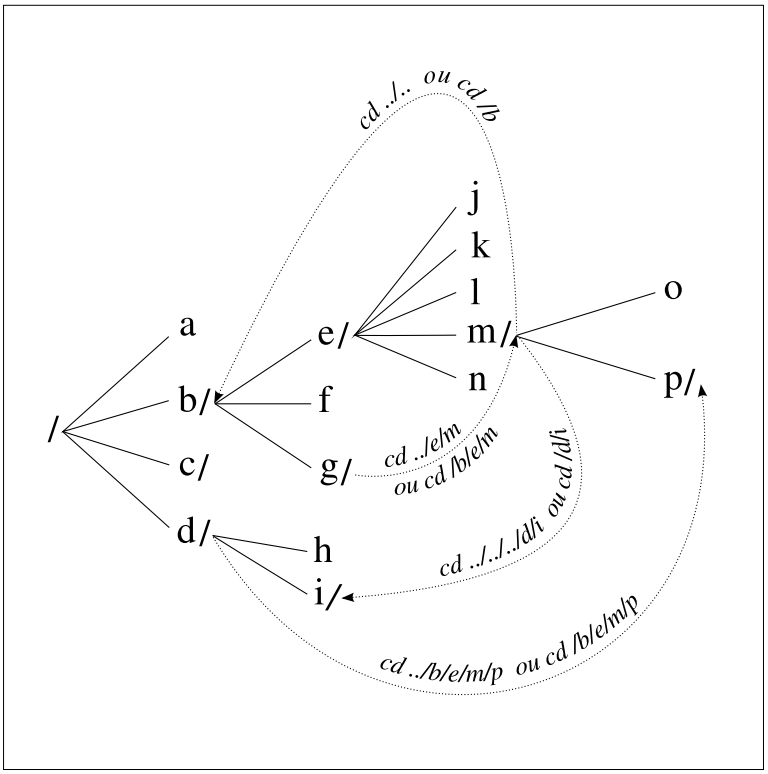

FIGURE 2 – Exemples de passage d'un répertoire à un autre, attention au sens des flèches

Exercice :

Ecrire les commandes permettant de faire les déplacements inverses de ceux indiqués par les quatre flèches de la figure 2.

La commande *cd* employée seule (sans « argument ») fait revenir au répertoire initial. Elle est donc équivalente à *cd* ∼.

ls : liste des fichiers du répertoire courant

Si on est dans le répertoire *e* :

$ ls

donne le résultat :

j k l m n

A l'écran les noms des répertoires apparaissent avec une couleur, ceux des fichiers avec une autre. Par exemple *j k l n* seront en noir, *m* en bleu. Si on veut que les noms de répertoires soient suivis d'un slash comme sur la figure 1, il faut écrire la commande *ls* avec l'option *F*[4].

$ ls -F

donne le résultat :

j k l m/ n

cp : copier un fichier

$ cp x y

copie le contenu du fichier *x* dans le fichier *y*. Si *y* n'existait pas il est créé, s'il existait son contenu antérieur est remplacé par celui de *x*. Dans ce dernier cas, si *cp* est employé avec l'option *i*, par précaution le système demande confirmation avant de remplacer le contenu de *y* par celui de *x*.

mv : changer le nom d'un fichier ou le déplacer

4. L'option d'une commande est précédée d'un tiret : « - » .

$ mv x y

Si x est un fichier et y n'existe pas, ou est aussi un fichier, le nom de x est changé en y, l'ancien x et l'ancien y s'il existait, disparaissent. Avec l'option i, une confirmation est demandée si y existait.
Si x est un fichier ou un répertoire et y un répertoire, x est déplacé du répertoire courant dans le répertoire y.
Si x est un répertoire et y n'existe pas, le nom de x est changé en y.

rm : supprimer un fichier

$ rm x

Avec l'option i, une confirmation de la suppression est demandée.

touch : créer un fichier vide

$ touch x

crée un fichier vide de nom x.

mkdir : créer un répertoire

$ mkdir x

crée un répertoire vide de nom x dans le répertoire courant

rmdir : supprimer un répertoire

$ rmdir x

Supprime le répertoire x à condition qu'il soit vide.

cp -a : copier récursivement un répertoire

$ cp -a x y

si x est un répertoire et y n'existe pas, crée une copie de x de nom y récursivement, c'est à dire contenant tous les répertoires descendant de x.

rename : permet de renommer des ensembles de fichiers

*$ rename 's/\.abc$/def/' *.abc*

change, pour tous les fichiers d'extension *abc*, cette extension en *def*. Mais si le fichier contient aussi la chaîne *abc* dans son nom, c'est à dire avant l'extension, cette chaîne ne sera pas modifiée.

locate : permet de savoir où se trouve un fichier dans l'arborescence

$ locate xyz

affiche le nom complet d'un fichier dont le nom contient la chaîne *xyz*. *locate* ne recherche pas directement dans l'arborescence actuelle mais dans une base de données qui n'est pas forcément à jour. Pour la mettre à jour il faut utiliser la commande *updatedb*.

find : permet de rechercher des groupes particuliers de fichiers

$ find lulu -name ''xyz'*' -print*

affiche le nom complet de tous les fichiers situés sous le répertoire *lulu* dans l'arborescence et qui contiennent la chaîne *xyz* dans leur nom. On peut imposer toutes sortes de conditions supplémentaires sur les propriétés des fichiers recherchés.

grep : permet de chercher une chaîne de caractères dans un ou un ensemble de fichiers textes dont les noms sont connus

$ grep toto fifi

affiche toutes les lignes du fichier *fifi* qui contiennent la chaîne *toto*

find et ***grep*** combinés : permet de chercher une chaîne de caractères dans un ou un ensemble de fichiers textes dont les noms sont inconnus

find lulu -print | xargs grep Jules

affiche le nom complet de tous les fichiers situés sous le répertoire *lulu* dans l'arborescence et qui contiennent la chaîne de caractères *Jules*. La ligne du fichier qui contient la chaîne de caractères *Jules* est également affichée.

man : afficher à l'écran le mode d'emploi d'une commande

$ man ls

explique le rôle et les différentes options de la commande *ls*.

which : indique où se trouve une commande dans l'arborescence

$ which ccc

donne le chemin d'accès de la commande *ccc*.

Exercice :

Créer une arborescence identique à celle de la figure 1. On peut créer plusieurs répertoires à la fois par *mkdir x y z* et plusieurs fichiers à la fois par *touch u v w*, par exemple.

Toutes ces commandes *Linux* et quelques autres sont également décrites dans le polycopié *Linux*, chapitre I, inclus dans le polycopié de TD.

1.2.5 Vue d'ensemble de l'arborescence de fichiers d'un système *Linux*

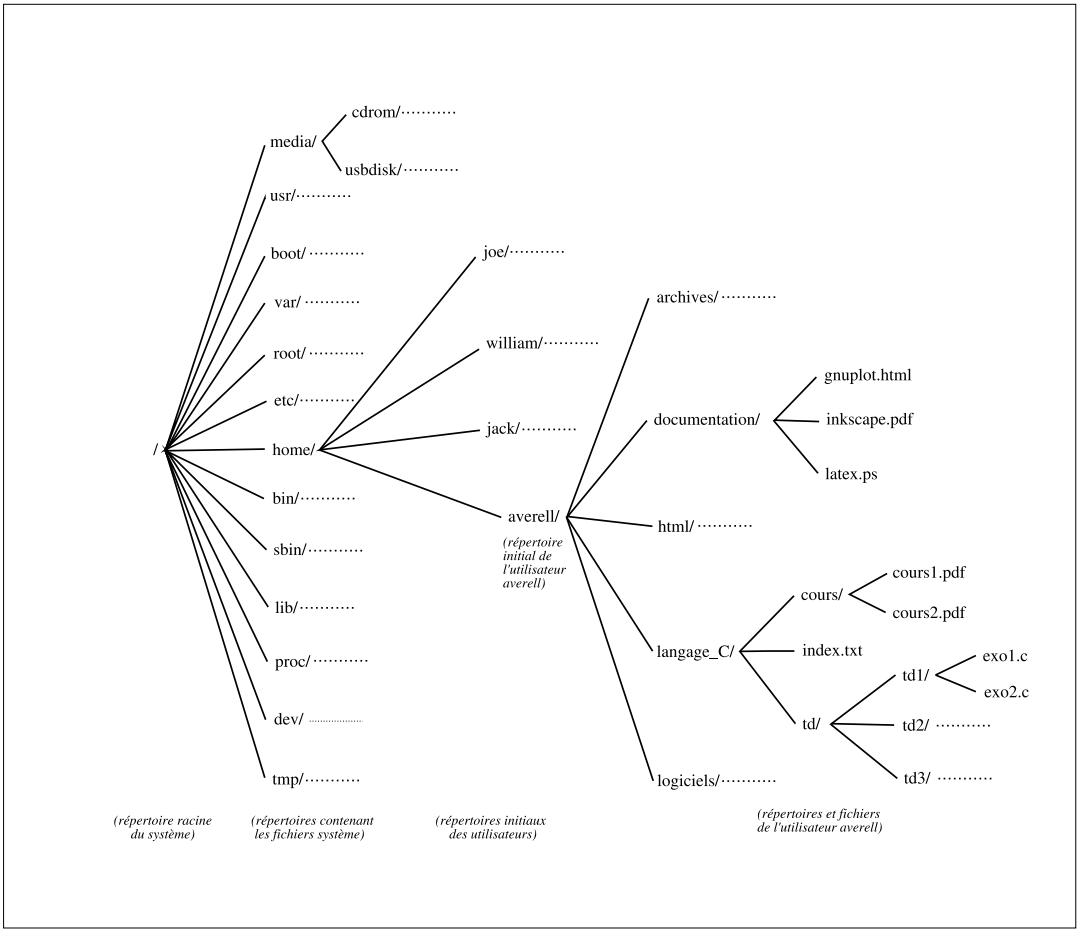

FIGURE 3 – Exemple d'arborescence de fichiers d'un système *Linux*

Les répertoires initiaux des utilisateurs[5] sont tous situés dans un répertoire nommé *home*. Ce répertoire fait partie d'une arborescence plus vaste constituée des fichiers système. Un utilisateur quelconque peut, à l'aide de la commande *cd*, se déplacer en dehors de ses répertoires personnels, à condition d'avoir les droits d'accès. Ainsi averell peut, en faisant, à partir de son répertoire initial, *cd ../joe*, aller dans les répertoires de l'utilisateur joe et y lire, copier, modifier et détruire des fichiers, à condition que joe lui en ait donné l'autorisation. De même un utilisateur quelconque peut lire, mais pas modifier, certains fichiers système. La façon de fixer les droits d'accès est expliquée dans un chapitre consacré à *Linux* .

5. Ici il y a quatre utilisateurs : joe, william, jack et averell.

1.3 L'éditeur

L'éditeur est l'ensemble des commandes qui permettent d'écrire dans un fichier à partir du clavier.
On dispose de deux éditeurs : *vi* et *emacs*, par ordre de sophistication croissante. On utilise *emacs* dont un mode d'emploi simplifié est donné dans le polycopié de TD.

2 Premier programme en C

Le programme le plus simple qu'on puisse écrire en C est :

```
int main() {
  return 0;
}
```

Il ne fait presque rien. Il mérite cependant d'être mentionné car tout programme est écrit à partir de ce point de départ.

`int main()`

indique le début du programme principal qui est celui dont le déroulement dirige tout le reste. Les accolades qui suivent contiennent l'unique instruction du programme principal :

`return 0;`

qui retourne la valeur 0 à la commande *Linux* qui a fait exécuter ce programme. Cette valeur 0 signifie que le programme s'est exécuté normalement.
Dans cet exemple il n'y a rien d'autre que le programme principal.

Considérons le programme :

```
#include<iostream>
using namespace std;
int main() {
  double x, y, z;
  x = 2.; y = 3.; z = x/y;
  cout << "Resultat : " << z << endl;
  return 0;
}
```

supposé écrit dans un fichier de nom, par exemple, *monprog.cpp* (on adopte comme convention obligatoire de suffixer les fichiers qui contiennent du C/C++ par *.cpp*).

Commentaire détaillé :

`#include<iostream>`
`using namespace std;`

indiquent qu'on utilise la bibliothèque d'entrée-sortie pour pouvoir écrire le résultat

`double x, y, z;`

indique qu'on déclare trois variables réelles nommées `x, y, z`

`x = 2.; y = 3.; z = x/y;`

attribue des valeurs à `x` et `y`, et attribue la valeur du quotient `x/y` à `z`

`cout << "Résultat : " << z << endl;`

fait imprimer à l'écran le mot *Résultat :* suivi de la valeur de `z`.
`endl` fait aller à la ligne après ce qui précède.

Une fois que ce programme est écrit il faut procéder à deux opérations pour obtenir son résultat :

1. la « compilation »
2. l' « exécution » .

2.1 Compilation

La compilation est la traduction des instructions du C en un langage plus adapté au fonctionnement de la machine que l'on utilise. Chaque instruction du programme C, assez proche de la pensée humaine, est développée en une suite d'instructions plus élémentaires dont l'écriture directe serait laborieuse. On peut comparer la compilation à ce qui se passe lorsqu'on fait une addition avec un boulier chinois. Le programme d'origine serait, par exemple, « ajouter 5 et 7 » et le programme compilé la description de la séquence d'opérations élémentaires à effectuer avec le boulier pour parvenir à ce résultat. Durant la compilation la machine analyse le programme sans exécuter les instructions. Elle évalue la taille des emplacements de mémoire qui sont nécessaires. Si elle détecte une incohérence ou un non respect des règles elle le signale : on dit qu'il y a une erreur à la compilation. Si elle ne dit rien ceci ne veut pas dire qu'il n'y a pas d'erreur : ce qu'on a écrit peut avoir un sens en C mais différent de celui auquel on pense et les ennuis viendront plus tard. Le résultat de la compilation est écrit dans un nouveau fichier dit « exécutable » qui, dans notre cas, porte le nom *a.out* quel que soit le nom du fichier dit « source » qui contient le C. En accord avec ce qui précède, le fichier exécutable est beaucoup plus volumineux que le fichier source. De plus, alors que le fichier source est indépendant de la machine utilisé, l'exécutable lui, ainsi que le compilateur qui le fabrique, est complètement lié à cette machine. On peut utiliser *monprog.cpp* tel quel sur n'importe quelle machine. Mais le *a.out* obtenu avec le compilateur d'une machine X ne peut, en général, fonctionner sur une machine Y. *monprog.cpp* doit être recompilé avec le compilateur de la machine Y. La compilation étant une traduction dans un langage plus proche de la machine, il n'est pas étonnant que le résultat dépende fortement de l'architecture de cette dernière. Pour l'utilisateur la compilation est une étape qui ne paraît pas très importante car pratiquement tout est caché mais c'est en réalité une opération extrêmement complexe.
En pratique, pour compiler le programme écrit dans le fichier *monprog.cpp*, il faut écrire la commande :

g++ -lm -Wall monprog.cpp

On peut vérifier la présence dans le répertoire courant du fichier nouvellement créé *a.out*, à l'aide de la commande *ls*.

2.2 Exécution

C'est l'étape durant laquelle la machine se met à exécuter une à une les instructions, dans l'ordre où elles lui sont indiquées et à l'issue de laquelle deux cas se présentent :

- l'exécution s'arrête en cours de route, en émettant un commentaire lapidaire. Il faut alors comprendre ce qui ne va pas : regarder jusqu'où le programme s'est déroulé correctement pour repérer l'endroit où survient le problème, au besoin en ajoutant des impressions régulièrement réparties. Relire aussi le fichier source en vérifiant les points qui sont le plus souvent à l'origine des erreurs et dont une liste est donnée à l'annexe **Erreurs les plus fréquentes**.
- l'exécution va à son terme et fournit des résultats. Bien sûr ce n'est pas parce que la machine fournit un résultat qu'il est juste et c'est alors à l'utilisateur d'utiliser toute sa perspicacité pour en décider. Ce qui est sûr c'est que la machine a fait ce qu'on lui a dit de faire, qui n'est pas forcément ce qu'on croit lui avoir dit. Elle ne se trompe pas aléatoirement, même rarement, ce qui est assez remarquable, vu le nombre gigantesque d'opérations élémentaires que nécessite l'exécution du moindre programme.

Remarque

Il y a deux sortes de langage :

- les langages compilés : toutes les instructions doivent être fournies à l'ordinateur avant qu'il ne commence à exécuter (exemples : C, Fortran, Pascal)
- les langages interprétés : l'exécution se fait immmédiatement après chaque instruction fournie (exemples : shell Linux, Maple, Mathematica, Python)

En pratique, pour faire exécuter le programme écrit dans le fichier *monprog.cpp*, il faut écrire la commande :

./a.out

On obtient alors sur l'écran :

Resultat : 0.666667

En réalité, il existe une abréviation[6] nommée *ccc*, créée pour l'environnement du Magistère, qui permet de compiler et d'exécuter en une seule commande :

ccc monprog.cpp

6. Plutôt qu'abréviation on emploie le terme « fichier de commande » .

fait apparaître directement le résultat à l'écran. C'est cette commande *ccc* qu'on utilisera, sauf cas particulier[7].

2.3 Indications complémentaires sur les éléments d'un programme

2.3.1 Marqueurs, directives, instructions de déclaration, instructions exécutables

Dans le programme *monprog.cpp* précédent on peut distinguer quatre types d'éléments :

- `int main()`, `{` et `}` sont de simples marqueurs, ils indiquent où commence et finit le programme principal. On dit que `int main()` est un « en-tête » et que `{` et `}` délimitent un « bloc » .
- `#include<iostream>` est une « directive » . Elle doit être placée avant la partie du fichier à laquelle elle doit s'appliquer (ici avant le programme principal). Elle doit être écrite en commençant au début de la ligne et il ne peut y en avoir deux par ligne. Elle est prise en compte lors de la compilation.
- `double x, y, z;` est une instruction de déclaration. Elle est prise en compte lors de la compilation. Les déclarations situées à l'intérieur du bloc contenant les instructions du programme principal doivent être placées au début de ce bloc.
- `x = 2.; y = 3.; z = x/y; cout << "Résultat : " << z << endl; return 0;` sont des instructions exécutables, prises en compte seulement lors de l'exécution.

Une instruction doit être terminée par un point virgule. Elle peut s'étendre sur plusieurs lignes, elle n'est pas terminée tant que le point virgule n'est pas rencontré. Inversement on peut regrouper plusieurs instructions sur la même ligne, à condition de les séparer par des points virgule. Il peut être intéressant de le faire, pour éviter que le fichier ne s'étire trop en longueur. Mais pour assurer la lisibilité du programme il ne faut regrouper que des instructions ayant entre elles un certain rapport de sens (par exemple l'attribution de valeur pour `x = 2.; y = 3.; z = x/y;` dans l'exemple).

2.3.2 Commentaires

Il existe trois façons d'insérer des commentaires dans le programme, c'est à dire du texte qui n'est pas pris en compte par l'ordinateur mais sert à l'humain qui écrit ou utilise le programme. Ce peut être des commentaires au sens premier du mot ou des portions de programme que l'utilisateur souhaite neutraliser provisoirement.

1. Deux slash contigüs transforment en commentaire tout ce qui les suit sur leur ligne :

   ```
   x = 2.5; // x est exprimé en mètres
   ```

   Ne permet de commenter qu'une ligne à la fois, pratique pour mettre en commentaire de très petites portions de programme.

2. Toute ce qui est compris entre `/*` et `*/` est transformé en commentaire :

   ```
   /*
   a = b*c;
   f = sin(u);
   */
   x = 2.5;
   ```

   Les instructions `a = b*c; f = sin(u);` sont neutralisées. Permet de commenter autant de lignes que l'on veut d'un seul coup mais on ne peut pas imbriquer :

   ```
   /*
   ...
   /*
   ...
   */
   ...
   */
   ```

   ne fonctionne pas, ce qui est gênant pour des programmes un peu longs.

3. Toute ce qui est compris entre `#if 0` et `#endif` est transformé en commentaire (existe uniquement en C++) :

7. Par exemple si *monprog.cpp* n'a pas été modifié depuis la dernière exécution, l'utilisation de *ccc* recompile inutilement. Vu que *ccc* supprime le fichier *a.out* après l'avoir exécuté, il est mieux d'utiliser explicitement la commande *g++* donnée ci-dessus pour compiler le programme une fois et puis l'exécuter autant de fois qu'on veut avec *./a.out*, si l'on compte exécuter le même programme plusieurs fois.

```
#if 0
a = b*c;
f = sin(u);
#endif
x = 2.5;
```

Même utilisation que `/* ... */`[8], mais on peut imbriquer.

2.3.3 Distinction minuscules-majuscules

En C il y a, comme dans *Linux* , distinction entre les minuscules et les majuscules.

3 Conseils de programmation

3.1 Ecriture

Il faut avoir défini une méthode, c'est à dire un algorithme complet, pour parvenir au but recherché avant de commencer à écrire les instructions (cet algorithme est généralement largement ou totalement indépendant du langage utilisé, C, Pascal, Fortran, Maple, etc. pour la programmation proprement dite). Au fur et à mesure de l'écriture, raisonner rigoureusement pour savoir si le programme va bien faire exactement ce qu'on veut en le faisant exécuter mentalement, comme si on était l'ordinateur. Il ne faut surtout pas écrire des instructions qui doivent approximativement faire le travail en se disant qu'on corrigera ensuite selon les diagnostics fournis par le compilateur et les résultats obtenus. En effet les diagnostics ne sont pas toujours clairs, les comprendre peut être laborieux. Ce qui est plus grave, une erreur du programmeur peut avoir, pour le compilateur, un sens, mais tout a fait différent du bon. Il n'y a évidemment pas, alors, de diagnostic et les résultats sont faux. Si l'utilisateur a un moyen de les estimer il s'aperçoit qu'il y a une erreur, sinon celle-ci passe inaperçue jusqu'à ce qu'un calcul indépendant soit fait, que le pont s'écroule ou que la fusée s'écrase.
Enfin il faut séparer au maximum les tâches indépendantes c'est à dire rendre le programme le plus modulaire possible. Le C n'exige rien de ce point de vue et c'est à l'utilisateur de bien structurer ses programmes.

3.2 Relecture

Le plus difficile n'est pas d'écrire un programme, mais de le relire après un certain temps pour comprendre ce qu'il fait et comment il le fait. Même pour un programme que l'on a écrit soi-même et dont on connaît exactement la finalité, après quelques mois il peut être très long de comprendre l'algorithme employé et sa programmation à partir de la seule lecture des instructions pour, par exemple, modifier ou développer sans faire d'erreurs.

Il faut, en tête du fichier, mettre un commentaire expliquant quelle est la tâche effectuée par le programme et suivant quel algorithme. S'il n'est pas possible d'écrire ceci en commentaire il faut citer une référence.

Il faut programmer de façon claire et naturelle en évitant les astuces inutiles, ne pas chercher à trop condenser sauf si cela apporte réellement un gain et dans ce cas mettre des commentaires. Inversement il faut bannir les instructions et les commentaires inutiles. Cela est encore plus vrai quand le programme doit être utilisé et modifié par d'autres.

4 Lexique Français-Anglais

identifiant = login name
mot de passe = password
répertoire initial = home directory
répertoire courant = working directory
répertoire racine = root directory
fichier de commande = script

8. Sauf que `#if 0` et `#endif` doivent se trouver chacun seuls sur une ligne, ce qui n'est pas nécessaire pour `/*` et `*/`.