Langage C et Programmation Système

Jean-Baptiste.Yunes@univ-paris-diderot.fr

U.F.R. Informatique, Université Paris Diderot

20 Septembre 2016

v0.5

Let it be — The Beatles —

When I find my code in tons of trouble,
Friends and colleagues come to me,
Speaking words of wisdom:
"Write in C."

As the deadline fast approaches,
And bugs are all that I can see,
Somewhere, someone whispers:
"Write in C."

Write in C, Write in C,
Write in C, oh, Write in C.
LOGO's dead and buried,
Write in C.

I used to write a lot of FORTRAN,
For science it worked flawlessly.
Try using it for graphics!
Write in C.

If you've just spent nearly 30 hours
Debugging some assembly,
Soon you will be glad to
Write in C.

Write in C, Write in C,
Write in C, yeah, Write in C.
Only wimps use BASIC.
Write in C.

Write in C, Write in C
Write in C, oh, Write in C.
Pascal won't quite cut it.
Write in C.

Write in C, Write in C,
Write in C, yeah, Write in C.
Don't even mention COBOL.
Write in C.

Brian Marshall, 1987.

Contenu

À propos du cours...

Mail :
Jean-Baptiste.Yunes@univ-paris-diderot.fr
Web :
www.irif.univ-paris-diderot.fr/~yunes
Moodle :
moodlesupd.script.univ-paris-diderot.fr
cours C6

Type	Horaire	Lieu	Horaire	Lieu	Enseignant
Cours	Je 8:30–11:30	Amφ 13E			Jean-Baptiste Yunès
TP Gr. 1	Lu 13:30–15:30	2031	Ve 8:30–10:30	2031	Maël Canu
TP Gr. 2	Je 15:00–17:00	2031	Ve 8:30–10:30	2032	Benjamin Hellouin
TP Gr. 3	Ma 9:00–11:00	2032	Ma 11:00–13:00	2032	Constantin Enea
TP MI	Je 13:00–15:00	2032	Ve 15:30–17:30	2032	Benjamin Hellouin

Question

Comment je peux obtenir les crédits de cette UE ?

En travaillant et en obtenant au moins 10/20.

Question

Comment je fais pour obtenir 10/20 ?

En travaillant en TP (contrôle continu, *aka* CC), sur le projet (*aka* P) et en composant à l'examen (*aka* E). La formule est alors :

Definition (*Secretum Secretorum*, *Dura Lex, Sed Lex*)

$$\text{Note}_{\text{finale}} = 0,2 \times CC + 0,4 \times P + 0,4 \times E$$

Objectifs

1. Programmer dans un langage de (relativement) bas-niveau
   - savoir manipuler finement la mémoire
2. Utiliser la ligne de commande
   - savoir commander le système
3. Manipuler le système de fichiers
   - savoir écrire des commandes relatives au SF

Autour de `Hello World`...

Le langage C

Le langage C est rangé dans la catégorie des **langages impératifs procéduraux**

Definition (impératif (Larousse))

Nécessité absolue qui impose certaines actions comme un ordre...

Definition (langage impératif)

Permet la description de tâches en séquences d'instructions modifiant l'état de la machine

Le langage C est rangé dans la catégorie des **langages impératifs procéduraux**

Definition (procédural (Larousse))

Marche à suivre..., succession d'opérations à exécuter... synonyme : sous-programme

Definition (sous-programme (Larousse))

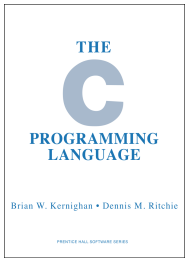
Séquence utilisée en plusieurs points...

Definition (langage procédural)

Qui favorise la factorisation du code et sa réutilisabilité via les procédures, fonctions, routines, sous-routines,...

Caractéristiques

- Créé en ~1972 par Dennis MacAllistair Ritchie (†2011), l'historique K&R



Source : wikipedia

- Utilisé pour écrire le système Unix
- Encore très utilisé
- Linux est écrit en C... (voilà c'est dit)

- Standardisé

1989	1990	1999	2011
ANSI-C	ISO-C	ISO C99	ISO C11

- Compilé
- Disponible sur (presque) toutes les plate-formes

Exemple

```
/*
 * bonjour.c
 * Author: JBY
 */
#include <stdio.h>

int main(void) {
  puts("Bonjour tout le monde.");
  return 0;
}
```

- commentaires
- inclusions de déclarations
- point d'entrée

Compilation

La **compilation** (et l'édition de liens) consiste à transformer le **code source** en un **exécutable**.
La compilation de l'exemple peut-être obtenue *via* :

```
$ gcc -o bonjour bonjour.c
```

Et son exécution :

```
$ ./bonjour
Bonjour tout le monde.
$
```

Note

On emploiera dans toute la suite de ce cours, le compilateur du projet GNU : gcc (GNU C Compiler)

Recommandations

- N'utilisez pas d'IDE (outil de développement intégré, Eclipse ou autre) pour ce cours, *i.e.* pas de compilation avec `F5`!
- Utilisez un éditeur de code type `emacs` ou `vi`. Ce sont des outils puissants, apprenez à les maîtriser.
- Utilisez la ligne de commande, votre expertise du système n'en sera que meilleure. C'est très important!
- Apprenez à compiler en commandant vous-même `gcc`.

La maîtrise de ces outils vous sera bénéfique à long terme...
Vous serez aussi évalués sur ces points.

Exemple

```c
// bonjour2.c
#include <stdio.h>
#include <string.h>

int main(void) {
  char nom[100];
  memset(nom,0,sizeof(nom));
  puts("Quel␣est␣ton␣prénom␣?");
  if (fgets(nom,100,stdin)!=NULL && nom[0]!=
      '\n') {
    nom[strlen(nom)-1] = '\0';
    fputs("Bonjour␣",stdout); puts(nom);
  } else
    puts("Bonjour␣tout␣de␣même!");
  return 0;
}
```


Cette fois la compilation est obtenue *via* :

```
$ make bonjour2
cc      bonjour2.c    -o bonjour2
$
```

⚠ Attention

Nous verrons plus tard pourquoi cela fonctionne. Cette ligne de commande, sans autre configuration préalable, ne fonctionne que pour des compilations élémentaires.

L'exécution produit :

```
$ ./bonjour2
Quel est ton prénom ?
dagobert
Bonjour dagobert
$
```

Question

Comment puis-je connaître la façon d'utiliser une fonction (par exemple `puts` ?)

Réponse : un manuel est disponible...

Les manuels (généralités)

les systèmes *NIX sont normalement livrés avec un ensemble de manuels. Ceux-ci sont rangés par catégories (*sections*), et on retrouve au moins :

1. commandes utilisateur,
2. API des appels systèmes,
3. API des bibliothèques logicielles,
4. fichiers spéciaux,
5. formats,
6. jeux,
7. inclassables,
8. commandes d'administration.

Il existe souvent d'autres sections, cela dépend de ce qui est installé.

Pour obtenir l'affichage d'un ou plusieurs manuels, ou des informations à propos du manuel, il faut faire appel à la commande `man`.

`man` `[-s` *section*`]` *mot*

permet d'obtenir le manuel correspondant au `mot` spécifié, la recherche étant éventuellement limitée à la `section` indiquée.
Le manuel étant disponible *via* une commande, il est possible d'obtenir le manuel du manuel...

Manuel

Pour effectuer des affichages nous avons utilisé deux fonctions `fputs` et `puts`. Il y a en a encore d'autres comme `fprintf` et `printf`, `fputc`/`putc` et `putchar`. Nous étudierons plus tard et en détail ces fonctions et surtout leurs arguments. Pour s'en faire une idée, on peut consulter leur manuel en ligne de commande :

Exemple

```
$ man fputs
FPUTS(3)     BSD Library Functions Manual
   FPUTS(3)

NAME
     fputs, puts -- output a line to a
        stream
...
```

Qu'y a t-il dans le manuel ? Les informations les plus utiles (dans un premier temps) :

HEADER l'entrée du manuel (`fputs`) et la section concernée, `(3)` les fonctions standards,

SYNOPSIS les directives d'inclusions utiles et les prototypes des fonctions concernées (même manuel pour plusieurs fonctions),

DESCRIPTION la sémantique des fonctions et leur mode opératoire

RETURN VALUES les valeurs retournées en cas de succès ou d'erreurs

...

fputs/puts

src — man fputs — man — less ‹ man fputs — 80×29

```
FPUTS(3)                 BSD Library Functions Manual                 FPUTS(3)

NAME
     fputs, puts -- output a line to a stream

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <stdio.h>

     int
     fputs(const char *restrict s, FILE *restrict stream);

     int
     puts(const char *s);

DESCRIPTION
     The function fputs() writes the string pointed to by s to the stream
     pointed to by stream.

     The function puts() writes the string s, and a terminating newline char-
     acter, to the stream stdout.

RETURN VALUES
     The functions fputs() and puts() return a nonnegative integer on success
     and EOF on error.
:
```

fputc/putc/putchar

src — man putc — man — less ‹ man putc — 80×29

```
PUTC(3)                 BSD Library Functions Manual                 PUTC(3)

NAME
     fputc, putc, putc_unlocked, putchar, putchar_unlocked, putw -- output a
     character or word to a stream

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <stdio.h>

     int
     fputc(int c, FILE *stream);

     int
     putc(int c, FILE *stream);

     int
     putc_unlocked(int c, FILE *stream);

     int
     putchar(int c);

     int
     putchar_unlocked(int c);

:
```


fprintf/printf/sprintf

src — man 3 printf — man — less ‹ man 3 printf — 80×29

```
PRINTF(3)               BSD Library Functions Manual               PRINTF(3)

NAME
     printf, fprintf, sprintf, snprintf, asprintf, dprintf, vprintf, vfprintf,
     vsprintf, vsnprintf, vasprintf, vdprintf -- formatted output conversion

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <stdio.h>

     int
     printf(const char * restrict format, ...);

     int
     fprintf(FILE * restrict stream, const char * restrict format, ...);

     int
     sprintf(char * restrict str, const char * restrict format, ...);

     int
     snprintf(char * restrict str, size_t size, const char * restrict format,
         ...);

     int
     asprintf(char **ret, const char *format, ...);
:
```

Affichage

On remarque que :

`*put*` désigne les fonctions qui affichent **directement** une donnée, *«output a line (or a character) to a stream»*,

`*print*` désigne les fonctions qui permettent d'afficher des données **formatées**, *«formatted output conversion»*.

printf

Exemple

```
printf("Un texte avec des trous, ici:%s, ou
   ici:%d, pour y placer des valeurs\n","
   message",34);
```

Definition (Directives de formatage)

Les «trous» spécifient quel type de conversion ou formatage doit être appliqué à la donnée correspondante (premier «trou», premier argument qui suit, second/second, etc).
Quelques formats simples :

`%d` argument entier affiché en décimal
`%x` argument entier affiché en hexadécimal
`%f` argument double affiché en notation pointée
`%s` argument *chaîne de caractères*

Back to *NIX

Question

Pourquoi dans certains cas suis-je obligé de taper la commande `./bonjour` (comme dans les exemples précédents) pour démarrer l'exécution de mon programme compilé ?
Alors même que pour démarrer l'exécution du compilateur, il me suffit de taper `gcc`.

Réponse : on interagit avec le système *via* un *shell* .
C'est-à-dire une application particulière, dont il faut comprendre le fonctionnement...

shell

Architecture fonctionnelle d'un système *NIX

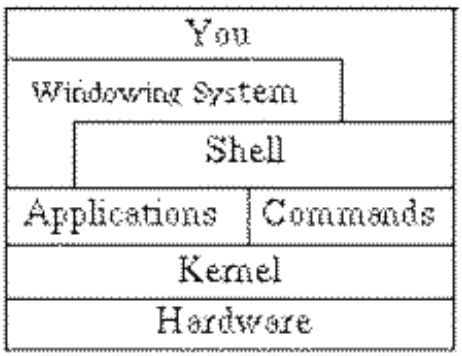


FIGURE 1. UNIX Layers

source : developeriq.in

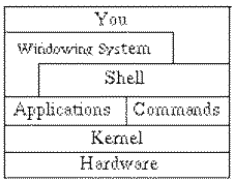


FIGURE 1. UNIX Layers

source : developeriq.in

Comme indiqué il y a deux façon d'interagir *manuellement* avec le *système* : *via* l'interface graphique ou *via* la *ligne de commande*.
Dans ce cours, on oubliera le paradigme des interfaces graphiques, pour se concentrer sur la **ligne de commande**.

Les shells

Un shell n'a rien de particulier du point de vue du système lui-même, c'est une application comme une autre. L'utilisateur, lui, fait la différence de part la façon d'interagir avec cette application. Il existe donc toute une variété de shells. Un bon exercice est de tenter d'en écrire un soi-même.
Attention, si écrire un shell élémentaire est chose aisée, un shell se comportant correctement dans toutes les situations envisageables est bien plus complexe qu'il n'y paraît.

familles de shells

Dans les toutes premières versions d'Unix, deux shells sont apparus successivement :

Thompson shell - 1971–1975
Kenneth Lane "Ken" Thompson

Source : wikipedia

Source : gcn.com

PWB Shell - 1975–1977
John Mashey
Using a Command Language as a High-Level Programming Language
2nd International Conference on Software Engineering, 1976: pp. 169–176.

C'est Steve Bourne qui a réécrit le shell pour la distribution Unix V7 de 1979. Son œuvre est restée car les *shells* les plus couramment utilisés sont issus de celui-ci.

Bourne shell - 1976–
Stephen Richard "Steve" Bourne
The UNIX shell is both an interactive command interpreter and a programming language
BYTE Magazine, 1982. pp. 197–204

Source : wikipedia

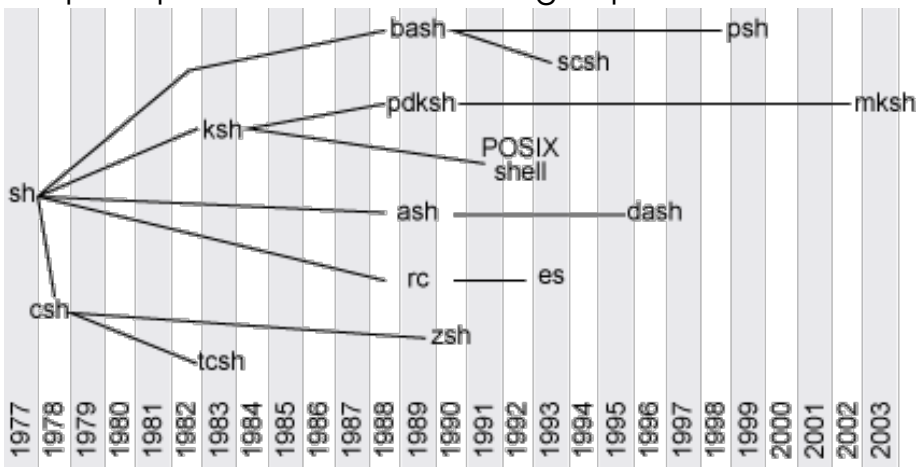
Il existe une seconde famille importante de *shells*, les C-shells qui furent initiés dans le projet BSD (une branche importante de la famille *nix). Leurs syntaxes diffèrent quelque peu du Bourne shell.

C-shell - 2BSD - 1978–
William Nelson "Bill" Joy

Source : wikipedia

On notera que Bill Joy a été le co-fondateur de la société Sun Microsystems.

Les principaux shells sont ainsi regroupés :



Source : ibm.com

Definition (Shell)

Un *shell* est un **interpréteur de commandes** fonctionnant en mode **interactif** avec l'utilisateur ou en mode automatique d'exécution de **scripts**.
Son mode principal d'interaction est une boucle infinie constituée de l'**entrée d'une commande** puis de la **réalisation** de celle-ci.
Les commandes doivent être spécifiées dans une **syntaxe** qui lui est propre, *i.e.* chaque *shell* possède sa syntaxe...

En mode interactif, le shell signale son attente à l'aide d'une invite (*prompt*). Dans la suite celle-ci est `$`.

Anatomie d'une commande

La syntaxe la plus commune des *shells* pour l'exécution (simple) d'une commande est :

`[`*chemin*`/]` *argument* `[`*argument*`...]`

Exemple

```
$ ls -ail
...
$ /bin/ls -l toto
...
```

La commande `$ ls -ail f` est constituée de 3 **arguments**: le nom du programme à exécuter suivi par des paramètres à transmettre.

- `ls` qui permet de désigner un exécutable à charger puis démarrer. C'est le premier argument du processus!
- `-ail` que l'on appelle parfois une option mais qui n'est, formellement, qu'un argument comme un autre. Une option permet d'obtenir une variante d'exécution, et la convention est de les préfixer par le caractère – pour les distinguer des autres arguments.
- `f` qui est un argument correspondant ici à un «fichier» qui sera traité par le processus.

Ces deux commandes sont **interactives**, c'est-à-dire que soit le shell donne la main à un programme qui s'exécute et en attend la terminaison avant de procéder à l'attente d'une autre entrée (la présence synchronisée de l'invite est claire sur ce point), soit le shell réalise lui-même la commande (alors dite interne) et reprend l'attente d'une nouvelle entrée.
La première ligne de commande possède deux arguments, alors que la seconde en possède trois.

Attention

On parle parfois de commande, d'options et d'arguments, mais du point de vue de la syntaxe, il n'y a pas de différence. Le shell découpe la ligne en lexèmes (*tokens*) chacun d'eux constituant un argument. Ce que l'on appelle habituellement la commande est le premier argument de la ligne commande, mais il s'agit d'un argument permettant de désigner un exécutable ou une commande interne du shell. La distinction entre option et argument concerne la «commande» elle-même.

Pour la seconde commande, la spécification est claire (?), il s'agit de la réalisation de l'exécution d'un exécutable dont le code est contenu dans le fichier `/bin/ls`, avec des arguments qui seront traités par elle.
Pour la première commande, la spécification de la commande à exécuter ne contient pas d'informations de localisation de l'exécutable.

Question

Comment cette commande est-elle retrouvée par le shell ?

Pour y répondre, il faut parler du **shell** et de son **environnement**.

Environnement (du shell)

Definition (Environnement)

Il constitue une partie du **contexte** dans lequel les programmes s'exécutent et en particulier les shells, pour lesquels il existe des commandes (internes) permettant de les manipuler.
L'environnement est constitué de clés associées à des valeurs, on parle de **variables d'environnement**.

Cet environnement permet par exemple d'indiquer avec quel type de terminal l'utilisateur interagit, où les fichiers temporaires doivent être placés, quelle est l'identité de l'utilisateur, dans quelle langue l'utilisateur souhaite obtenir les messages, etc.

Exemple

```
$ env
...
TERM=xterm-256color
SHELL=/bin/zsh
HOME=/Users/yunes
LOGNAME=yunes
USER=yunes
PATH=/Library/Java/JavaVirtualMachines/jdk1.8.0_40.
    jdk/Contents/Home/bin:/usr/local/bin:/usr/bin:/bin
    :/usr/sbin:/sbin:/opt/X11/bin:/usr/local/go/bin:/
    usr/local/MacGPG2/bin:/Library/TeX/texbin::/usr/
    local/bin:/usr/local/mysql/bin:/Users/yunes/
    Library/Haskell/bin:/Users/yunes/.cabal/bin:/Users
    /yunes/.gem/ruby/2.0.0/bin
SHLVL=1
PAGER=less
LC_CTYPE=fr_FR.UTF-8
LSCOLORS=Gxfxcxdxbxegedabagacad
$
```

La commande **env** permet d'obtenir la liste des variables de l'environnement courant.
Ainsi `TERM=xterm-256color` indique qu'une variable `TERM` est définie et a pour valeur associée `xterm-256color`.
L'affichage d'une variable peut s'effectuer à l'aide de la commande interne `echo` :

Exemple

```
$ echo $LOGNAME
yunes
$
```

La variable qui nous intéresse maintenant est `PATH`.

PATH

La variable d'environnement `PATH` est utilisée par le *shell* afin de trouver un exécutable correspondant à un nom de commande, c'est-à-dire lorsque le premier lexème de la commande ne spécifie pas de chemin (*i.e.* ne contient pas de caractère `/`).
La valeur de cette variable est une suite de chemins chacun désignant un répertoire et tous séparés par le caractère `:`.

Exemple

```
$ echo $PATH
PATH=/usr/local/bin:/usr/bin:/bin
$
```

La valeur `/usr/local/bin:/usr/bin:/bin` correspond à la **suite ordonnée** des répertoires `/usr/local/bin`, `/usr/bin` et `/bin`.
Lorsqu'un simple nom de commande est entré en premier argument de la ligne, la commande sera alors recherchée dans chacun des répertoires spécifiés et dans l'ordre indiqué. Ainsi dès qu'un exécutable de nom correspondant y sera trouvé le système procèdera à son exécution dans le contexte approprié et avec les arguments spécifiés en ligne de commande.
Si l'exécution n'est pas possible ou qu'il n'existe pas de commande de ce nom dans les répertoires indiqués un message d'erreur approprié sera retourné.

Exemple

```
$ blabla
zsh: command not found: blabla
$
```

On notera que le format de ce message est spécifique à chaque *shell* mais que le sens général est toujours le même dans ce cas : `command not found`.

⚠ Attention

Il doit être clair que cela ne signifie pas qu'il n'existe pas d'exécutable de ce nom dans le système, mais juste qu'il n'existe pas d'exécutable de ce nom dans la liste des répertoires spécifiés par la variable `PATH`.

Question

Pourquoi dans certains cas suis-je obligé de taper la commande `./bonjour` (comme dans les exemples précédents) pour démarrer l'exécution de mon programme compilé ?
Alors même que pour démarrer l'exécution du compilateur, il me suffit de taper `gcc`.

Les éléments entrevus permettent de répondre à la seconde partie de la question (sous réserve que le compilateur est accessible depuis l'un des répertoires spécifiés dans `PATH`), mais quid de la partie principale ?

Il suffit simplement que la variable `PATH` désigne le répertoire contenant l'exécutable que l'on vient de compiler, soit de façon absolue, soit en indiquant que l'on désire inclure systématiquement le répertoire de travail, c'est-à-dire la référence relative `.`.

Attention

Ne pas oublier que l'ordre est important! D'autre part, il est souvent considéré comme dangereux d'inclure le répertoire de travail dans la variable `PATH`.

La question des références absolues/relatives est traitée à partir de la page 440.

Exemple

```
$ ls
bonjour
$ bonjour
bonjour: command not found
$ ./bonjour
Bonjour
$ PATH=$PATH:.
$ bonjour
Bonjour
```

Inclure un chemin «vide» permet d'obtenir le même effet.

La modification d'une variable de l'environnement s'effectue *via* l'affectation (dans les *shells* de la famille Bourne).

Exemple

```
$ echo $TERM
xterm-256color
$ TERM=blabla
$ echo $TERM
blabla
$
```

Potacherie

Une bonne blague consiste à modifier la variable `PATH` de votre camarade à l'aide de la commande (interne) :

```
$ PATH=
$
```

Pourquoi ?
Est-il possible de se sortir d'une telle situation ?
Comment ?

Environnement exportable ou non

Attention

Le contexte des exécutions ne définit qu'un seul type d'environnement (celui accessible *via* `env` par exemple). Mais les *shells* utilisent deux types de variables : celle de l'environnement et celles qui leurs sont propres.
Si l'on tente l'affectation d'une variable qui n'existe pas celle-ci est créée mais ne fait pas partie de l'environnement. Pour la placer dans l'environnement il est nécessaire de le spécifier explicitement *via* la commande `export` (dans les *shells* de la famille Bourne).

Exemple

```
$ env
pas de variable FOO dans l'environnement
$ FOO=bar
$ env
pas de variable FOO dans l'environnement
$ echo $FOO
bar
$ export FOO
$ env
...
FOO=bar
...
$
```


Arborescence des fichiers

Nous avons aussi parlé de références, fichiers, etc. Comment sont-ils organisés ?

⚠ Attention

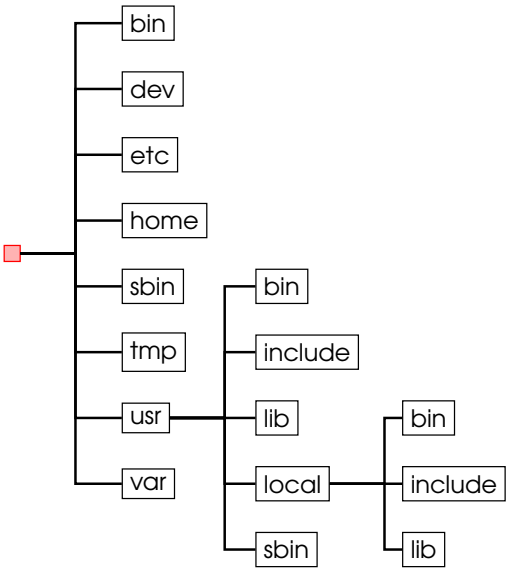
La description qui suit est une vision simplifiée de la nature des fichiers. Elle correspond à l'intuition «naturelle» de l'organisation de données, mais elle est trompeuse sur de nombreux points comme on le verra plus tard.

Ainsi les «fichiers» sont «rangés» dans une structure hiérarchique appelée **arborescence des fichiers**. Il s'agit ni plus ni moins d'organiser le rangement en dossiers, sous-dossiers et documents. C'est une façon assez naturelle de classer, même si celle-ci n'est plus aujourd'hui totalement satisfaisante.
D'autres systèmes utilisent les termes de «dossiers» et «documents» (avec des raffinements divers - OSX, WINDOWS).

Les systèmes contenant de nombreux fichiers dont les contenus sont particuliers, l'organisation hiérarchique reflète souvent les distinctions d'usage:

- exécutables, commande ou applications,
- configuration,
- traces,
- données...

Un bout d'une arborescence typique *NIX.



`bin`

Ces répertoires contiennent normalement des «binaires» (**bin**aries) exécutables, c'est-à-dire le code machine exécutable de commandes diverses.
Par exemple: `/bin/ls`
Il existe différents répertoires `bin`:

`/bin` pour les binaires «minimaux», c'est-à-dire considérés comme essentiels pour un fonctionnement minimal du système;

`/usr/bin` pour les binaires d'usage général fournis par le système d'exploitation;

`/usr/local/bin` pour les binaires d'exécutables propres à l'instance du systèmes d'exploitation, c'est ici qu'en général on «installe».

sbin

Ces répertoires contiennent des binaires dont l'usage est habituellement réservé à l'administration du système (**s**ystem **bin**aries).
Ces répertoires sont aussi déclinés en `/sbin`, `/usr/sbin` et `/usr/local/sbin`.

/dev

Ce répertoire (il est normalement unique) contient l'ensemble des périphériques (**dev**ices) théoriquement accessibles sur l'instance installée.
On reviendra plus tard sur ces fichiers très spéciaux; mais mentionnons dès maintenant l'existence :

`/dev/null` le puits sans fond...

`/dev/zero` la source inépuisable d'octets nuls...

`/dev/random` la source inépuisable d'octets aléatoires...

etc

Ces répertoires contiennent habituellement les fichiers de configuration de divers logiciels et outils.

/var

Ce répertoire (**var**iable) contient normalement des données utilisées durant l'exécution du système d'exploitation.

/tmp

Ce répertoire contient des fichiers temporaires (**temp**oraries). La notion de «temporaire» étant discutable, il s'agit ici de fichiers qui ne survivront pas à l'extinction du système.

lib

Ces répertoires contiennent des bibliothèques logicielles. On étudiera leur rôle plus tard, mais ils sont essentiels au processus de production d'exécutable.

include

Ces répertoires contiennent des fichiers d'entête à inclure (to **include**) lors de compilations. Nous étudierons aussi leur rôle plus tard.

Noms de fichier

Sous *NIX et du point de vue de l'utilisateur, les fichiers portent des noms.
Selon les variantes et versions des systèmes d'exploitation ainsi que des systèmes de fichiers employés, les caractères autorisés dans la composition d'un nom de fichier (*filename*) peuvent varier. Sous *NIX, il est toutefois strictement interdit d'employer les caractères `NUL` (`\0` de code ASCII 0) et `/` (code ASCII 47).

Attention

Il vaut mieux éviter d'employer des caractères trop «exotiques» comme les caractères de contrôles, les guillemets divers, la plupart des signes... On se limite généralement aux caractères alphanumériques, au tiret `-`, au souligné `_`, au caractère d'espacement et au point `.`.

Question

Comment s'en sortir avec un fichier qui s'appelle `-i` ou `-l` ?

chemin, nom de fichier

Definition (chemin)

Un **chemin** (*pathname*) permet de désigner un objet du système de fichier à partir d'un répertoire de départ. Un chemin est constitué d'une suite ordonnée de nom de fichiers séparés par le caractère `/` et commençant possiblement par ce même caractère, par exemple: `/bin/ls`, `/usr/local/zip`, `toto` ou `cours/langc/exemple/toto.c`.

- Si le chemin commence par `/` le point de départ du chemin est la **racine** du système de fichiers. Il s'agit d'un **chemin absolu**.
- sinon le point de départ est le **répertoire courant** ou **répertoire de travail**. Il s'agit d'un **chemin relatif**.

Note

On remarquera qu'un chemin absolu désigne un objet du système de fichier de façon unique.
Un chemin relatif, sans spécification explicite du point de départ, ne désigne rien de particulier en tant que tel.

Dans chaque répertoire il existe deux noms de fichiers particuliers: `.` et `..` (on ne peut les créer ni faire disparaître soi-même directement comme on le verra plus tard).

Definition (`.`)

Le nom de fichier `.` désigne dans tout répertoire le répertoire lui-même. C'est une auto-référence, comme le mot «moi» ou «je» de la langue française.

Definition (`..`)

Le nom de fichier `..` désigne dans tout répertoire le répertoire qui le contient. C'est un chemin qui permet de «remonter» dans l'arborescence.