

# Cours 1

Quelques applications de l'informatique aux Sciences du Vivant

Organisation des cours

Introduction aux commandes UNIX

Le langage C

Vincent Lepetit

`vincent.lepetit@epfl.ch`

# Organisation des cours

- 1<sup>er</sup> semestre: Septembre → Décembre
  - Apprentissage pratique de la programmation à travers les langages C & C++;
  - 2 heures *ex cathedra*: cours + exercices sur papier;
  - 2 heures d'exercices sur ordinateur.
- 2<sup>ème</sup> semestre (*Jamila Sam - Informatique II*):
  - Programmation Objet en C++.
- 2<sup>ème</sup> année (*Sebastian Gerlach - Informatique III*):
  - Développement de projets en C++.

# Notes

- Examen intermédiaire au début du mois de novembre (coefficient 1).
- Examen final la dernière séance de décembre (coefficient 2).

Sur papier: tout document autorisé, machines interdites.

Compte pour moitié pour la note de l'année en Informatique.

# Accéder aux fichiers: Le système de fichiers sous Unix/Linux

Les fichiers sont organisés en arborescence:

Le répertoire racine est noté /

répertoire = *directory* ≈ Dossier sous Windows

Ce répertoire contient d'autres répertoires:

`home`: contient les répertoires des utilisateurs;

`usr`: contient en particulier les programmes;

`etc`: contient en autres des fichiers de configurations;

`tmp`: contient des fichiers temporaires;

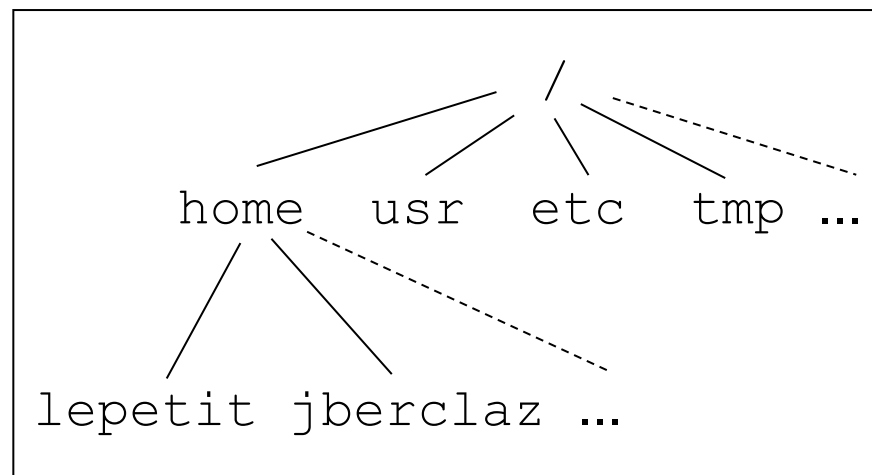
...

Chaque utilisateur dispose d'un répertoire personnel, appelé *home directory*

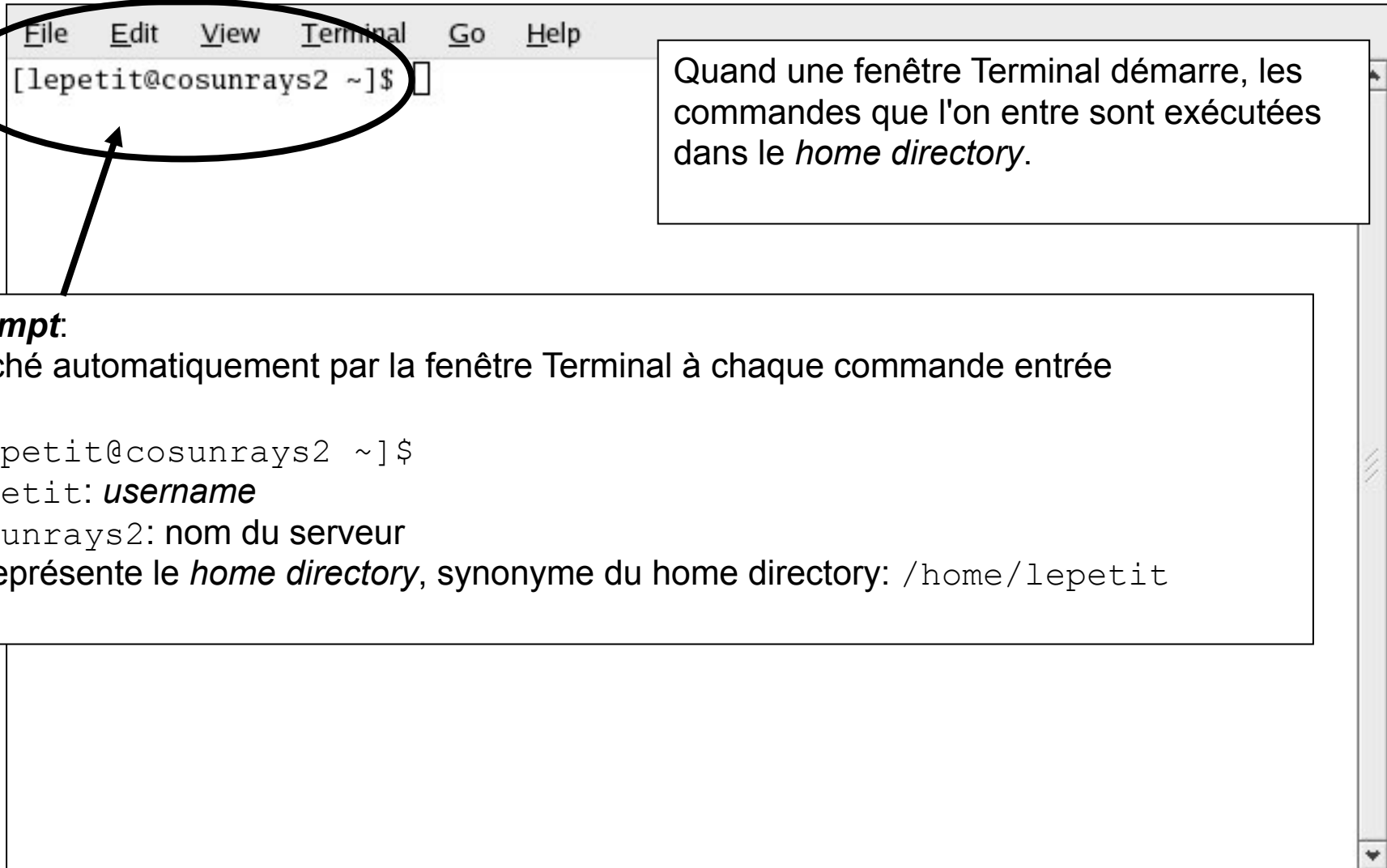
`/home/username`

Ce *home directory* se trouve dans le répertoire `home` et a pour nom le nom de l'utilisateur (*user name*).

Conservé pendant un an, accessible de n'importe quel terminal des salles CO20-22.



# Fenêtre Terminal



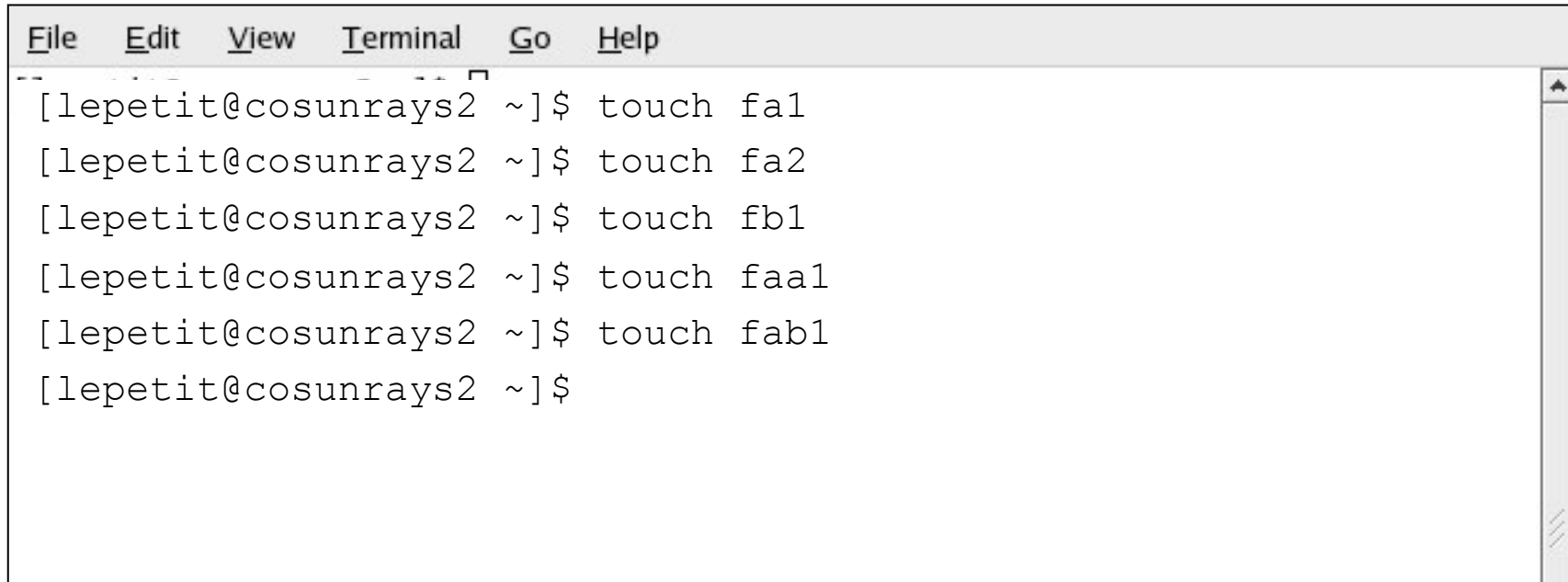
# Une première commande: touch

Pour l'instant, le *home directoy* est vide. Créons des fichiers:  
la commande `touch` permet de créer des fichiers vides.

Par exemple:

```
touch fa1
```

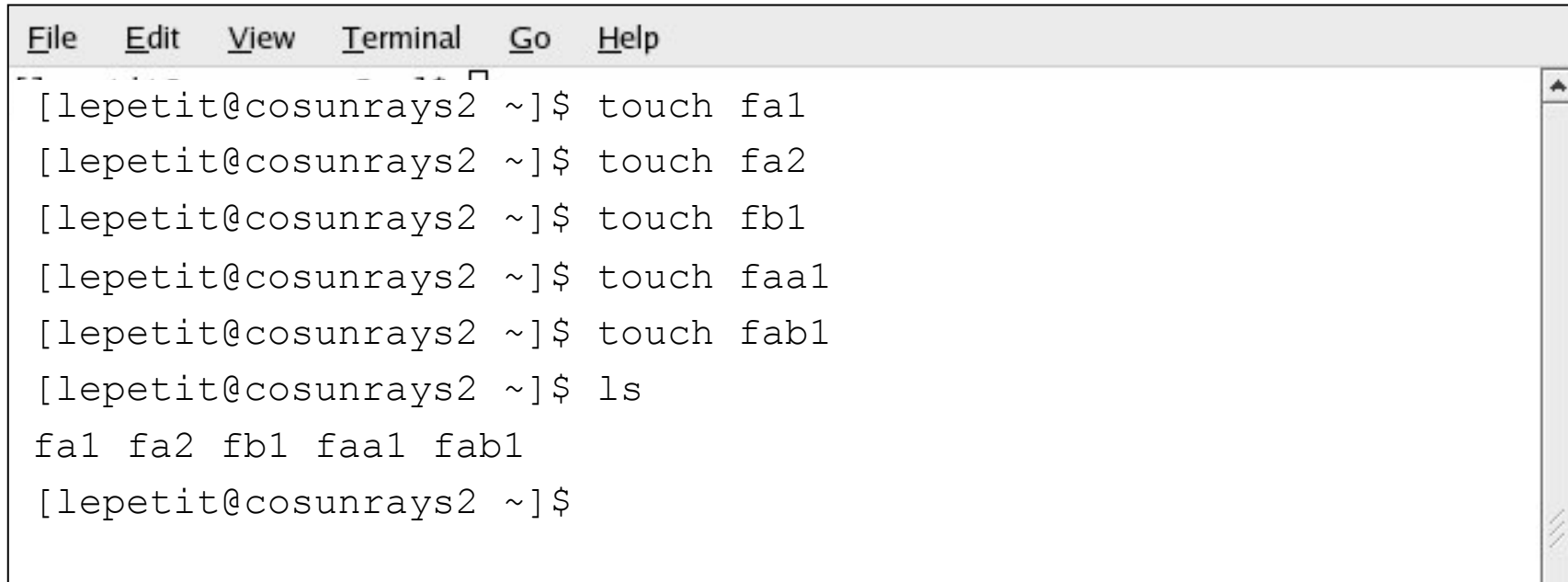
créé un fichier vide appelé `fa1`.



```
File Edit View Terminal Go Help
[lepetit@cosunrays2 ~]$ touch fa1
[lepetit@cosunrays2 ~]$ touch fa2
[lepetit@cosunrays2 ~]$ touch fb1
[lepetit@cosunrays2 ~]$ touch faa1
[lepetit@cosunrays2 ~]$ touch fab1
[lepetit@cosunrays2 ~]$
```

# Lister le contenu d'un répertoire: `ls`

La commande `ls` permet de voir le contenu d'un répertoire:

A terminal window with a menu bar containing 'File', 'Edit', 'View', 'Terminal', 'Go', and 'Help'. The terminal text shows a user creating five files and then listing them.

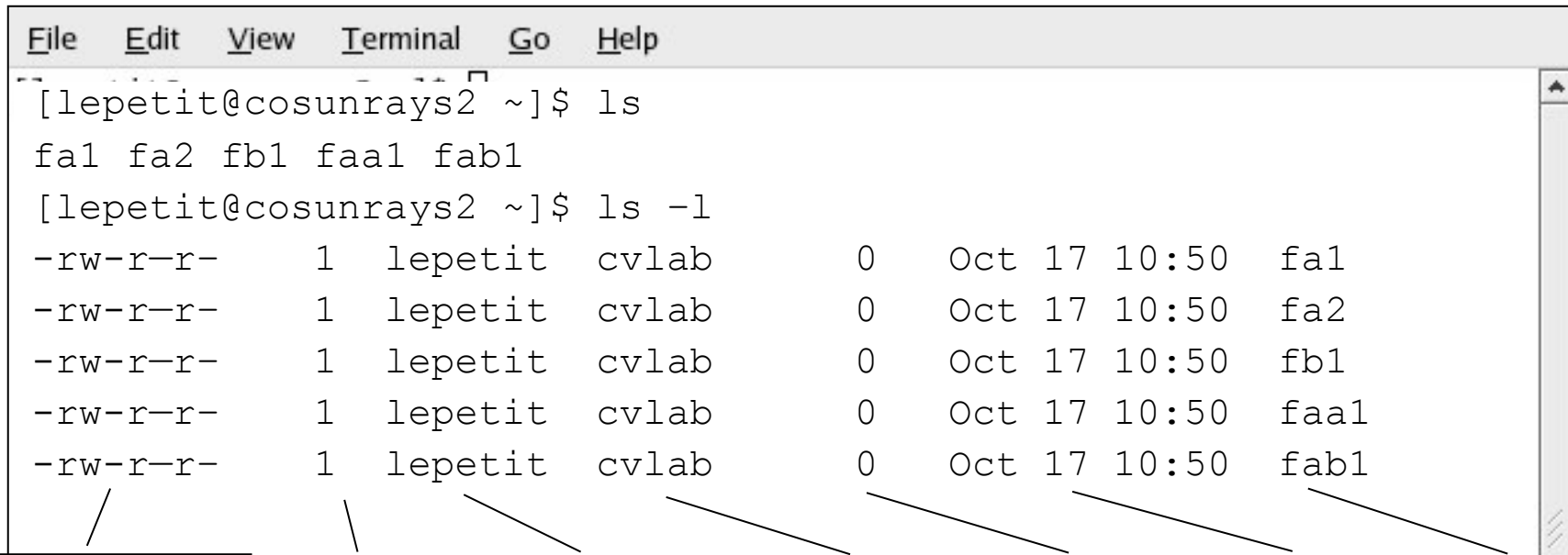
```
File Edit View Terminal Go Help
[lepetit@cosunrays2 ~]$ touch fa1
[lepetit@cosunrays2 ~]$ touch fa2
[lepetit@cosunrays2 ~]$ touch fb1
[lepetit@cosunrays2 ~]$ touch faa1
[lepetit@cosunrays2 ~]$ touch fab1
[lepetit@cosunrays2 ~]$ ls
fa1 fa2 fb1 faa1 fab1
[lepetit@cosunrays2 ~]$
```

# Les commandes ont souvent des options

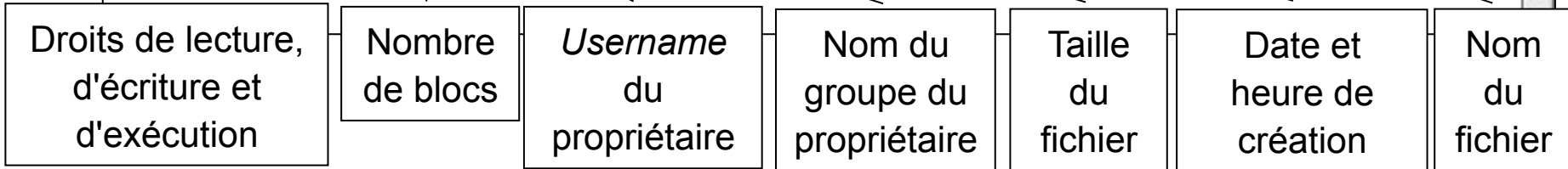
Les options commencent généralement par le caractère -

Par exemple, la commande `ls` a une option notée `-l` qui permet d'obtenir plus d'information sur les fichiers.

Il faut mettre un (ou plusieurs espaces) entre la commande et l'option.



```
File Edit View Terminal Go Help
[lepetit@cosunrays2 ~]$ ls
fa1 fa2 fb1 faa1 fab1
[lepetit@cosunrays2 ~]$ ls -l
-rw-r--r-- 1 lepetit cvlab 0 Oct 17 10:50 fa1
-rw-r--r-- 1 lepetit cvlab 0 Oct 17 10:50 fa2
-rw-r--r-- 1 lepetit cvlab 0 Oct 17 10:50 fb1
-rw-r--r-- 1 lepetit cvlab 0 Oct 17 10:50 faa1
-rw-r--r-- 1 lepetit cvlab 0 Oct 17 10:50 fab1
```



# Les caractères jokers \* et ?

? représente n'importe quel caractère (un et un seul);

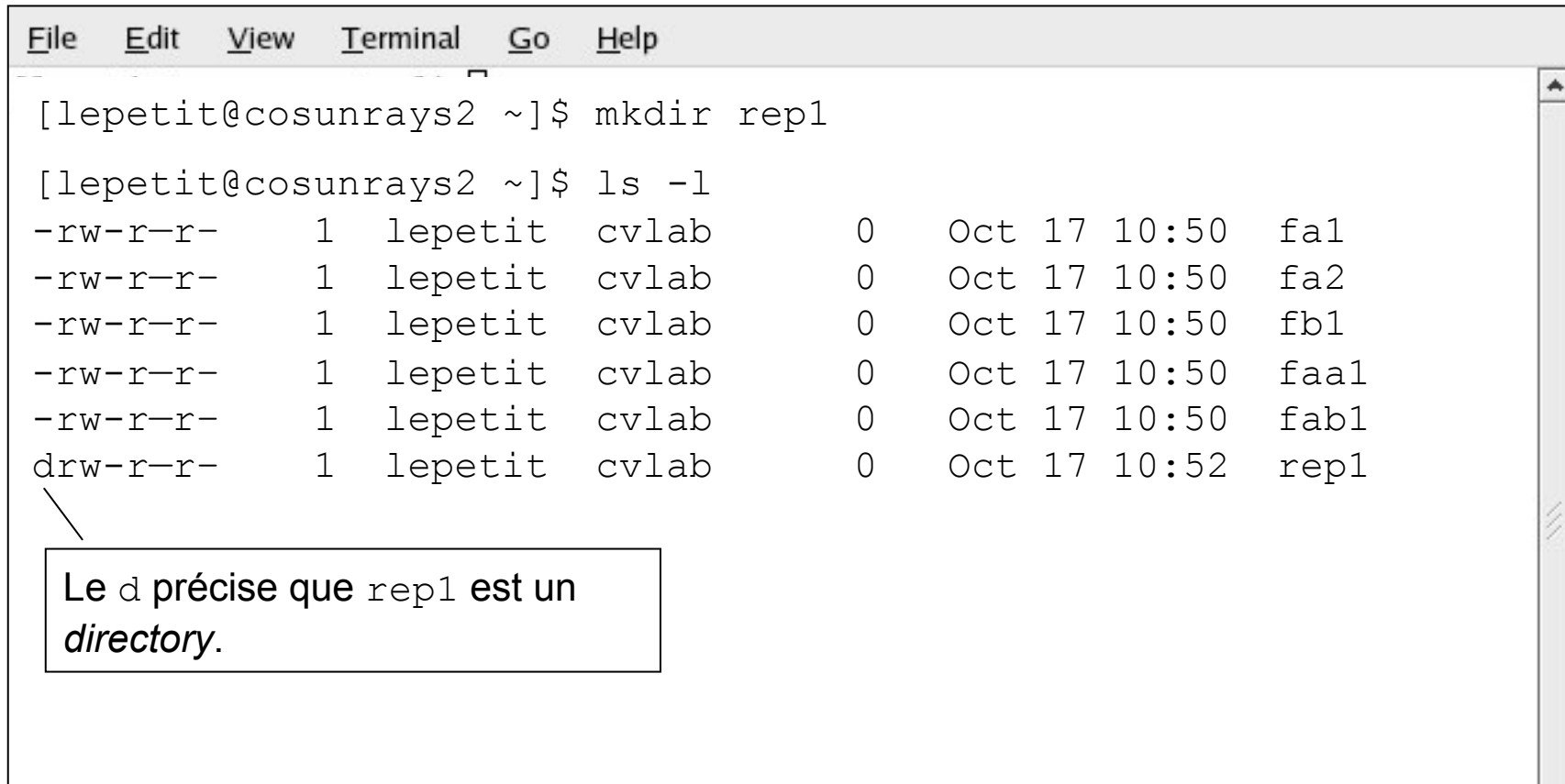
\* représente n'importe quelle séquence de caractères.

```
File Edit View Terminal Go Help
[lepetit@cosunrays2 ~]$ ls
fa1 fa2 fb1 faa1 fab1
[lepetit@cosunrays2 ~]$ ls fa?
fa1 fa2
[lepetit@cosunrays2 ~]$ ls fa?1
faa1 fab1
[lepetit@cosunrays2 ~]$ ls fa*
fa1 fa2 faa1 fab1
[lepetit@cosunrays2 ~]$ ls fa*1
fa1 faa1 fab1
```

# Création d'un répertoire `mkdir`

La commande `mkdir` (pour *make directory*) permet de créer un répertoire. Par exemple:

```
mkdir rep1
```



A terminal window with a menu bar (File, Edit, View, Terminal, Go, Help) and a title bar. The terminal shows the execution of `mkdir rep1` and `ls -l`. The output of `ls -l` shows a list of files and a newly created directory `rep1`. A callout box points to the `d` in the permissions of `rep1`.

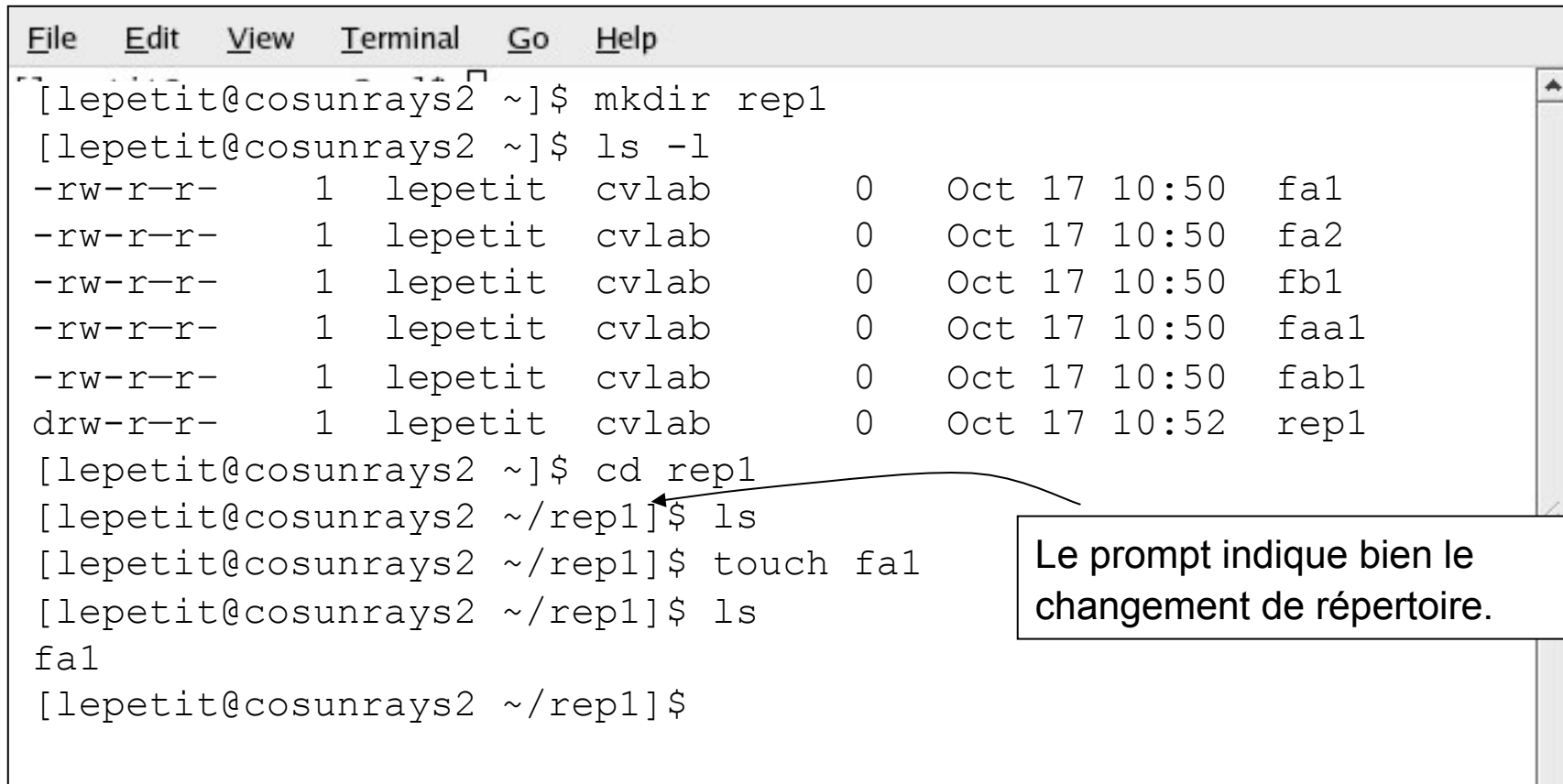
```
[lepetit@cosunrays2 ~]$ mkdir rep1
[lepetit@cosunrays2 ~]$ ls -l
-rw-r--r--  1 lepetit  cvlab    0  Oct 17 10:50  fa1
-rw-r--r--  1 lepetit  cvlab    0  Oct 17 10:50  fa2
-rw-r--r--  1 lepetit  cvlab    0  Oct 17 10:50  fb1
-rw-r--r--  1 lepetit  cvlab    0  Oct 17 10:50  faa1
-rw-r--r--  1 lepetit  cvlab    0  Oct 17 10:50  fab1
drw-r--r--  1 lepetit  cvlab    0  Oct 17 10:52  rep1
```

Le `d` précise que `rep1` est un *directory*.

# Se déplacer dans l'arborescence `cd`

La commande `cd` (pour change directory) permet de modifier le répertoire dans lequel les commandes seront exécutées. Par exemple:

```
cd rep1
```



A terminal window with a menu bar (File, Edit, View, Terminal, Go, Help) and a title bar. The terminal shows the following sequence of commands and output:

```
[lepetit@cosunrays2 ~]$ mkdir rep1
[lepetit@cosunrays2 ~]$ ls -l
-rw-r--r--  1 lepetit  cvlab    0  Oct 17 10:50  fa1
-rw-r--r--  1 lepetit  cvlab    0  Oct 17 10:50  fa2
-rw-r--r--  1 lepetit  cvlab    0  Oct 17 10:50  fb1
-rw-r--r--  1 lepetit  cvlab    0  Oct 17 10:50  faa1
-rw-r--r--  1 lepetit  cvlab    0  Oct 17 10:50  fab1
drw-r--r--  1 lepetit  cvlab    0  Oct 17 10:52  rep1
[lepetit@cosunrays2 ~]$ cd rep1
[lepetit@cosunrays2 ~/rep1]$ ls
[lepetit@cosunrays2 ~/rep1]$ touch fa1
[lepetit@cosunrays2 ~/rep1]$ ls
fa1
[lepetit@cosunrays2 ~/rep1]$
```

An arrow points from a text box to the prompt `~/rep1` in the terminal output.

**Le prompt indique bien le changement de répertoire.**

# Une autre option de `ls`: `-a`

## Les répertoires `.` et `..`

```
ls -a ou
```

```
ls -al
```

permettent de voir les répertoires et les fichiers "cachés", c'est-à-dire ceux dont le nom commencent par le caractère `.` (point)

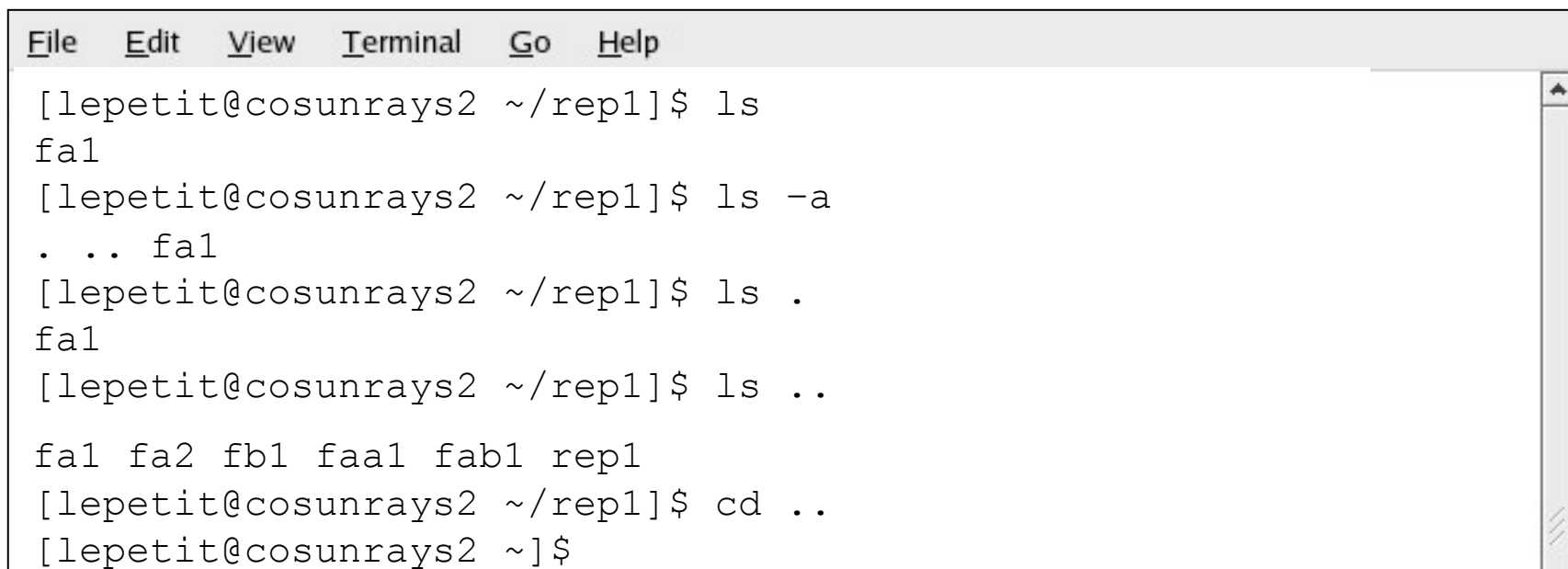
Tous les répertoires contiennent un répertoire `.` et un répertoire `..`

- `.` désigne le répertoire courant

- `..` désigne le repertoire supérieur dans l'arborescence

```
ls . = ls
```

```
cd .. : retour au répertoire supérieur dans l'arborescence
```



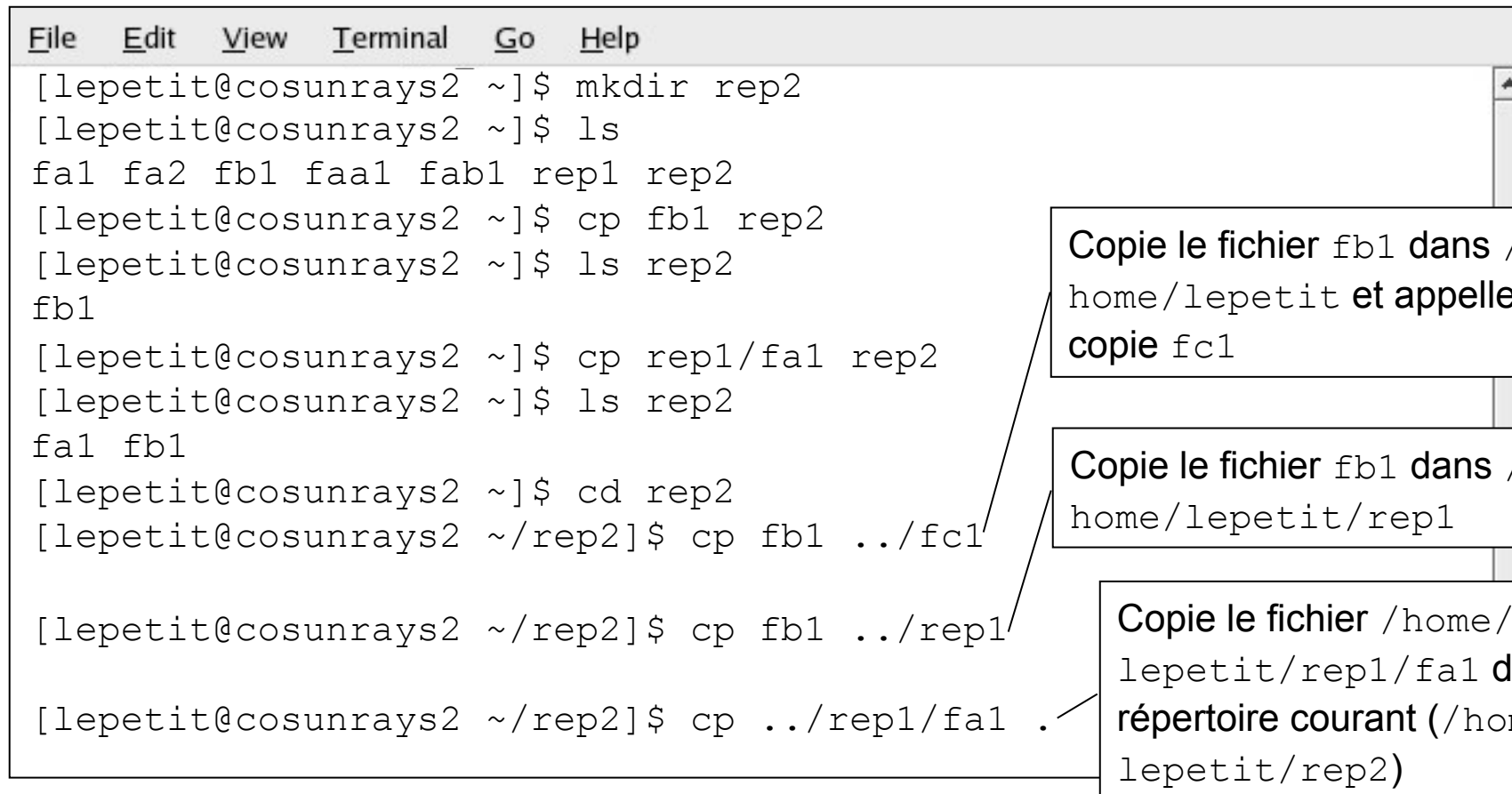
```
File Edit View Terminal Go Help
[lepetit@cosunrays2 ~/rep1]$ ls
fa1
[lepetit@cosunrays2 ~/rep1]$ ls -a
. .. fa1
[lepetit@cosunrays2 ~/rep1]$ ls .
fa1
[lepetit@cosunrays2 ~/rep1]$ ls ..
fa1 fa2 fb1 faa1 fab1 rep1
[lepetit@cosunrays2 ~/rep1]$ cd ..
[lepetit@cosunrays2 ~]$
```

# Copier des fichiers: cp

La commande `cp` (pour *copy*) permet de copier des fichiers.

```
cp fb1 rep2
```

copie le fichier `fb1` dans le répertoire `rep2`



```
File Edit View Terminal Go Help
[lepetit@cosunrays2 ~]$ mkdir rep2
[lepetit@cosunrays2 ~]$ ls
fa1 fa2 fb1 faa1 fab1 rep1 rep2
[lepetit@cosunrays2 ~]$ cp fb1 rep2
[lepetit@cosunrays2 ~]$ ls rep2
fb1
[lepetit@cosunrays2 ~]$ cp rep1/fa1 rep2
[lepetit@cosunrays2 ~]$ ls rep2
fa1 fb1
[lepetit@cosunrays2 ~]$ cd rep2
[lepetit@cosunrays2 ~/rep2]$ cp fb1 ../fc1
[lepetit@cosunrays2 ~/rep2]$ cp fb1 ../rep1
[lepetit@cosunrays2 ~/rep2]$ cp ../rep1/fa1 .
```

**Copie le fichier `fb1` dans /home/lepetit et appelle la copie `fc1`**

**Copie le fichier `fb1` dans /home/lepetit/rep1**

**Copie le fichier `/home/lepetit/rep1/fa1` dans le répertoire courant (`/home/lepetit/rep2`)**

# Déplacer et renommer des fichiers: `mv`

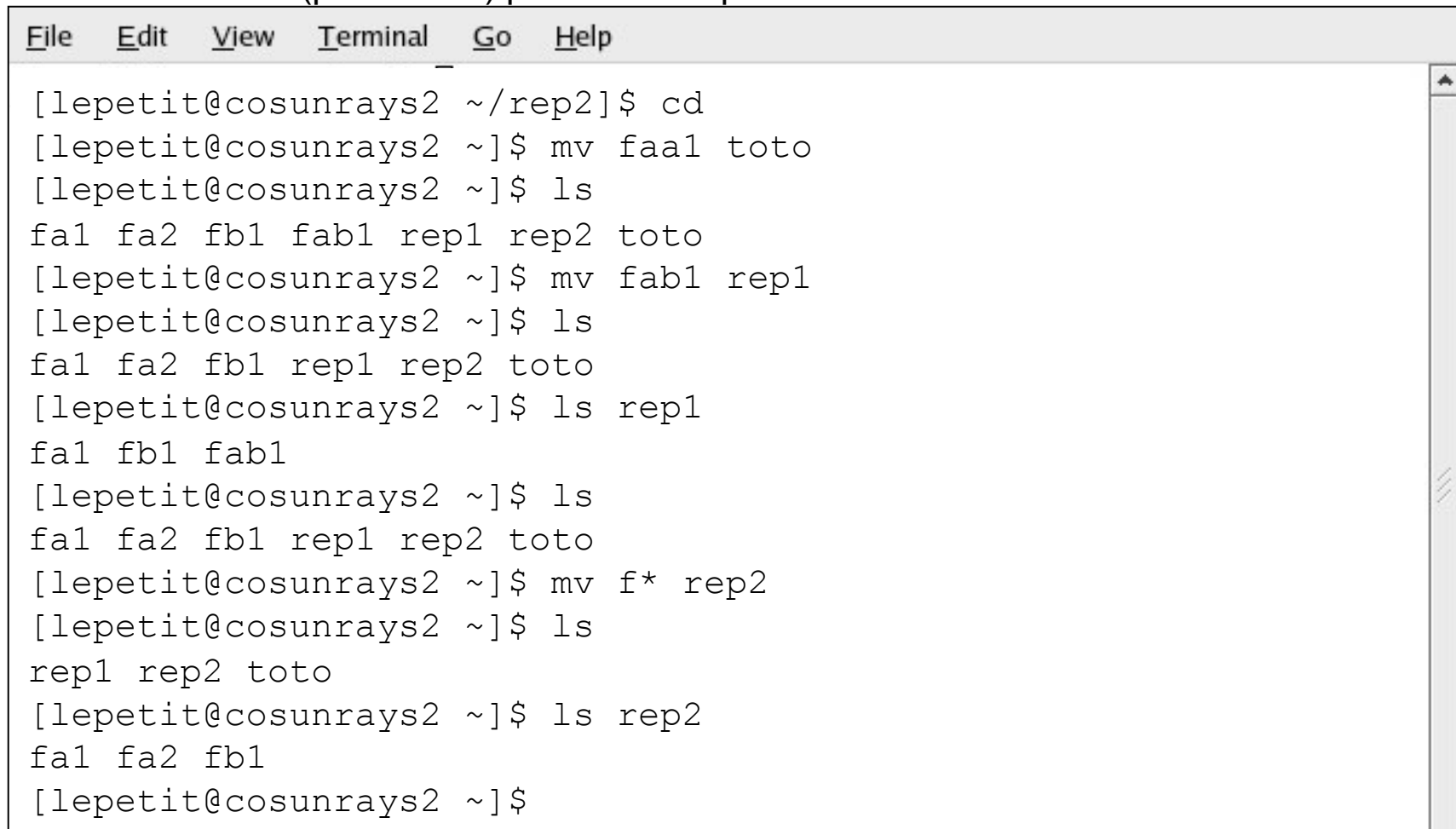
Revenir dans son *home directory*:

`cd ..` si on est dans un répertoire situé dans son *home directory*

ou simplement

`cd` quel que soit l'endroit

La commande `mv` (pour *move*) permet de déplacer ou renommer des fichiers.

A terminal window with a menu bar (File, Edit, View, Terminal, Go, Help) and a scroll bar on the right. The terminal shows a series of commands and their outputs. The user starts in ~/rep2, moves to ~, then moves 'faa1' to 'toto'. They list files, then move 'fab1' to 'rep1', list files again, and list files in 'rep1'. Finally, they move all files 'f\*' to 'rep2' and list files in 'rep2'.

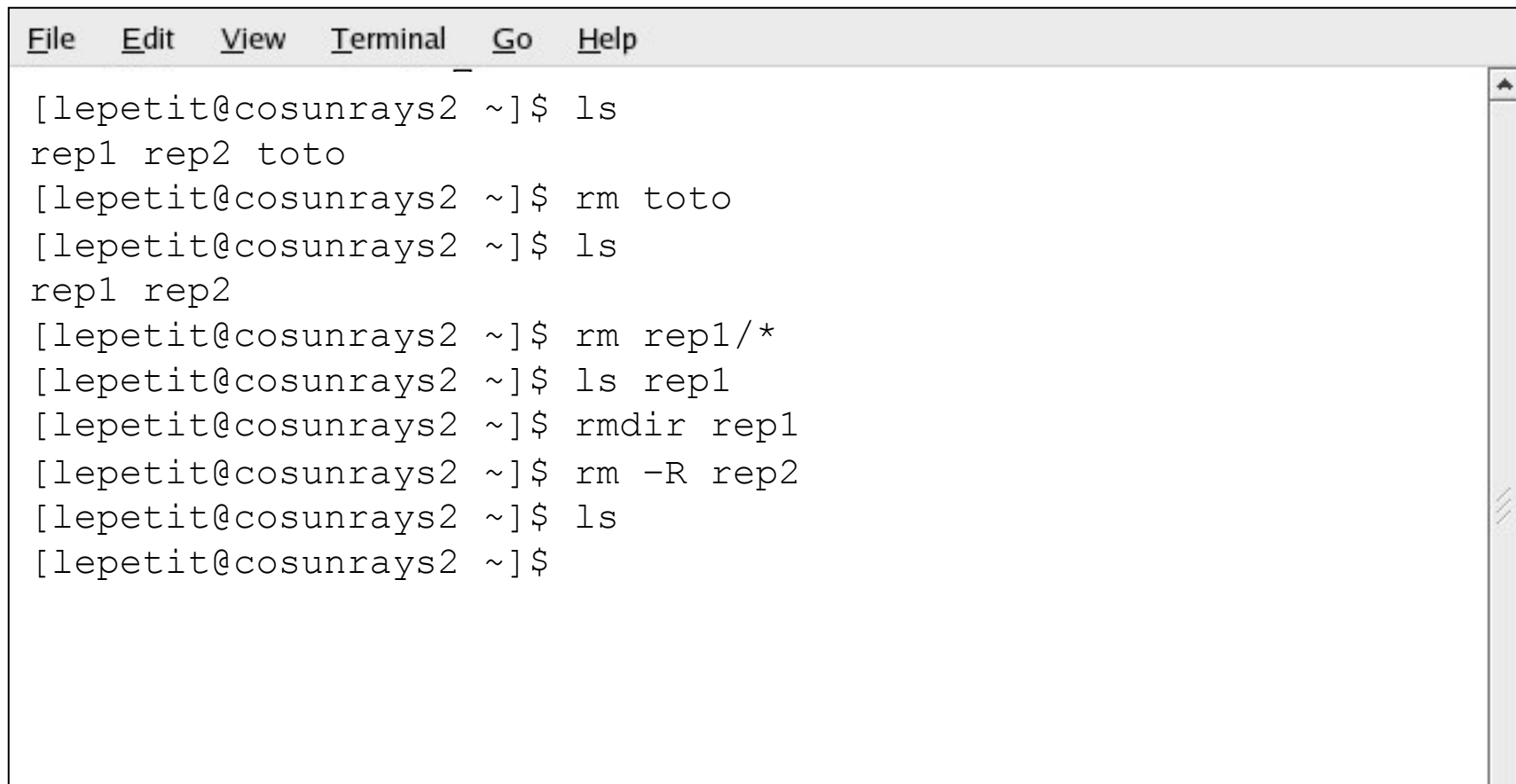
```
[lepetit@cosunrays2 ~/rep2]$ cd
[lepetit@cosunrays2 ~]$ mv faa1 toto
[lepetit@cosunrays2 ~]$ ls
fa1 fa2 fb1 fab1 rep1 rep2 toto
[lepetit@cosunrays2 ~]$ mv fab1 rep1
[lepetit@cosunrays2 ~]$ ls
fa1 fa2 fb1 rep1 rep2 toto
[lepetit@cosunrays2 ~]$ ls rep1
fa1 fb1 fab1
[lepetit@cosunrays2 ~]$ ls
fa1 fa2 fb1 rep1 rep2 toto
[lepetit@cosunrays2 ~]$ mv f* rep2
[lepetit@cosunrays2 ~]$ ls
rep1 rep2 toto
[lepetit@cosunrays2 ~]$ ls rep2
fa1 fa2 fb1
[lepetit@cosunrays2 ~]$
```

# Effacer des fichiers ou un répertoire: `rm` et `rmdir`

La commande `rm` (pour *remove*) permet d'effacer un fichier;

La commande `rmdir` permet d'effacer un répertoire, seulement s'il est vide.

L'option `-R` (pour *recursive*) de `rm` permet d'effacer un répertoire *et* ses fichiers.

A terminal window with a menu bar containing 'File', 'Edit', 'View', 'Terminal', 'Go', and 'Help'. The terminal text shows a series of commands and their outputs: 'ls' showing 'rep1 rep2 toto', 'rm toto', 'ls' showing 'rep1 rep2', 'rm rep1/\*', 'ls rep1', 'rmdir rep1', 'rm -R rep2', and 'ls' showing an empty directory.

```
File  Edit  View  Terminal  Go  Help
[lepetit@cosunrays2 ~]$ ls
rep1 rep2 toto
[lepetit@cosunrays2 ~]$ rm toto
[lepetit@cosunrays2 ~]$ ls
rep1 rep2
[lepetit@cosunrays2 ~]$ rm rep1/*
[lepetit@cosunrays2 ~]$ ls rep1
[lepetit@cosunrays2 ~]$ rmdir rep1
[lepetit@cosunrays2 ~]$ rm -R rep2
[lepetit@cosunrays2 ~]$ ls
[lepetit@cosunrays2 ~]$
```

# Chemins absolus

Jusqu'ici, nous avons essentiellement vu des chemins relatifs:

```
rep1  
../rep2
```

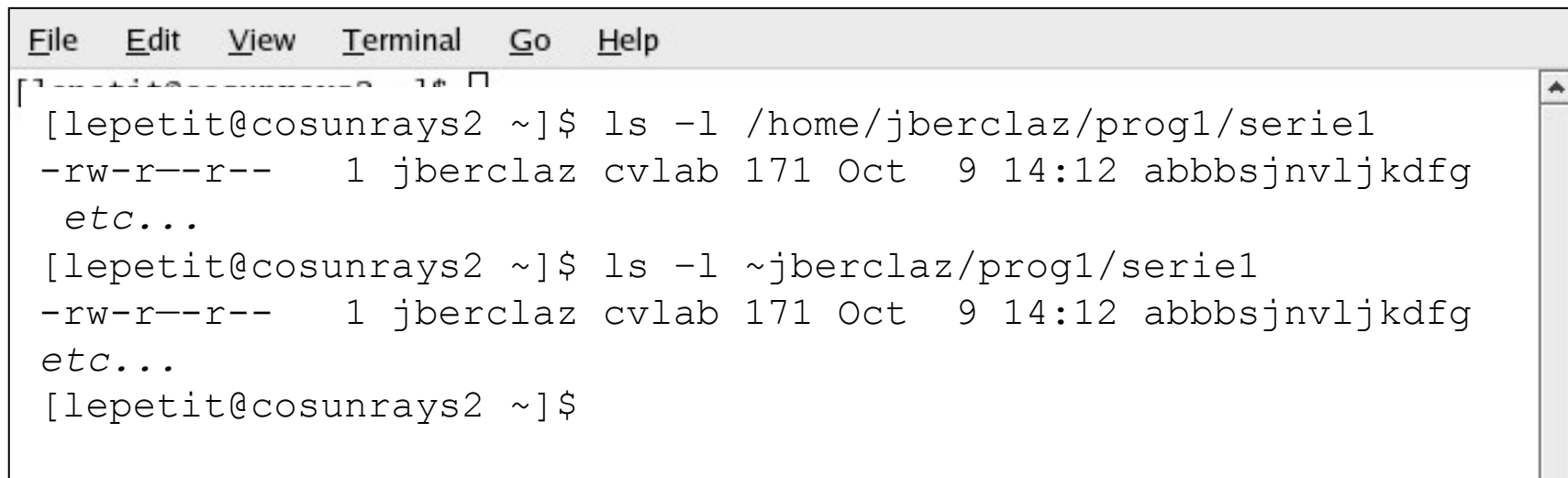
qui définissent un répertoire par rapport au répertoire courant.

On peut également définir un répertoire par son chemin absolu, en partant du répertoire racine /, par exemple:

```
/home/jberclaz/prog1/serie1
```

**Remarque:** `~jberclaz` est un synonyme de `/home/jberclaz`

**Plus généralement,** `~username` est un synonyme de `/home/username`

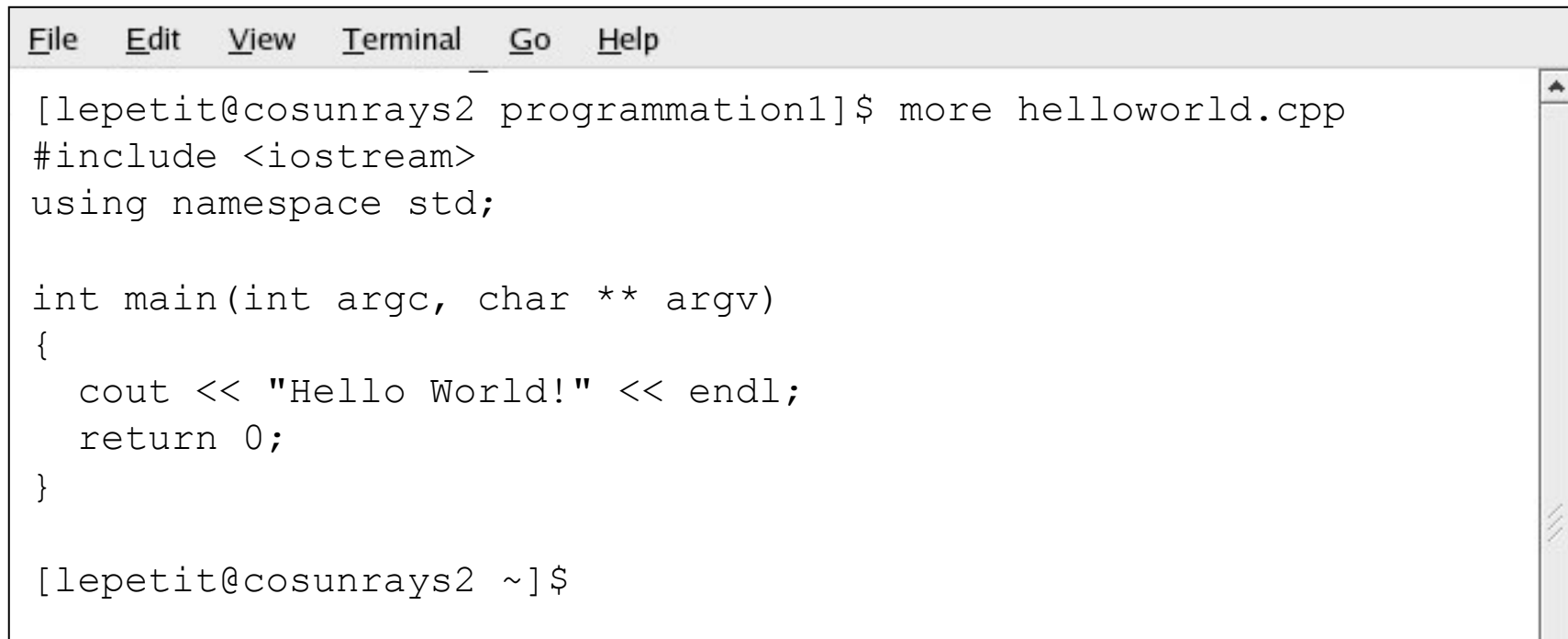
A terminal window with a menu bar (File, Edit, View, Terminal, Go, Help) and a scroll bar on the right. The terminal shows two commands and their outputs. The first command is `ls -l /home/jberclaz/prog1/serie1` and the second is `ls -l ~jberclaz/prog1/serie1`. Both show the same file listing: `-rw-r--r-- 1 jberclaz cvlab 171 Oct 9 14:12 abbbsjnvljkdffg etc...`.

```
File Edit View Terminal Go Help  
[lepetit@cosunrays2 ~]$ ls -l /home/jberclaz/prog1/serie1  
-rw-r--r-- 1 jberclaz cvlab 171 Oct 9 14:12 abbbsjnvljkdffg  
etc...  
[lepetit@cosunrays2 ~]$ ls -l ~jberclaz/prog1/serie1  
-rw-r--r-- 1 jberclaz cvlab 171 Oct 9 14:12 abbbsjnvljkdffg  
etc...  
[lepetit@cosunrays2 ~]$
```

# Afficher le contenu d'un fichier (more)

Exemple:

```
more helloworld.cpp
```

A terminal window with a menu bar containing 'File', 'Edit', 'View', 'Terminal', 'Go', and 'Help'. The terminal text shows a user running 'more helloworld.cpp' in a directory 'programmation1'. The output displays the content of the file, which is a C++ program that prints 'Hello World!'. The terminal prompt returns to the user's home directory.

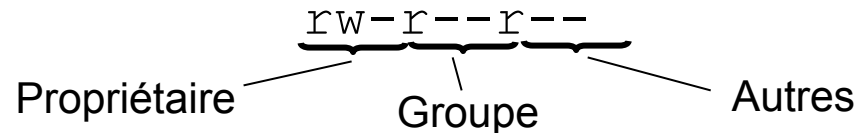
```
File  Edit  View  Terminal  Go  Help
[lepetit@cosunrays2 programmation1]$ more helloworld.cpp
#include <iostream>
using namespace std;

int main(int argc, char ** argv)
{
    cout << "Hello World!" << endl;
    return 0;
}

[lepetit@cosunrays2 ~]$
```

# Droits de lecture, d'écriture et d'exécution

Le `rw-r--r--` qui apparaît quand on utilise l'option `-l` de `ls` précise les droits d'accès aux fichiers.



Les 3 premières lettres (ici `rw-`) définissent les droits d'accès du propriétaire du fichier:  
La première lettre vaut `r` ou `-` : Le propriétaire a le droit de **lire** (`r`) ou non (`-`) le fichier;  
La deuxième lettre vaut `w` ou `-` : Le propriétaire a le droit d'**écrire** (`w`) ou non (`-`) le fichier;  
La troisième lettre vaut `x` ou `-` : Le propriétaire a le droit d'**exécuter** (`x`) ou non (`-`) le fichier.

Les 3 lettres suivantes définissent les droits pour les membres du même groupe que le propriétaire.

Les 3 dernières lettres définissent les droits pour tous les autres utilisateurs.

```
File Edit View Terminal Go Help
[lepetit@cosunrays2 ~]$ ls -l /home/jberclaz/prog1/serie1
-rw-r--r--  1 jberclaz cvlab 171 Oct  9 14:12 abbbsjnvlijkdfg
etc...
[lepetit@cosunrays2 ~]$ rm ~jberclaz/prog1/serie1/*
rm: cannot remove ...: Permission denied
[lepetit@cosunrays2 ~]$
```

# Changer les droits (chmod)

```
chmod g+w fa1
```

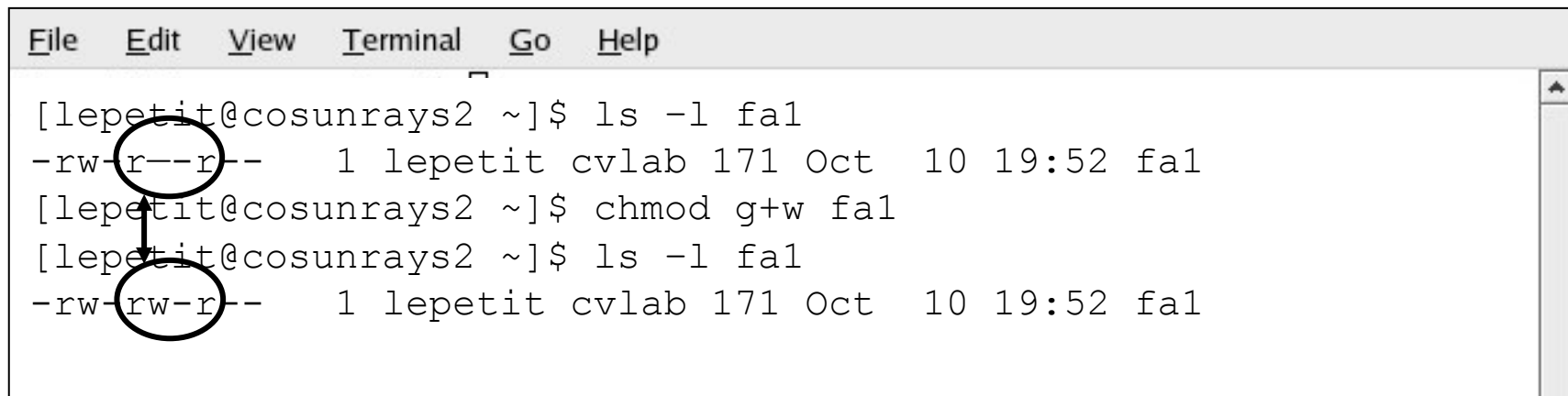
donne (+) les droits d'écriture (w) aux membres du groupe (g)

```
chmod o-r fa1
```

enlève les droits de lecture aux utilisateurs qui n'appartiennent pas au même groupe.

```
chmod og-r fa1
```

enlève les droits de lecture à tout utilisateur à part le propriétaire.



```
File Edit View Terminal Go Help
[lepetit@cosunrays2 ~]$ ls -l fa1
-rw-r--r--  1 lepetit cvlab 171 Oct  10 19:52 fa1
[lepetit@cosunrays2 ~]$ chmod g+w fa1
[lepetit@cosunrays2 ~]$ ls -l fa1
-rw-rw-r--  1 lepetit cvlab 171 Oct  10 19:52 fa1
```

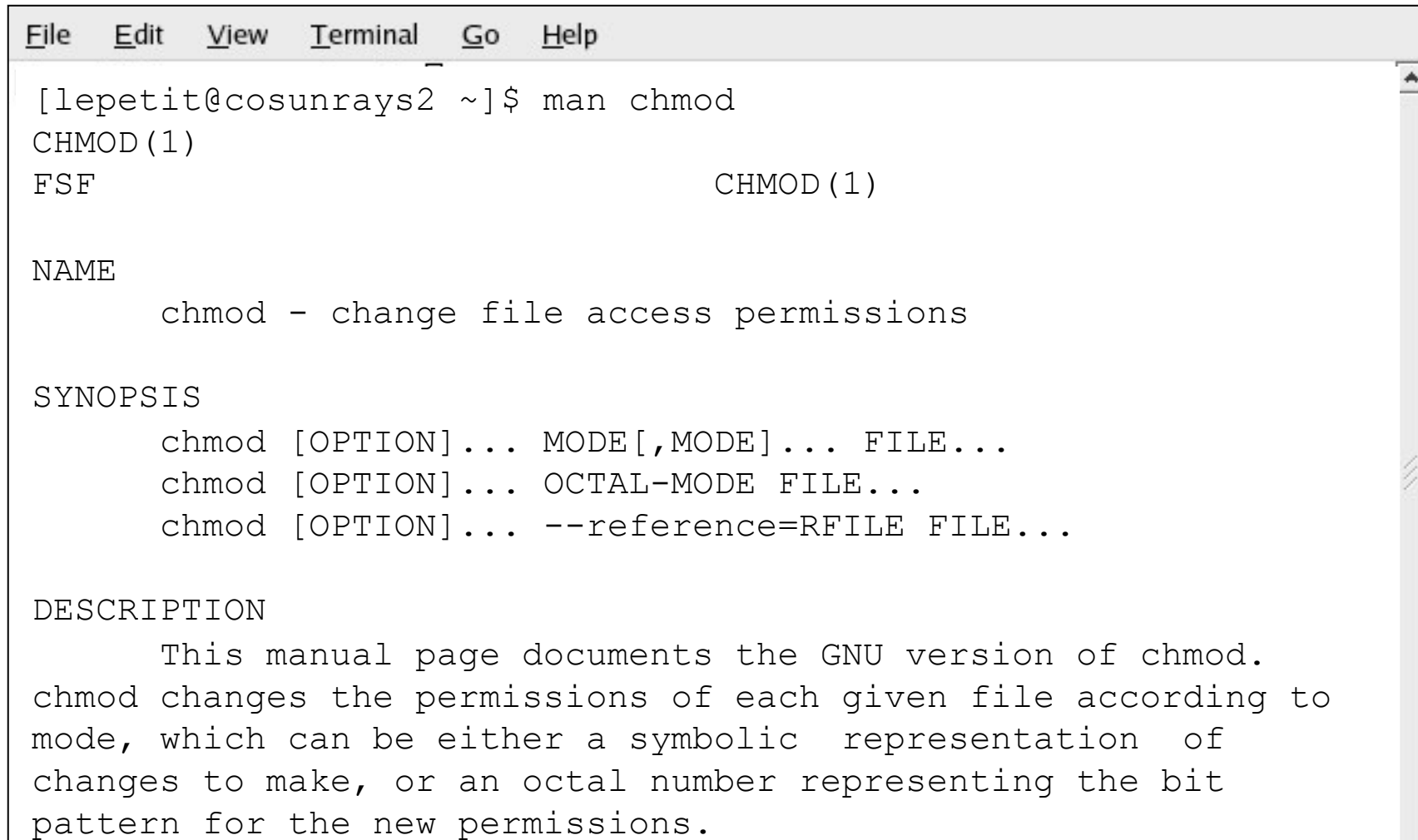
The terminal window shows the command `chmod g+w fa1` being executed. The permissions for `fa1` change from `-rw-r--r--` to `-rw-rw-r--`. Two circles are drawn around the permission strings in the original image to highlight the change.

# Manuel en ligne (man)

man chmod

décrit la commande chmod.

**Presser 'q' pour sortir.**



```
File Edit View Terminal Go Help
[lepetit@cosunrays2 ~]$ man chmod
CHMOD(1)
FSF                                CHMOD(1)

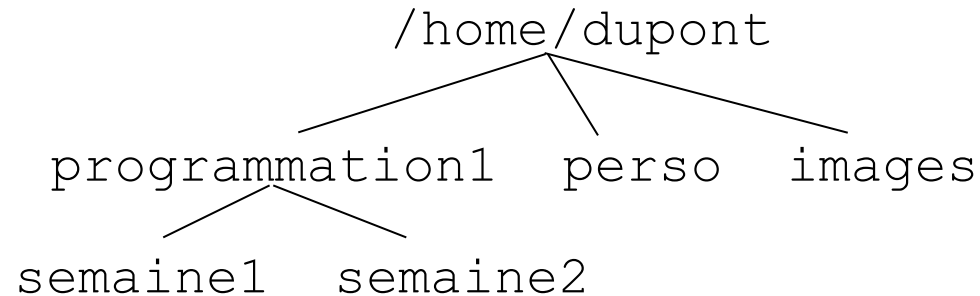
NAME
    chmod - change file access permissions

SYNOPSIS
    chmod [OPTION]... MODE[,MODE]... FILE...
    chmod [OPTION]... OCTAL-MODE FILE...
    chmod [OPTION]... --reference=RFILE FILE...

DESCRIPTION
    This manual page documents the GNU version of chmod.
    chmod changes the permissions of each given file according to
    mode, which can be either a symbolic representation of
    changes to make, or an octal number representing the bit
    pattern for the new permissions.
```

# Exercices

Supposons que votre username est *dupont*, et que votre *home directory* contienne l'architecture suivante:



- Le répertoire courant est votre *home directory*. Comment interdire l'accès à votre répertoire *perso* aux autres utilisateurs ?

```
chmod og-rx perso
```

- Vous avez créé par mégarde les fichiers *hello.cpp*, *hello.o* et *hello* dans votre *home directory* alors que leur place est plutôt dans *programmation1/semaine1*. Comment les déplacer tous au bon endroit *en une commande* ?

```
mv hello* programmation1/semaine1
```

- Comment déplacer tous les fichiers *images* se terminant par l'extension *.jpg* de *perso* vers *images* en une commande ?

```
mv perso/*.jpg images
```

- Comment changer le répertoire courant en *programmation1/semaine1* en une commande ?

```
cd programmation1/semaine1
```

# Le langage C

# Le langage C

Créé en 1972;

A l'époque: langage de haut niveau, très portable.

Reste un langage très rapide, et l'un des langages les plus utilisés actuellement.

Le C++ est un langage orienté objet qui a été créé à partir du langage C en 1983.

# Pourquoi le langage C ?

De nombreux langages de programmation existent, pourquoi apprendre le langage C?

- il est très général;
- il a inspiré de nombreux autres langages, basés sur les mêmes principes:
  - C++ (langage objet souvent utilisé pour la programmation de logiciels),
  - Java (programmes portables utilisables entre autres sur le web),
  - C#
  - Pascal,
  - Matlab, Maple (programmes mathématiques),
  - PHP (programmation de pages html dynamiques)
  - etc...
- la syntaxe de ces langages est souvent proche de celle du langage C.

→ le langage C est un bon acquis pour l'apprentissage d'autres langages.

*Nous apprendrons un compromis entre le C et le C++:*

*le C++ sans la partie programmation orientée objet.*

*(la programmation orientée objet sera vue lors du cours du deuxième semestre)*

# N'achetez pas de livres !

Les ouvrages sur le langage C sont assez décevants.

Le cours au format pdf sera mis d'une semaine à l'autre sur le site du cours (voir plus loin pour l'adresse).

Vous aurez de nombreux sujets d'exercices avec leur corrigé.

Contrairement aux autres matières, on peut facilement créer soi-même les énoncés ("*et si je changeais mon programme pour qu'il fasse ça ?*").

On peut également facilement trouver d'autres sujets d'exercices sur le web.

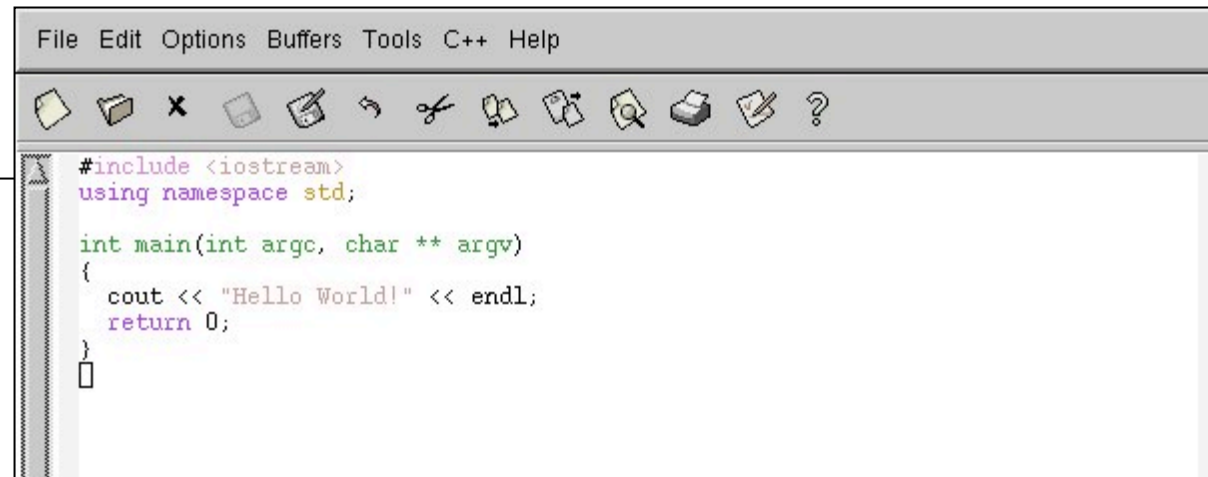
# Premier programme en langage C/C++: Hello world

Affichage d'un message à l'écran.

Un programme en langage C est un fichier texte:

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

int main(int argc, char ** argv)
{
    cout << "Hello World!" << endl;
    return 0;
}
```

A screenshot of a C++ IDE window. The window has a menu bar with 'File', 'Edit', 'Options', 'Buffers', 'Tools', 'C++', and 'Help'. Below the menu bar is a toolbar with various icons for file operations and editing. The main text area contains the same C++ code as shown in the previous block, with syntax highlighting: keywords are in green, strings are in red, and other identifiers are in black. The code is: 

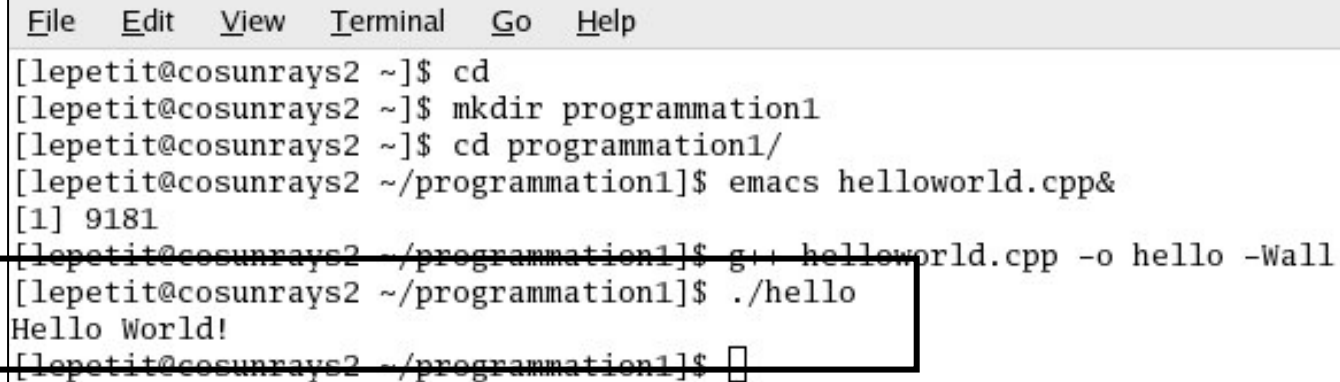
```
#include <iostream>
using namespace std;

int main(int argc, char ** argv)
{
    cout << "Hello World!" << endl;
    return 0;
}
```

# Ce que fait ce programme:

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

int main(int argc, char ** argv)
{
    cout << "Hello World!" << endl;
    return 0;
}
```

A terminal window with a menu bar (File, Edit, View, Terminal, Go, Help) and a scroll bar on the right. The terminal shows the following commands and output:

```
[lepetit@cosunrays2 ~]$ cd
[lepetit@cosunrays2 ~]$ mkdir programmation1
[lepetit@cosunrays2 ~]$ cd programmation1/
[lepetit@cosunrays2 ~/programmation1]$ emacs helloworld.cpp&
[1] 9181
[lepetit@cosunrays2 ~/programmation1]$ g++ helloworld.cpp -o hello -Wall
[lepetit@cosunrays2 ~/programmation1]$ ./hello
Hello World!
[lepetit@cosunrays2 ~/programmation1]$
```

# Structure d'un programme en langage C/C++

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

int main(int argc, char ** argv)
{
    cout << "Hello World!" << endl;
    return 0;
}
```

En-tête de la fonction main

Bloc:

- délimité par des accolades { }
- contient le corps de la fonction.

Une instruction: les instructions se terminent par un point-virgule ;

# Instructions

Le bloc suivant l'en-tête de la fonction `main` est composé d'**instructions**.

Une instruction en C/C++ peut être:

- une instruction simple, terminée par un point-virgule: ';'. Par exemple:  

```
cout << "Hello World!" << endl;
```
- une instruction de contrôle (condition `if`, boucle `for`...)

**Les instructions sont exécutées les unes après les autres:** on dit que le C est un langage *itératif*.

*Sauf si des instructions de contrôle (for, if, while, ...) sont utilisées, comme nous le verrons dans la suite du cours.*

# Pour écrire à l'écran: cout

L'instruction:

```
cout << "Hello World!" << endl;
```

est un exemple d'affichage à l'écran à partir d'un programme C.

`endl` correspond à "retour à la ligne": ce qui sera affiché après le sera au début de la ligne suivante.

Autre exemple:

```
cout << "Hello World!" << " bonjour" << endl << "Good bye";
```

affichera:

```
Hello World! bonjour
```

```
Good bye
```

Les deux premières lignes du programme:

```
#include <iostream>
```

```
using namespace std;
```

permet d'inclure un «fichier en-tête» (`iostream`, pour *input/output stream*), qui contient la déclaration de `cout`.

En pratique, cette ligne permet d'utiliser `cout` dans un programme C++.

# Déroulement du programme

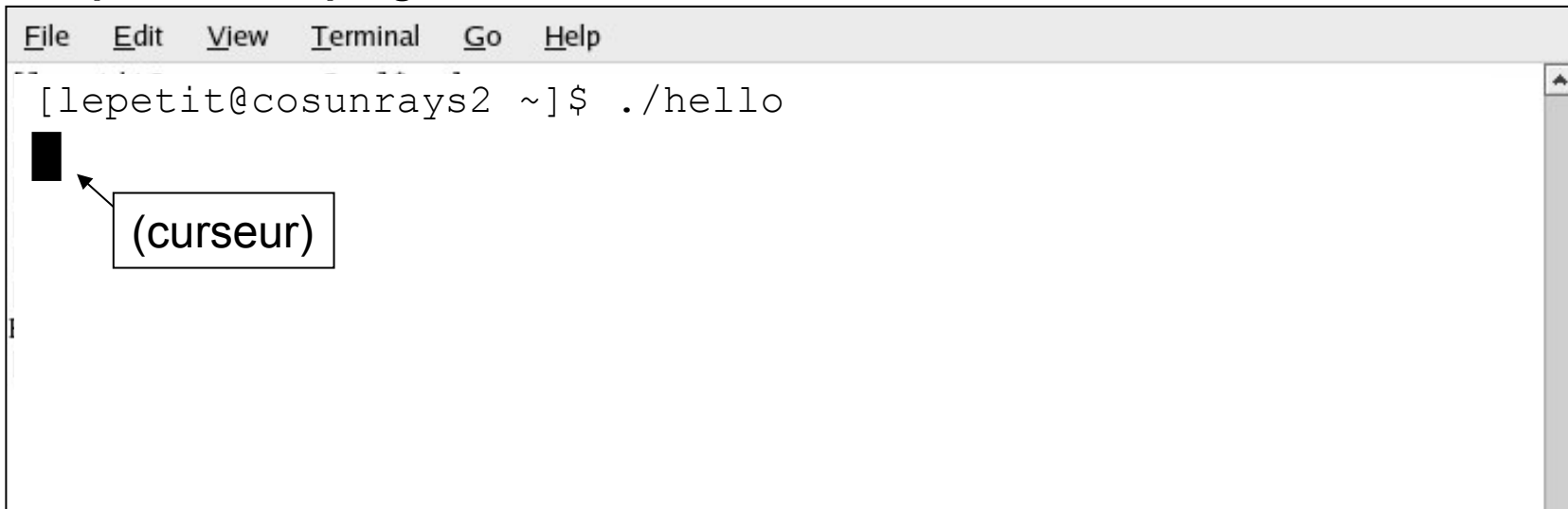
## *Hello world pas-à-pas*

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

int main(int argc, char ** argv)
{
    cout << "Hello World!" << endl;
    cout << "Bonjour" << endl;
    return 0;
}
```



**Ce qu'affiche le programme:**



```
File Edit View Terminal Go Help
[lepetit@cosunrays2 ~]$ ./hello
█
(curseur)
```

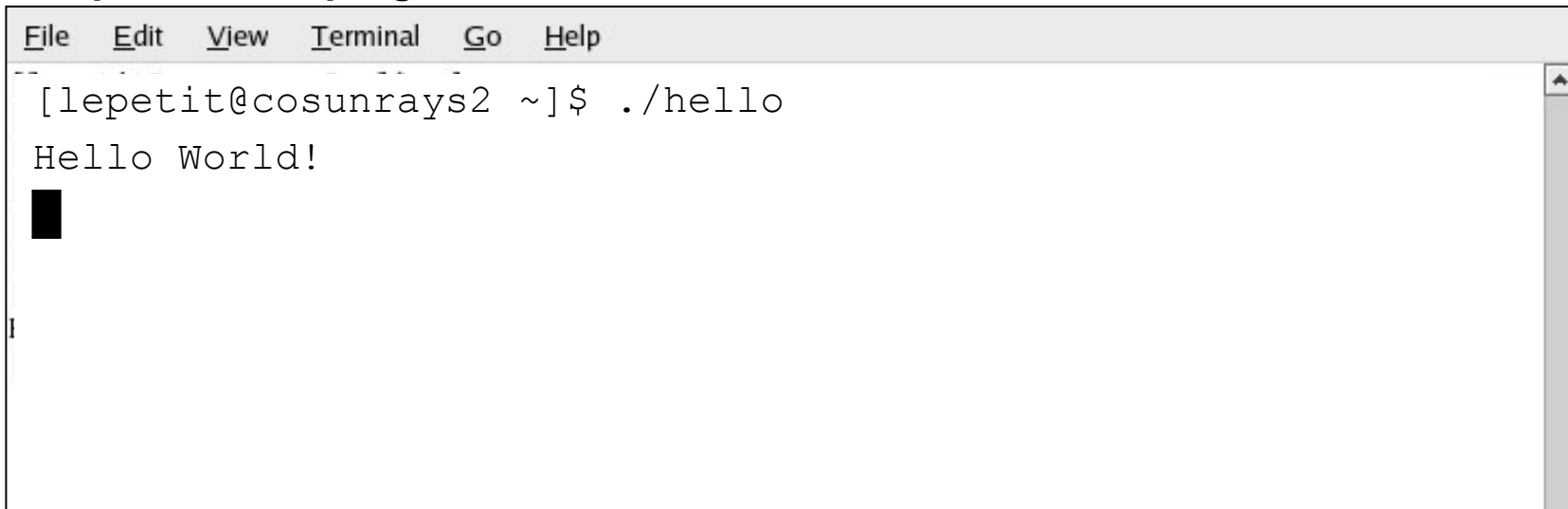
# Déroulement du programme

## *Hello world pas-à-pas*

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

int main(int argc, char ** argv)
{
    cout << "Hello World!" << endl;
    cout << "Bonjour" << endl;
    return 0;
}
```

**Ce qu'affiche le programme:**



A terminal window with a menu bar containing 'File', 'Edit', 'View', 'Terminal', 'Go', and 'Help'. The terminal text shows the command `./hello` being executed, resulting in the output `Hello World!` followed by a cursor.

```
File Edit View Terminal Go Help
[lepetit@cosunrays2 ~]$ ./hello
Hello World!
█
```

# Déroulement du programme

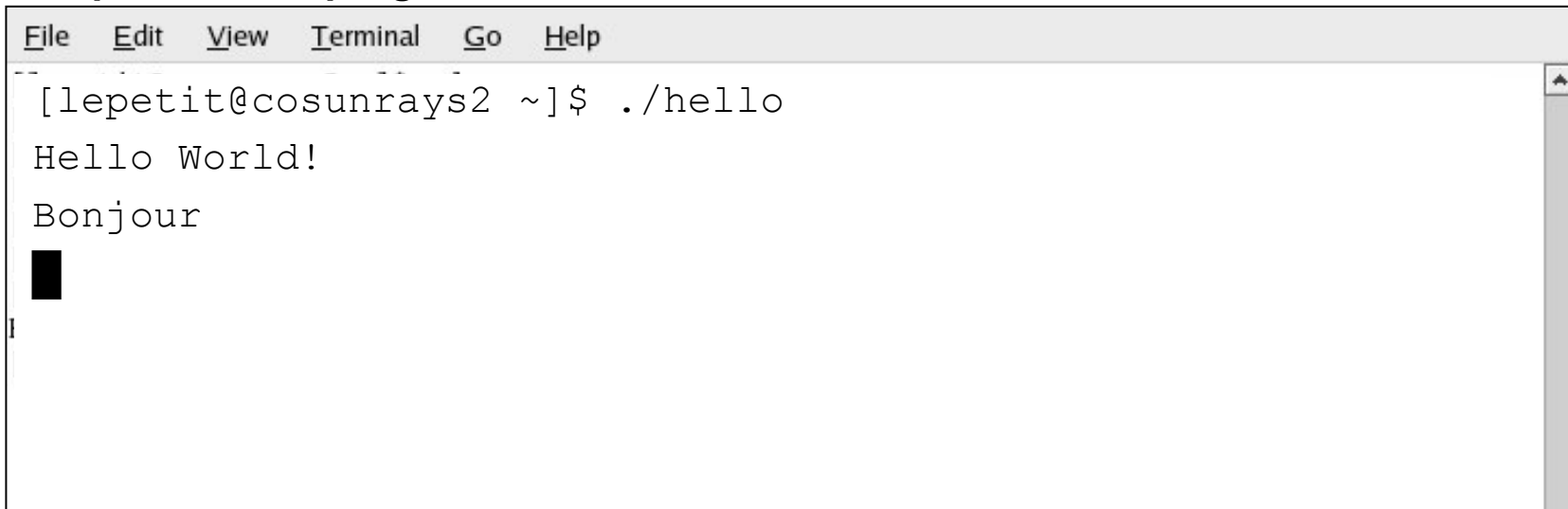
## *Hello world pas-à-pas*

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

int main(int argc, char ** argv)
{
    cout << "Hello World!" << endl;
    cout << "Bonjour" << endl;
    return 0;
}
```



**Ce qu'affiche le programme:**



```
File Edit View Terminal Go Help
[lepetit@cosunrays2 ~]$ ./hello
Hello World!
Bonjour
█
```

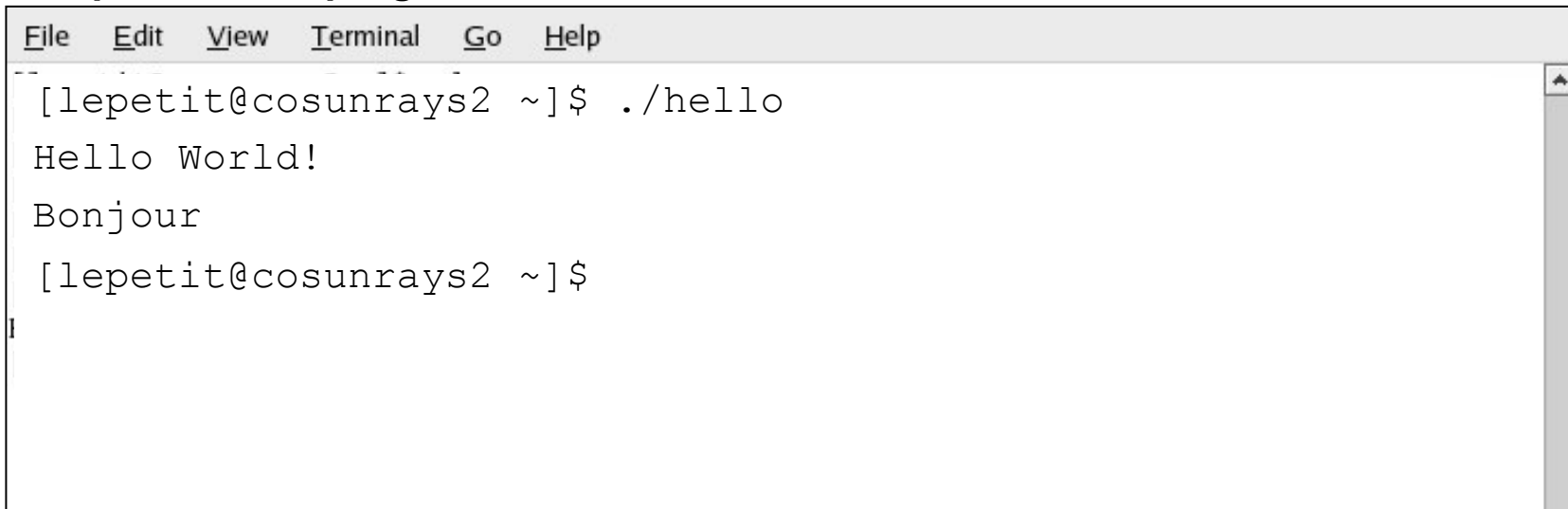
# Déroulement du programme

## *Hello world pas-à-pas*

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

int main(int argc, char ** argv)
{
    cout << "Hello World!" << endl;
    cout << "Bonjour" << endl;
    return 0;
}
```

### Ce qu'affiche le programme:



```
File Edit View Terminal Go Help
[lepetit@cosunrays2 ~]$ ./hello
Hello World!
Bonjour
[lepetit@cosunrays2 ~]$
```

# La compilation

Un programme en langage C est un **fichier texte**, que l'on écrit à l'aide d'un **éditeur de texte**.

Ce programme en langage C n'est pas exécutable directement par la machine: il doit être *compilé* pour pouvoir être exécuté par l'ordinateur.

La compilation est réalisée par un programme appelé **compilateur**. Le compilateur crée un **fichier exécutable**.

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

int main(int argc, char ** argv)
{
    cout << "Hello World!" << endl;
    cout << "Bonjour" << endl;
    return 0;
}
```

Programme en langage C:  
*Fichier texte compréhensible  
par un programmeur*

Compilateur

Programme  
exécutable par  
l'ordinateur

*Fichier compréhensible  
par l'ordinateur*

# Erreurs de programmation

Deux types d'erreurs peuvent survenir quand on écrit un programme:

## 1. Les *erreurs de syntaxe*, qui surviennent à la compilation du programme:

Un programme doit respecter précisément la syntaxe du langage C pour être accepté par le compilateur.

En cas d'erreur de syntaxe, le compilateur signale l'erreur (ou plusieurs erreurs). Par exemple, si on oublie le point-virgule à la fin du `cout`:

```
...  
6  cout << "Hello World!" << endl  
7  return 0;  
}
```

le compilateur affichera le message d'erreur:

```
helloworld.cpp: In function `main':  
helloworld.cpp:7: error: parse error before 'return' token
```

Dans ce cas, le compilateur s'arrête *sans créer d'exécutable*

## 2. Les erreurs qui surviennent lors de l'exécution du programme (*bugs*):

Le programme ne fait pas ce qui est attendu, le programme "plante",...

# Savoir trouver ses erreurs

***Savoir résoudre les erreurs fait partie de l'apprentissage de la programmation:***

***→ les assistants n'apporteront leur aide qu'après recherche personnelle.***

Dans le cas des erreurs de syntaxe:

- **Toujours commencer par corriger la première erreur:** les erreurs suivantes en découlent peut-être.
- Il faut savoir exploiter le message donné par le compilateur pour trouver l'erreur:
  - le compilateur indique le numéro de ligne ou il *estime* que l'erreur s'est produite.
    - Attention au piège: le numéro de ligne n'est qu'indicatif !**
  - le compilateur décrit l'erreur qu'il a trouvé.

# Le message d'erreur du compilateur

1. *Le compilateur indique le numéro de la ligne de code où il a trouvé une erreur:*

```
helloworld.cpp: In function `main':  
helloworld.cpp:7: error: parse error before 'return' token
```

**Attention**, la "vraie" erreur peut être située à la ligne *précédant* celle donnée par le compilateur:

```
6 cout << "Hello World!" << endl  
7 return 0;
```

2. *Il faut savoir exploiter la description de l'erreur qu'en fait le compilateur:*

```
helloworld.cpp:7: error: parse error before "return"
```

Le programme comporte une erreur avant le `return` de la ligne 7 (*parse error* = erreur de syntaxe). Ici, le compilateur trouve l'instruction `return` alors qu'il pensait trouver un point-virgule.

# Exemples de messages d'erreur

Exemples de messages d'erreur:

```
• int main(int argc, char ** argv)
  {
    x = 10;
  }
```

provoque l'erreur:

```
helloworld.cpp:5: error: `x' undeclared (first use in this function)
```

```
helloworld.cpp:5: error: (Each undeclared identifier is reported only
once
```

```
helloworld.cpp:5: error: for each function it appears in.)
```

→ la variable `x` est utilisée sans avoir été déclarée.

```
• #include <iostrime>
```

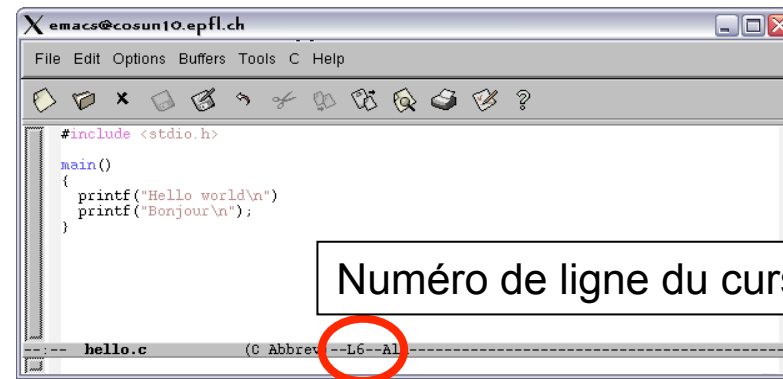
provoque l'erreur:

```
helloworld.cpp:1:21: iostrime : No such file or directory
```

```
• #includ <iostream>
```

provoque l'erreur:

```
helloworld.cpp:1:2: invalid preprocessing directive #includ
```



# Exemples de messages d'erreur (2)

La ligne

```
cout << "hello" << end;
```



provoque, sur certains compilateurs, BEAUCOUP d'erreurs:

```
hello.cpp(10) : error C2065: 'end' : undeclared identifier
```

```
hello.cpp(10) : error C2593: 'operator <<' is ambiguous
```

```
    /usr/include/ostream(434): could be 'std::basic_ostream<_Elem,_Traits>::_Myt
&std::basic_ostream<_Elem,_Traits>::operator <<(std::basic_ostream<_Elem,_Traits>::_Mysb *)'
    with
    [
        _Elem=char,
        _Traits=std::char_traits<char>
    ]
```

```
    /usr/include/ostream(434): could be 'std::basic_ostream<_Elem,_Traits>::_Myt
&std::basic_ostream<_Elem,_Traits>::operator <<(std::basic_ostream<_Elem,_Traits>::_Mysb *)'
    with
    [
        _Elem=char,
        _Traits=std::char_traits<char>
    ]
```

etc...

**Pas de panique ! Il faut juste changer le `end` en `endl` et toutes les erreurs disparaissent...**

# *warning* (Avertissement)

Le compilateur peut également afficher des messages d'avertissement (*warning*) quand il pense que le programme fait quelque chose de bizarre.

Ces messages ne sont pas provoqués par des erreurs de syntaxe, et le compilateur *crée* l'exécutable.

Par exemple:

```
x = x / 0;
```

est syntaxiquement valide mais provoque le *warning*:

```
helloworld.cpp:7: warning: division by zero
```

En général, quand le compilateur affiche un *warning*, le programmeur a effectivement commis une erreur.

On veillera à ce que la compilation s'effectue sans l'affichage de *warning*.

# Erreurs de syntaxe

Trouvez les erreurs de syntaxe de ce programme:

```
include <iostream>;  
using namespade std;  
  
int main(int argc, char ** argv)  
{  
    cout "Hello world!!! << endl;  
  
    return          0;  
}
```

# Erreurs de syntaxe

Manque le #

Pas de ; à la fin de #include

Il manque le <<

namespade  
au lieu de  
namespace

Il manque le "

```
include <iostream>  
using namespace std;  
int main(int arge, char ** argv)  
{  
    cout "Hello world!!! << endl;  
  
    return 0;  
}
```

Pas de problème, on peut mettre autant d'espaces que l'on veut (au moins un), même si c'est peu lisible.

# Erreurs de syntaxe

## Programme corrigé:

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

int main(int argc, char ** argv)
{
    cout << "Hello world!!!" << endl;

    return 0;
}
```

# Organisation des séances d'exercices

2 heures d'exercices sur machine:

- terminaux sous Linux (Ubuntu), salles CO020 à CO023;
- **UNE** personne par ordinateur.
- 1 assistant, 7 assistants-étudiants.

***Si vous n'avez pas fini les exercices d'une séance***

***→ vous pouvez finir en dehors des heures de cours avant la séance suivante: les salles sont ouvertes de 7h à 21h du lundi au samedi.***