Refresher course in computer science

Advanced Scientific Computing Master

Université Lille 1

Adrien Poteaux *

adrien.poteaux@lifl.fr

Year 2014-2015

This work is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

http://creativecommons.org/licenses/by-nc-sa/3.0/

*inspired by a lot of works with creative commons license on the web or in Lille 1, see annexes
# Contents

1 **Using Linux** ........................................... 5
   1.1 Command line interface ........................................... 5
   1.2 Running a command ........................................... 6
   1.3 Structure of files and directories ................................... 8
      1.3.1 Dealing with the filesystem ................................... 8
      1.3.2 Viewing files and directories ................................... 9
   1.4 Basic and useful commands ..................................... 9
      1.4.1 Getting help: the man command ................................ 9
      1.4.2 Getting files from the web .................................... 10
      1.4.3 Data Compression ............................................ 10
      1.4.4 Distant connection .......................................... 11
      1.4.5 Making links ................................................. 11
      1.4.6 Finding files ................................................ 12
      1.4.7 Filtering data ............................................... 12
   1.5 Permissions on files .......................................... 13
   1.6 More details on shell and commands ......................... 14
      1.6.1 Aliases .................................................... 14
      1.6.2 Interactions with commands .................................. 14
      1.6.3 Jobs ...................................................... 16
   1.7 Variables .................................................. 17
   1.8 Scripts ................................................... 18
      1.8.1 First definitions ........................................... 18
      1.8.2 Shell parameters .......................................... 19
      1.8.3 Some control structures .................................... 19
      1.8.4 One example ............................................. 20

Practical works ................................................. 22
   Exercise 1.1: discovering shell commands .......................... 22
   Exercise 1.2: text manipulations ................................... 23
   Exercise 1.3: around the find function ................................ 24
   Exercise 1.4: first scripts ......................................... 25
   Exercise 1.5: getting memory informations ............................ 26
   Exercise 1.6: generating files ...................................... 27

2 **Makefile** ............................................... 29
   2.1 The make command ............................................ 29
   2.2 More details following a basic example ............................ 29
   2.3 Macro and abbreviations ....................................... 31

3 **C language** ............................................... 35
   3.1 What’s C? ................................................ 35
      3.1.1 ANSI C: a very small and simple language ..................... 35
      3.1.2 A low-level programming language ............................. 36
      3.1.3 A first example ........................................... 36
      3.1.4 Warning ................................................ 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.5</td>
<td>Principle of compilation</td>
</tr>
<tr>
<td>3.2</td>
<td>More details</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Operation priorities</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Identifiers</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Control structures</td>
</tr>
<tr>
<td>3.3</td>
<td>Preprocessor</td>
</tr>
<tr>
<td>3.4</td>
<td>Structures</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Table</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Struct</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Enumeration</td>
</tr>
<tr>
<td>3.4.4</td>
<td>union</td>
</tr>
<tr>
<td>3.5</td>
<td>Pointers</td>
</tr>
<tr>
<td>3.5.1</td>
<td>What is a pointer?</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Pointer arithmetic</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Dynamic allocation</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Pointers and tables</td>
</tr>
<tr>
<td>3.5.5</td>
<td>Pointers and structures</td>
</tr>
<tr>
<td>3.5.6</td>
<td>Linked list</td>
</tr>
<tr>
<td>3.6</td>
<td>Function and variables</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Variables</td>
</tr>
<tr>
<td>3.6.2</td>
<td>The main function</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Pointers on function</td>
</tr>
<tr>
<td>4</td>
<td>Fortran language</td>
</tr>
<tr>
<td>4.1</td>
<td>First practical works in Fortran</td>
</tr>
<tr>
<td>4.2</td>
<td>Computing areas and volumes for a given radius</td>
</tr>
<tr>
<td>4.3</td>
<td>Loop order for matrices</td>
</tr>
<tr>
<td>4.4</td>
<td>Dynamic allocation in Fortran</td>
</tr>
</tbody>
</table>

First exercises in C
- Exercise 3.1: Priority of operations
- Exercise 3.2: Introduction to the C language
- Exercise 3.3: first programs in C
- Exercise 3.4: around floating point numbers
- Exercise 3.5: the gdb debugger
- Exercise 3.6: around the preprocessor

First project: the exponential function
- Exercise 4.1: Newton’s method
- Exercise 4.2: Computing areas and volumes for a given radius
- Exercise 4.3: Loop order for matrices
- Exercise 4.4: Dynamic allocation in Fortran

Practical works on structures and tables
- Exercise 3.7: dealing with big positive integers
- Exercise 3.8: sorting function
- Exercise 3.9: numerical computations

Exercises on pointers and structures
- Exercise 3.10: pointers and matrices
- Exercise 3.11: length of a string
- Exercise 3.12: circular shift
- Exercise 3.13: cross products with pointers
- Exercise 3.14: concatenating strings
- Exercise 3.15: defining sparse polynomials
5 POSIX
  5.1 Filesystem ......................................................... 81
    5.1.1 File descriptors ............................................. 81
    5.1.2 The open and close functions ................................. 81
    5.1.3 Reading and writing in a file ................................ 82
    5.1.4 The strace command ........................................... 84
    5.1.5 Getting information on a node ................................. 85
  5.2 Coding in POSIX .................................................. 85
    5.2.1 POSIX constraints ............................................. 85
    5.2.2 POSIX source code ............................................. 86

Tutorial works ............................................................. 87
  Exercise 5.1: Our which command .................................... 87
  Exercise 5.2: a mfind command ....................................... 89
  Exercise 5.3: Starting a new process ................................ 90
  Exercise 5.4: Redirection of the standard output ................... 90

6 Final project ............................................................ 91
  6.1 A generic quicksort algorithm .................................... 91
  6.2 Disc usage ......................................................... 92
  6.3 A pipe function .................................................. 93
  6.4 Testing the different commands ................................... 94

7 Resources ............................................................... 95
  7.1 Emacs .............................................................. 95
    7.1.1 Some hints about Emacs ...................................... 95

8 Annexes ................................................................. 97
  8.1 Inspirations of this lecture ..................................... 97
1 Using Linux

An Operating System enables applications to use the material resources of the computer and to communicate (using RAM, hard drive...). In this lecture, we will focus only on the Linux operating system. We will provide more details on that operating system (how the system deals with the filesystem, how process work...) in chapter 5; for now on, we describe an important tool you should use a lot: the shell, how it works, useful Linux commands, and how to write some scripts (i.e. programming in shell). We will also describe the structure of files and directories in Linux for a understanding well how to use the terminal.

Note that the first exercise of the practical work is done so that you try some basic commands.

Contents

1.1 Command line interface .................................. 5
1.2 Running a command ....................................... 6
1.3 Structure of files and directories ......................... 8
  1.3.1 Dealing with the filesystem ............................ 8
  1.3.2 Viewing files and directories .......................... 9
1.4 Basic and useful commands .............................. 9
  1.4.1 Getting help: the man command ....................... 9
  1.4.2 Getting files from the web ........................... 10
  1.4.3 Data Compression ..................................... 10
  1.4.4 Distant connection .................................... 11
  1.4.5 Making links .......................................... 11
  1.4.6 Finding files .......................................... 12
  1.4.7 Filtering data ........................................ 12
1.5 Permissions on files ...................................... 13
1.6 More details on shell and commands ..................... 14
  1.6.1 Aliases .................................................. 14
  1.6.2 Interactions with commands ........................... 14
  1.6.3 Jobs ..................................................... 16
1.7 Variables .................................................. 17
1.8 Scripts ..................................................... 18
  1.8.1 First definitions ....................................... 18
  1.8.2 Shell parameters ...................................... 19
  1.8.3 Some control structures ............................... 19
  1.8.4 One example .......................................... 20

1.1 Command line interface

What is the shell? A shell is both a command language and a programming language that provides an interface to the UNIX Operating System. To summarise, it enables to:

- start a process (considered as a filter),
- combine (redirection, pipe) filters, thus associated data and effects,
• program using control structures,
• send signals to process (kill),
• provide variables to a process started from a shell.

One can access by using a terminal emulator such as xterm, konsole, terminal... (there are many others, use the one of your choice). Under Linux, one can also access to a terminal by the key combinations Ctrl + Alt + F1 to Ctrl + Alt + F6 (you will have to log in there; one can come back to the graphical interface by the key combination Ctrl + Alt + F7).

There are many variants of Unix shells. In this lecture, we will assume that bash is used. You can see what is used by default in your computer with the following command:

```
ls -l /bin/sh
```

**Why using command line and not graphic tools?** At first, command lines were the only way to interact with a computer. Since then, graphical interfaces appeared, usually more intuitive to use at first. Using command lines requires some knowledge, but is very useful: it is a lot faster for most operations once you’re used to it, and there are operations that can be done only with commands. Here are a couple of examples (you should be able to understand them with the informations given in this chapter).

- `ls *.jpg | wc -l` will count the number jpeg pictures in the current directory (assuming they are correctly named).

- `for i in *.JPG; do mv "$i `basename "$i .JPG`.jpg"; done` change the extension JPG by jpg for any file concerned in the current directory.

Finally, it can appear that you will not have access to any graphical interface (for instance, when connecting to make computations on a server that does not provide one). In such cases, command lines will be your only option.

### 1.2 Running a command

**Basic command.** The basic way to run a command is to write something as

```
cmd [options] parameters
```

on the terminal, and then press on Enter. For instance, the command `ls` will list the files in the current directory. According to the command you use, parameters are optional or not. For instance, the command `ls dir` will print the content of the directory dir (if there is a file dir in the current directory that is a directory).

Also, one usually have available options with a command. They are actually parameters, that commonly begin with one or two dash: one dash for short options (one letter), and two dash for longer ones (sometimes, a short option is just a shortcut for a longer one). For instance, `ls -l` with list the files in the current directory together with additional informations, `ls --size` will print them together with their allocated size, as will `ls -s`. Finally, one can usually combine several short options with only one dash: `ls -lh` is similar to `ls -l -h`.

To know more on the way a command works, you should always consult its man page at first (see section 1.4.1).

Finally, a very useful fact is the auto-completion of commands: you do not have to type every character of a command! Write the first characters, then hit the Tab key, and the shell will either complete your typing (if there is a single command beginning with these letters), either prompt all the commands that begin with these letters (making completion up to the point where these commands do not begin the same). For instance, you can type `his`, then hit `Tab`, and it should be completed in `history`.

---

**Command-line editing.** Here are some shortcuts that one can use on a shell and that might be useful to you once you’ll be more used to the shell.

- ↑ and ↓ arrows scroll through your bash history
- Ctrl + r followed by a word will search backward through your bash history the last command containing this word. Another Ctrl + r searches the previous one, and so on (see also the history command).
- Ctrl + l clears the terminal screen (similar to the clear command; note also the existence of the reset command).
- Ctrl + d sends the end of file (EOF) message to the terminal. If the command line is empty, it will close the terminal (as the exit command); otherwise, it will erase the current digit. While running some program (like a C program that runs until the EOF character is read - see chapter 3), this can be the only way to finish it properly,
- Shift + PgUp and Shift + PgDown enables you to respectively scroll up / down in the terminal (the mouse wheel does the same),
- Ctrl + a put the cursor to the beginning of the line (similar to the Home button),
- Ctrl + e put the cursor to the end of the line (similar to the End button),
- Ctrl + u deletes every character before the cursor on the current line,
- Ctrl + k deletes every character after the cursor on the current line,
- Ctrl + w deletes the first word on the left of the cursor (words are separated by spaces),
- Alt + d deletes the first word on the right of the cursor (note that several terminal emulators overload this key combination)
- Ctrl + y paste deleted text by one of the last key combination above (Ctrl + u, Ctrl + k, Ctrl + w or Alt + d).
- the Ctrl key amplifies some editing functions in the bash shell, so that Ctrl + ← and Ctrl + → will move the cursor by a word instead of by a character.

**Regular expressions.** A regular expression is a formal way of expressing a pattern. This is useful for dealing with parameters: for instance, `ls * .c` will list all the C files in the current directory. We just present some basic regular expressions here:

- ? is any character: `b?l` corresponds to the words `bol` or `bal`, but also `bwl`, `b3l`...
- brackets `[ ]` specifies a set of characters
  - `dupon[dt]` is either `dupond` or `dupont` (no other choice),
  - `dupon[d-t]` is `dupon` plus any character between `d` and `t`,
  - `dupon[^dt]` is `dupon` plus any character except `d` and `t`,

- * is 0, 1 or any number of any character (cf the example above).

Note that regular expressions are not working the same way everywhere (even if the global idea is similar). For instance, if you use regular expressions inside emacs, you will notice differences with the above examples.
1.3 Structure of files and directories

Files are organised in a kind of tree (it is actually more a graph, see below), as shown on figure 1.1. The root / of this tree is the only node that does not have any parent; every other file has a parent: the directory that contains it. To complete a full path, the separator is /; for instance, ls /home/poteaux will list the files inside the directory poteaux, wherever you run the command from.

There are two ways to provide the path of a given file: either by the full path, from the root /; either by giving the relative path, i.e. the path from the current node (where we are in the tree). ~ and $HOME (see section 1.8.2 for the definition of variables) always correspond to our home directory; .. corresponds to the current directory, and . . is the parent of the current node.

1.3.1 Dealing with the filesystem

Here are some basic commands useful to get informations on the filesystem

- mkdir newdir creates a new directory (make directory),
- cd newdir moves inside this directory (change directory); if no parameter is given, cd moves to your home directory (same as cd ~ or cd $HOME),
- rmdir removes a directory,
- pwd prints the (full) path of the current directory (print working directory),
- which cmd tells us where is located the command cmd,
- ls lists files in the current directory. You can have a look to the result of the following commands: ls -lt, ls -lat, ls -ls, ls -lSh, ls -lRh (see the man page to understand these options more precisely),
- touch filename creates an empty file or modify the date of the file if it already exists,
- cp, mv and rm respectively copy, move and remove files,
- du -sh * evaluates the size of the files of the current directory (disk usage); if a file is a directory, this is computed recursively by adding the sum of the sizes of all files inside this directory.
1.3.2 Viewing files and directories

- `file` provides the type of a file,
- `cat`, `more` and `less` enable to see the content of a text file (see below for some details on the last one),
- `head` and `tail` print respectively the beginning and the end of a given file,
- `wc file` tells you how many characters, words, and lines a file has,
- `diff file1 file2` tells you which lines differ and where,

The less viewer. Here are a few shortcuts that can be useful while using the less viewer:

- `h` will display the available commands,
- `q` will quit the viewer,
- `/` followed by an expression will search the next apparition of this expression in the document, move to the first results, and highlight the others,
- `n` moves to the next apparition of the searched expression,
- `N` moves to the previous apparition of the searched expression,
- `?` do the same as `/` but the search is made forward (therefore `n` and `N` are somehow reversed),

Writing code. It is very important to use a good text editor when writing code. Such editors normally provide specific modes that correspond to the language we are writing (text colouring, automatic indentation, comment a paragraph, compile from the text editor with links to errors, running the debugger etc).
Some details for Emacs are given in section 7.1.

1.4 Basic and useful commands

In this section we describe some native Linux commands and behaviour that you might need this year. There are obviously many other...

1.4.1 Getting help: the man command

The `man` command is very important command: it will enable you to get help on basic Linux command (understand the behaviour, know which option to use...). You should use that command before searching on the web for help on a command. For instance, `man ls` will print on the screen the documentation on the `ls` command.

Remark 1.1. *The man pages are shown using the less viewer (see section 1.3.2)*

Man pages are separated in several sections. The following ones should be useful during the year (`man man` will provide you more information !):

1. Programs that can be executed by the user from within a shell,
2. System calls (see chapter 5),
3. Library calls (most of the libc functions, see chapter 3),
7. Overview, conventions, and miscellaneous

If you are interested in getting help on something that is documented in several sections, you need to precise the section you are looking for; by default, the smallest one will be considered.

Example 1.1. `man printf` will give you information on the shell command `printf`, as would do `man 1 printf`. If you want information on the C function `printf`, you need to precise the section 3: `man 3 printf`.

1.4.2 Getting files from the web

`wget` will retrieve `ftp` or `http` URLs; it can continue a failed retrieval, create a new name if a file already exists, and support a huge number of other options.

For instance, the following command will download the file script.sh from my webpage (note that one usually put the URL inside double quotes).

```bash
wget "http://www.lifl.fr/~poteaux/fichiers/teaching/script.sh"
```

If you run this command a second time, it will be saved as `script.sh.1`. As usual, see the manpage for more details.

1.4.3 Data Compression

Compression is useful because it helps reduce resource usage, such as data storage space or transmission capacity. We present here a few commands that enable to compress (or decompress) data. There are lots of compression utilities (the main point being the trade-off we want between the quality of the compression and the time it takes to compress/decompress it). You can have a look on the web to find the most appropriate one.

**gzip/gunzip.** The gzip family is probably the most popular (de)compression utility on Linux. It will reduce the size of a sequence file to about 30% of the original.

`gzip somefile` will result in only the compressed file `somefile.gz`. To decompress such a file, one can either use the `gunzip` command, or the `gzip` one, with the option `-d`.

Note the existence of the `pigz` program for parallel (de)compression on multi-chore computers; this would be useful only for large files nonetheless.

**bzip2/bunzip2.** Another major (de)compression tool on Linux is called bzip2 and is slightly more efficient, and slightly slower than gzip. Basic usage is similar to gzip.

**tar.** `tar` is the default archive type for Linux/Unix. It creates a single file that contains everything that it’s pointed to, which provides some packing efficiency, and especially when it is explicitly compressed, a tar file will take only 30-50% of the original storage. Some examples of usage:

- `tar -tvf file.tar` lists the content of a tar archive,
- Extraction can be done as `tar -xvf file.tar`,
- `tar` also supports decompressing while untarring: `tar -xzvf file.tar.gz`. This also works for instance with a file compressed with bunzip2 (option `-j`, cf the man page).

Remark 1.2. For the practical work you will be asked to send your files in a `tar.gz` archive. You can do so with such command:

```
tar tzf archive-name.tar.gz file1 file2 file3
```

1.4.4 Distant connection

**ssh.** You can connect to a server (or a computer that enables such connection) with one of the following commands:

ssh login@server-name  
ssh -l login server-name

(you will be asked for your password).

Note that with such command, you will not have access to any graphical interface while working on the server, even if it provides one. Options -X will enable that.

**Example 1.2.** For instance, I can connect to the seventh computer of this room with the following command:

ssh -l poteaux mathcalc07.priv.univ-lille1.fr

Note however that this should work only from the university (a functional vpn is needed from the outside).

**scp.** You can copy files from one computer to another, or from a computer to a server by using the command `scp` (secure copy) in the following way:

scp file-to-send account@server:path

If not precised, account is the same than the one from the computer you are using, and path is your home directory (see section 1.3).

**Example 1.3.** The following command

scp file1.txt poteaux@mathcalc07.priv.univ-lille1.fr:/tmp/

copies file1.txt on the seventh computer of the room (note that your home directories are actually located on a server in the M2 building; the `/tmp` directory is however located on the computer; this remark is important when working on exercise 1.3).

**rsync.** Note the existence of this command to copy files on another computer/server. In particular, it is useful to synchronise your files with the one of a server (for instance): the second time you run the same command, if one of the file did not change, it will not be copied a second time. Since this is not important for this lecture, we do not provide more details here.

1.4.5 Making links

We first presented the Unix filesystem as a tree in section 1.3. It is actually more a graph than a tree.

Additionally to the tree structure, there are 3 additional kind of path inside the filesystem organisation, as illustrated on figure 1.2:

- . always represents the current directory; for instance, `cd .` will actually do nothing.
- ../ is the parent directory; `cd ../` will move one step higher in the “tree”.
- It is possible to create link with (for instance) the `ln` command. The link represented on figure 1.2 can be obtained with the following command (from poteaux’s home directory):

`ln -s /directory/file link`

Links can be “hard” (the created link represents exactly the same file) or “symbolic” (the created link is somehow the “arrow” to reach the linked file); on the example, as we used the `-s` option, the created link is a symbolic one.

---

1.4.6 Finding files

Even the most organized among you will occasionally lose track of where your files are, or where are stored some Linux files. We present here a couple of tools to find them.

**locate.** It is a very fast command; indeed, it is actually searching in a database. Nevertheless, it will only be able to find files that have been indexed (there is a command `updatedb` to do so, but it can be run only by the root user). Thus, in practice, it can only help you to locate commands or such.

**find.** This is a very powerful tool to search for a file, but also execute operations on found files. It requires however some knowledge...

One runs the command as `find where what 'what to do'`. A basic example to search for a file `toto.txt` in the current directory and its children (recursively) is:

```
find -name toto.txt
```

Exercise 1.3 is about that function.

1.4.7 Filtering data

**grep.** The grep command is essential, and you might use it very often. Basically, the command

```
grep regexp file
```

will print every line of the given file containing a word satisfying the regular expression `regexp`.

**Example 1.4.** `grep alias ~/.bashrc` _will print every line of your_. `bashrc` _file containing the word alias (i.e. probably all the aliases you defined there)_.

If no file is given, the command `grep` is run on the standard input (which is really useful to use it as a filter via a pipe). Note that regular expressions of `grep` have differences with the one of the shell:

- `.` is any character,
- `^` represents the beginning of a line (useful to find only words that start at the beginning of a line),
• $ represents the end of a line,
• square brackets [ ] are similar to the shell (any character inside ; remember that a ^ after the opening square bracket means that we are considering any character except the ones inside the square brackets),
• ?: the previous element is optional (appears 0 or 1 times),
• *: the previous element appears any number of times (0, 1 or more),
• #: the previous element appears one or more times (at least 1),

Example 1.5. Let consider the following file:

```
$ cat toto
andliftests
afra345endqr
this is a test file
12and too close
23 ird with a space
nothing here
end of the test file
```

Then the output of the following grep commands gives:

```
$ grep test toto
andliftests
this is a test file
end of the test file
$ grep "^\[0-9\]\+[ ]?\[aei][rn]\d" toto
12and too close
23 ird with a space
```

As usual, they are a lot of options that can be used: −n to numerate the found lines, −r to make a recursive search in a given directory etc (as usual, cf man grep; it should also provide more details on regular expressions).

Removing columns of a file. There are two main commands to do so: cut and colrm. The first one removes columns that are defined by a separator (default is tabulation), while the second removes “classical” columns.

1.5 Permissions on files

Linux has a Unix heritage so everything has an owner and a set of permissions. When you ask for an ls −l listing, the 1st column of data lists the following:

```
$ ls −l |head
total 820
drwxr-xr-x 5 adrien adrien 4096 aug 22 16:51 c
−rw-r--r-- 1 adrien adrien 1913 aug 29 17:20 comlnew.sty
drwxr-xr-x 8 adrien adrien 4096 jul 9 14:43 corrections
drwxr-xr-x 6 adrien adrien 4096 aug 22 16:51 fortran
−rw-r--r-- 1 adrien adrien 5103 jul 9 14:43 ideas.txt
−rw-r--r-- 1 adrien adrien 642 sep 1 13:37 makefile
```
Let see on an example the meaning of these informations:

```
3.8.2 permissions information
lrwxrwxrwx 1 adrien adrien 49 aug 25 11:55 mylogos.sty -> ../mylogos.sty
+-+-+-+-+-
| | | | | | | other permissions
| | +----- group permissions
| +-------- user permissions
----------- link/directory bit
```

Let see on an example the meaning of these informations:

```
drwxr-xr-x 5 adrien adrien 4096 aug 22 16:51 c
+-+-+-+-+-
| | | | | | other can read and execute
| | +----- group can read and execute
| +-------- user can read, write and execute
----------- it's a directory
```

One can change permissions of this directory by using the `chmod` (change mode) command:

```
chmod -R o-rx c
```

The result is:

```
$ ls -ld c
```

Finally, note that the `chown` (change ownership) enables to change owner and group of a file; however, it usually requires root’s privileges.

### 1.6 More details on shell and commands

#### 1.6.1 Aliases

A useful tool while using bash are aliases. This enables to create a shorter version of a command (that you’re suppose to run quite often).

For instance, this creates an alias `nu` that prints the 22 newest files in the current directory:

```
alias new="ls -lth | head -22"
```

It is possible to load such aliases automatically at the opening of a terminal by adding such a command in your `.bashrc` file (it is in your home directory).

#### 1.6.2 Interactions with commands

By default, common commands print their results on the screen; it is however to redirect this output into a file. More generally, a unix command has one standard input and two standard outputs, as shown in figure 1.3:
• the standard input is a way to interact with the command; by default, input can be provided via the keyboard,

• the standard output is where the result are printed (by default the screen, i.e. the shell),

• the error output is where error messages are printed (also the screen by default).

**Redirections.** It is possible to redirect input and outputs of a command. > and >> enable to redirect the output of a command to a file: the first one redirects the standard output to a given file (overwriting any data this file could contain before), the second one adds it to the given file. In both cases, the file will be created if it does not exist already.

**Example 1.6.**

- `ls > toto.txt` does not print on the screen, but store the result of the command inside the file `toto.txt` instead.

- Assuming one then runs the command `ls .. > toto.txt`, the previous storage is overwritten.

- If we had run `ls .. >> toto.txt` instead, the file `toto.txt` would contain the result of the two commands `ls` and `ls ..` one after the other.

**Remark 1.3.** It might happen that you do not care of the result of a command (i.e. you do not want to store the result in a file, neither “pollute” the shell with the result); in such a case, it is possible to redirect the result in `/dev/null` (it is a kind of black hole; any data going there vanishes).

The previous redirection concerns only the standard output; on the previous examples, error messages are still printed on the screen. Redirections > and >> are actually the same than respectively `1>` and `1>>`. To redirect error outputs, one can thus use `2>` and `2>>` (note that one can output the two outputs on two different files).

It is also possible to merge both outputs with the following code: `2>&1`. This makes error output and standard output the same output (writing at the same place and with the same rule about overwritten).

**Example 1.7.** Here are some examples of redirections of the different outputs:

- `cmd 1>output.txt 2>error.txt` will write the standard output of the command in the file `output.txt`, and the standard error in the file `error.txt`; both will be overwritten.
• `cmd >output.txt 2>&1` will write both standard and error outputs in the file `output.txt`, overwriting the file if it already exists; the command `cmd >output.txt 2>&output.txt` would do the same; the command `cmd >>output.txt 2>&1` is similar, but here the file is not overwritten.

• `cmd >>output.txt 2>error.txt` is almost the same than the first item; the only difference is that the standard output will not be overwritten (but the error one is).

Finally, it is also possible to provide the standard input from a file by using `<`: the command `cmd <input.txt` will read its standard input inside the file `input.txt` instead of waiting that input from the keyboard.

**Pipe.** It is also possible to combine several command by using pipes, i.e. the character `|`. Let’s explain this one the example given in section 1.1:

```
ls *.jpg |wc -l
```

Here, the command `wc -l` takes its standard input from the standard output of the command `ls *.jpg`, i.e. there are here two redirections (the command `ls` will not be printed on the screen, and the command `wc` will not wait data from the user’s keyboard). The behaviour of the pipe will be detailed more precisely in chapter 5.

### 1.6.3 Jobs

When you type a command, you cannot use the terminal anymore, until the command finishes its execution. For instance, if you run the command `emacs`, you will not be able to use your shell until you close Emacs. This is because the command is run in *foreground*. It is possible to run it in *background* by adding a `&` at the end of the command line: `emacs &`. If you forgot to do so, you can send the foreground job by first typing `Ctrl - z` and then running the command `bg` (this command takes a job-id as a parameter, by default it will put in background the last stopped job). It is also possible to put back a job in foreground (`fg`), to kill a job in foreground (`Ctrl - c`) or in background (via the command `kill`). Figure 1.4 summarises these possibilities. Once again, more details will be given in chapter 5 (specially on signals).

![Figure 1.4: Jobs, background, foreground and signals](image-url)

Also, for any process, the system stores a lot of informations: its PID (*process identifier*), PPID (*parent pid*), state (running, zombie, stopped...) and many others. There are several commands to get informations. We already mentionned `jobs`, let also mention `top` and `ps`. Note that the second one usually
needs to be used with another function to extract the information we are looking for: by default, it only lists processes run by the user in the current shell; it ones want to find any process, option \texttt{-e} can help, but output is too big; filtering it with the \texttt{grep} command is usually the way to go.

**Example 1.8.** The following command will print all \texttt{emacs} process running on the computer:

\begin{verbatim}
$ ps -ef |grep emacs
poteaux 4223 3442 0 13:56 pts/0 00:00:31 emacs slides-shell.tex
poteaux 9157 3442 0 15:39 pts/0 00:00:00 grep emacs
\end{verbatim}

With this information, one can kill process 4223 (corresponding to the command \texttt{emacs slides-shell.tex}) with the command \texttt{kill 4223}.

We saw two different utilisation of the \texttt{kill} command: either we provide the process identifier (4223 on the previous example), either we provide the job identifier (preceded by a \%). Finally, not also the existence of the \texttt{killall} command.

\section*{1.7 Variables.}

A variable is defined as soon as it is affected: $ FOO="Hello world"

\begin{verbatim}
echo prints its given argument:

$ echo FOO
FOO
\end{verbatim}

To evaluate a variable, one adds $ before its name:

\begin{verbatim}
$ echo $FOO
Hello world
\end{verbatim}

In a shell, \textit{everything is a character chain}. Each command is a chain evaluated by the shell. There are three delimiters:

- quotes ‘ ’ disable evaluation;
- quotation mark " " make a character chain after evaluation of what is inside;
- backquotes ` ` make a chain evaluated as a command.

\begin{verbatim}
$ echo '$FOO'
$FOO
$ echo "echo '$FOO'"
echo 'Hello world'
$ BAR="anything you want" ; echo $BAR
anything you want
$ BAR=`anything you want`
anything: Command not found.
\end{verbatim}

\textbf{Around variable names.} It is possible to extract parts of a variable content with the followings:

- $\{\text{parameter}\%\text{regexp}\}$ removes the shortest suffix defined by regexp in the evaluation of parameter,
- $\{\text{parameter}\%\text{regexp}\}$ removes the longest suffix defined by regexp in the evaluation of parameter,
• ${parameter#regexp}$ removes the shortest prefix defined by regexp in the evaluation of parameter,
• ${parameter##regexp}$ removes the longest prefix defined by regexp in the evaluation of parameter

It might be easier to understand the behaviour of this trick with examples:

Example 1.9.

$ FOO=babarerre.tar.gz
$ echo ${FOO%ba*}  $ echo ${FOO#ba}
ba            barerre.tar.gz
$ echo ${FOO%%ba*}  $ echo ${FOO##ba}
$ echo ${FOO%re*}  $ echo ${FOO##ba*}
baba          baberre.tar.gz
$ echo ${FOO%re*}  $ echo ${FOO%ba}
babarerre.tar.gz
$ echo ${FOO%ba}  $ echo ${FOO%.*}  $ echo ${FOO##ba*}
babarerre.tar.gz

1.8 Scripts

1.8.1 First definitions

One can put shell commands in a file foo; then, one can either

• Interpret the file with the current shell via . foo;
• Makes the file executable (\$ chmod u+x foo) and use it directly ; that will be interpreted in another shell.

Definition 1.1. A script is a file containing shell commands. If one want it to be executable, we put in the first line which interpret we want to use (for instance, #!/bin/sh)

Remark 1.4. /bin/sh is just a link to the shell interpreter you are using. It can be for instance bash, dash or csh. You can see which version you are using via the command $ ls -l /bin/sh

Remark 1.5. Of course, a script can also be used by the function call sh -[opt] name_of_the_script

The position parameters in a script are the arguments of the line using it. Going to a new line in the file terminates the command. For instance, the command

if [ $# -ne 2 ] ; then echo "pb" ; fi

is the same thing than writing

if [ $# -ne 2 ]
then echo "pb"
fi

Function. It is also possible to define functions in a script. The syntax is the following:

name_of_the_function()
instructions
redirection of an exit status

One can call a function as follows:

name_of_the_function parameters
1.8.2 Shell parameters

Definition 1.2. A parameter of a shell is either a number, a special character (see below) or a name (sequence of alphanumeric characters that is not a number or a special character). A variable of a shell is a parameter corresponding to a name. A position parameter is a parameter that is not special and not a variable.

We say that a parameter is allocated if it has a value (null is a value). To disallocate a variable, the only way is through the command `unset`.

Special and positions parameters When writing scripts, there are several predefined parameters that will be useful:

- 0: current command name;
- #: number of position parameters;
- *, @: all the position parameters;
- 1 to 9: the 9 first position parameters;
- x: the position parameter \(x(>9)\);
- $: the pid (process identifier) of the current command;
- _: the last used parameter;
- -: the flags (options) of the current command;
- ?: the `exit-status` of the last command you ran.
  - everything ok: \(? = 0\),
  - something abnormal: \(? \neq 0\);
- $!: the pid of the last run process in background

The `shift` command shifts the numbered parameters (1 is lost and # is updated)

1.8.3 Some control structures

- Simple condition:

  ```bash
  if instruction-test
    then instruction
  elif other_instruction-test
    then other_instruction
  else last_instruction
  fi
  ```

  If `instruction-test` has an exit status equal to 0, then `instruction` is executed, otherwise, if `other_instruction-test` has an exit status equal to 0, then `other_instruction` is executed. Otherwise, `last_instruction` is executed.

- Multiple conditions: the function `case`
Once a condition has been satisfied, the other conditions are not executed (this is different in C).

- **Numerated iteration:**

  For *var* in *expr* do
  
  *instruction*
  
  done
  
  - *If* *expr* is empty, the instruction is not executed,
  
  - *in* *expr* can be omitted; by default, this will be in "$@".

- **Conditional iteration**

  while *instruction-test* do
  
  *instruction*
  
  done

- **Inverse conditional iteration**

  until *instruction-test* do
  
  *instruction*
  
  done

**The test command.** Most of the time, the *instruction-test* used after a *if*, *while* or *until* keyword will be provided using the *test command*, that can be called by using brackets [ ]:

- *test expression* is the same as [ *expression* ] (spaces are mandatory). This enables to check the type of a file, to make comparisons in between values (*if* [ *$A* -eq 3 ] ; then *cmd* ; fi will run the command *cmd* if the variable *A* is equal to 3), to compare strings... cf the man page for more information.

### 1.8.4 One example

```bash
#!/bin/bash
NBPARAM=2 # number of parameters of the script

usage()
# How to use this script
{
  echo "Usage: `basename $0` firstparam secondparam"
  echo "Print "'firstparam' and 'secondparam'"."
  return 0 # exit status of the function
}

# We test the number of parameters and use
# the usage() function if it is different from 2
if [ $# -ne $NBPARAM ]
then
```
usage # the function call does not use ()
exit 1
fi

# We output the parameters
echo $1 and $2
# and everything is fine
exit 0
Practical work on shell

Contents

Exercise 1.1: discovering shell commands ................................. 22
Exercise 1.2: text manipulations ............................................. 23
Exercise 1.3: around the find function ..................................... 24
Exercise 1.4: first scripts ...................................................... 25
Exercise 1.5: getting memory informations ................................. 26
Exercise 1.6: generating files .................................................. 27

Preliminary: in this lecture, a line beginning with a $ means a shell command. Remember the man command to get informations on the different commands used here.

Exercise 1.1. [First steps on a shell] To discover the basic commands, we will first create some directories and files by using a script. Open a terminal and download this script by using the wget command:

$ wget "http://www.lifl.fr/~poteaux/fichiers/teaching/script.sh"

The command ls shows the content of a directory. With the option -l, we get more information about each file.

poteaux@computer:~$ ls -l
-rw-r--r-- 1 poteaux poteaux 513 7 sept. 18:45 script.sh

One can see here that the file script.sh is not executable. This is the case after the following command:

$ chmod u+x script.sh

The result of this command can be seen by using once more the ls function:

poteaux@computer:~$ ls -l
-rwxr--r-- 1 poteaux poteaux 513 7 sept. 18:46 script.sh

Remark 1.6. The command man read the manual entry on a command (or some other points). You can use the command $ man chmod to understand the previous instruction you just did.

Now we can execute the script by the command $ ./script.sh

Remark 1.7. There are other ways to run this script. For instance, one could have use the commands sh script.sh, bash script.sh, source script.sh or . script.sh. With such commands, the file does not need to be executable. Note that the last two ones does not give the same result. This is the topic of the second question of exercise 1.6.

You can now explore what this script created by using some basics unix commands. Remember the man function to have more details about these commands and its options.

- cd directory changes the current directory. Without argument, one returns to the home directory.
  cd – moves back to the previous directory.
• `cat`, `more`, `less` enable to see the content of a file.

You can also try the following ones:

• `rm` deletes a file.
• `mv` moves one or several files, or renames one.
• `pwd` shoes the current directory.
• `cp` copies a file.
• `mkdir` create a new directory.
• `rmdir` removes an empty directory.

Then, you can look at the content of the downloaded script and try to understand the different commands it generated (once again, remember the `man` command).

**Exercise 1.2.** [text manipulations] In this exercise, we will manipulate the famous intimate mail exchange between George Sand and Alfred de Musset (note that you do not need to copy-paste these data: they have already been downloaded while running the script of exercise 1.1).

1. We first want to keep only the odd lines of the following text, that you can find in the `ex2` directory created in the previous exercise:

   Je suis très émue de vous dire que j’ai bien compris l’autre soir que vous aviez toujours une envie folle de me faire danser. Je garde le souvenir de votre baiser et je voudrais bien que ce soit là une preuve que je puisse être aimée par vous. Je suis prête à vous montrer mon affection toute désintéressée et sans calcul, et si vous voulez me voir aussi vous dévoiler sans artifice mon âme toute nue, venez me faire une visite. Nous causerons en amis, franchement. Je vous prouverai que je suis la femme sincère, capable de vous offrir l’affectation la plus profonde comme la plus étroite en amitié, en un mot la meilleure preuve dont vous puissiez rêver, puisque votre âme est libre. Pensez que la solitude où j’habite est bien longue, bien dure et souvent difficile. Ainsi en y songeant j’ai l’âme grosse. Accourrez donc vite et venez me faire oublier par l’amour où je veux me mettre.

To do that, you can use the following guide:

a) Use the command `nl` to number the lines of the file `Sand.msg`

b) As we are only interested in the parity of the line, we only care of the last digit of each number: remove the useless column with the function `colrm`
c) Use the command `grep` to keep only the odd lines.

d) Remove the useless columns to print the result

2. Do the same process by using only a one line command (you’ll need to use the pipe `|` character)

3. The answer from Musset was an acrostic you get the meaning by reading the first word of each line:

Quand je mets à vos pieds un éternel hommage
Vous avez capturé les sentiments d’un cœur
Que pour vous adorer forma le Créateur.
Je vous chéris, amour, et ma plume en délire
Couche sur le papier ce que je n’ose dire.
Avec soin, de mes vers lisez les premiers mots
Vous saurez quel remède apporter à mes maux.
Bien à vous, Eric Jarrigeon

The answer of Sand was on the same kind:

Cette insigne faveur que votre cœur réclame
Nuit à ma renommée et répugne à mon âme.

Using the files `Musset.rep` and `Sand.rep`, first create files containing only the first column of each answer; this can be done with the `cut` function (for instance, one can use " " as a delimiter, and take only the first field). Then, create a single file containing these two columns written in two separated line. Finally, make the whole operations in a single line (the function `echo`, with the correct option, may help).

**Correction:**

```bash
$ echo -e `cut -d" " -f1 Musset.rep` 
"n"`cut -d" " -f1 Sand.rep`
```

**Exercise 1.3.** [function find] A phylogenetic tree⁴ is a “tree showing the inferred evolutionary relationships among various biological species or other entities based upon similarities and differences in their physical and/or genetic characteristics”. A part of the tree that you can find on the wikipedia page has been made in a file phylogenetic-tree.tar.bz2. But other data, that have nothing to do with biology, have also been introduced. The point of the exercise will be to find them and follow the instructions hidden in the tree.

**Remark 1.8.** As your data is not stored on your computer but on a server, it is important that you run this exercise from the `/tmp` directory (that is local to the computer) and not in your home directory; otherwise, you will request a lot of interactions with the server, and that will slow down a lot every use of the `find` command.

1. Use the command `wget` to get the file `phylogenetic-tree.tar.bz2`. You can find it at the url

http://www.lifl.fr/~poteaux/fichiers/teaching/phylogenetic-tree.tar.bz2

2. What is the size of this file ?

---

⁴http://en.wikipedia.org/wiki/Phylogenetic_tree
3. Decompress the file by using the command `bunzip2`.

4. What is the size of the file you get?

5. This is an archive file. You need to extract it by using the `tar` command.

6. What is the size taken by the extracted files (function `du`)?

Correction: The option `-sb` is useful here.

7. Find the file `charlie_01.txt` in the tree by using the command `find`, read it, and follow the instructions inside.

Correction: Once inside the `racine` directory,

```
$ less `find -name charlie_01.txt`
$ less `find -name lisezmoi.txt* ! -empty`
$ less `find ! -name lisezmoi.txt ! -name charlie_01.txt -type f`
$ find -type f -exec rm () ;
```

8. Now that you know a little more what is inside the original tree, do you see a way to find all the necessary files in only one search?

Correction: Once inside the `racine` directory,

```
$ less `find -type f ! -empty`
```

Exercise 1.4. [scripts]

- Without using `$@`, `$*` or `$#`, write a script that print all its parameters line by line

Correction:

```sh
#!/bin/sh
while [ -n "$1" ]; do
  echo $1
  shift
done
```

- Write a script that print the files inside a given directory line by line, with the type of the file written before (on a separated line). The script must test if there is a single parameter and that it is indeed a directory ; if not, this must print how to use the function

Correction:

```sh
#!/bin/sh
usage()
# How to use this script
{
  echo "Usage: `basename $0` directory"
  echo "Print the files inside directory and their type"
  echo "Must have a single parameter which must be a directory"
  return 0 # exit status of the function
}
if [ $# -eq 1 -a -d $1 ]
then
```
for i in `ls $1` ; do
  file -b $1/$i
  echo $i
done
else
  usage
fi

• Write a script that copy all the files with the extension .txt in a repertory given as an input, and that replace the extension by .old

**Correction:**

```bash
#!/bin/bash
usage()
# How to use this script
{
  echo "Usage: `basename $0` directory path1/file1.txt ... pathN/fileN.txt"
  echo "Copy the N files in directory, with an extension .old"
  echo "First argument must be a directory."
  echo "If a file does not have a .txt extension, it is ignored".
  return 0 # exit status of the function
}
if [ -d $1 ] # test if the first parameter is a directory
  then
    DIR=$1
    shift # we now consider only the files
    for i in "$@" ; do
      if [ ${i## *.} != "txt" ]
        then
          usage
        else
          cp $i $DIR/`basename $i .txt`.old
          # here we use the basename function and not something like {${i%.txt}
          # in order not to have a problem with the path of the new name
          fi
      done
    else
      usage
  fi
```

**Exercise 1.5.** [memory informations] The external command `ps` list the process currently running on the computer. With the options `-ux`, one get informations on the process of the current user: for each process, its PID, the percentage of CPU and memory and the size in kilobytes used VSZ, and a few more informations.

1. Write a script `memtot` that compute the sum in kilobytes of the memory used by the user’s process

**Correction:**

```bash
#!/bin/bash
# memtot
```
# A variable that contains the result
SUM=0

for i in `ps ux | cut -c26-31`
do
    SUM=$(( SUM + i ))
done

echo $SUM

2. Write a script `topuser` that print the user taking the most CPU; we will consider the round part of each percentage only

```bash
#!/bin/bash

MAX=0

for i in `ps aux | tail -n +2 | cut -f1 -d' ' | sort | uniq`
do
    SOMCPU=0
    for CPU in `ps aux | tail -n +2 | grep $i | cut -c16-19`
do
        CPU=${CPU%.*}
        SOMCPU=$(( SOMCPU + CPU ))
done
    if [ $SOMCPU -gt $MAX ]
        then
            MAX=$SOMCPU
            MAXUSER=$i
    fi
done

echo "$MAXUSER uses around $MAX percent of the CPU"
```

Exercise 1.6. [generating files]

1. Write a script that, given a file cmd.ext (ext can be any extension name) containing shell commands outputs a text file out.ext that contains the commands we would get. For instance, if one runs this script on a file cmd.txt that contains:

```
A=4
echo A
echo $A
B=5
expr A+B
```

one should create a file out.txt like that:

```
$ A=4
```
$ echo A
A
$ echo $A
4
$ B=5
$ expr A+B
A+B

2. Try your script for these commands:

ls thisfiledoesnotexist
echo $?

Run also them on a shell. Do you get the same result ? Why ? How to correct this ?

**Correction:** This answer is given in two files, but this is obviously not mandatory.

```bash
#!/bin/bash
# generateExec.sh
RET=0
# the option -r for the command read is used to
# desactivate the interpretation of the \nwhile read -r line; do
  if ! echo "$line" | grep -q "^[\ ]*$"; then
    if echo "$line" | grep -q "^#"; then
      echo "$line"
    else
      echo "$ $line"
      # this enables to get the old $?
      (exit $RET)
   enderror $line
    RET=$?
  fi;
fi;
done
exit 0
```

```bash
#!/bin/bash
#generateAll.sh
for FIC in cmd.*[^~]; do
  OUT=`echo $FIC | sed -e s/^cmd/out/`
echo "$FIC -> $OUT"
  rm -f $OUT
  ./generateExec.sh < $FIC > $OUT 2>&1
  if [ [ "$?" -ne "0" ] ]; then
    echo "Error on $FIC";
    exit 1
  fi;
done
```
2 Makefile

Remark 2.1. Makefiles are useful in a lot of ways. In this lecture, we will mainly use them when compiling code. Therefore, if you are not already familiar with C or Fortran, you might want to have a look at chapters 3 or 4 before.

When we separate a program in several files, one can improve the compilation phase (and save a lot of typing !) by using the Unix make command. The main idea of this command is to compile only what is necessary to create an executable. For instance, if only one of the source files has been modified, one do not want to recompile all the others. The make command searches in the current directory a file named makefile, or Makefile if it does not find it. This file specifies the dependencies between the different source files, objects and executable.

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The make command</td>
<td>29</td>
</tr>
<tr>
<td>2.2</td>
<td>More details following a basic example</td>
<td>29</td>
</tr>
<tr>
<td>2.3</td>
<td>Macro and abbreviations</td>
<td>31</td>
</tr>
</tbody>
</table>

2.1 The make command

This command automatically builds executable programs and libraries from source code by reading files called makefiles. These files specify how to derive the target program. This program is widely used for this purpose, especially in Unix. More generally, it can be used to manage any project where some files must be updated automatically from others whenever the others change. To do so, one has to write a makefile containing a list of dependence rules in the following way:

target: dependency list
<TAB> Unix commands

The first line specifies the target file, and then the list of files it depends on (separated by spaces), so that the next lines (which begins by a tabulation) are executed if and only if one of this file has been modified since the last modification of the target file.

Then, one can run the make command as:

$ make target

where target is one of the target defined in the makefile. If none is provided, make will run the first target defined in the makefile.

Finally,

2.2 More details following a basic example

For instance, one can write a makefile for computing the factorial of an integer in the following way (the main function, asking for the value of n, is defined in test.c):

test: test.c fact.c fact.h
gcc -o test test.c fact.c
Running the command $ make test will create the executable test by the command

$ gcc -o test test.c fact.c

If one does not provide any parameter to the command make, the first target file of the makefile will be used. But this does not use completely the possibilities of make. Indeed, such direct compilation corresponds to 3 different steps (actually more, see section 3.1.5):

1. the compilation of the source file fact.c, creating an object file fact.o,
2. the compilation of the source file test.c, creating an object file test.o,
3. the link editions between the two previous object files, creating the executable.

That leads to the following makefile:

test: test.o fact.o
gcc -o test test.o fact.o

test.o: test.c fact.h
gcc -o test.o -c test.c

fact.o: fact.c
gcc -o fact.o -c fact.c

If one run the make command for the first time, we get the following:

$ make
gcc -o test.o -c test.c
gcc -o fact.o -c fact.c
gcc -o test test.o fact.o

If we modify the file fact.c, the file test.o does not need to be modified. Therefore, only two step will be executed by make:

$ touch fact.c
$ make
gcc -o fact.o -c fact.c
gcc -o test test.o fact.o

Remark 2.2. There is an easy way to determine the dependencies of the different files we are using: the option -MM of gcc. For instance, one get:

$ gcc -MM fact.c test.c
fact.o: fact.c
test.o: test.c fact.h

At this point, the makefile does not manage everything:

- one cannot generate several executables at once,
- Intermediary files created are not removed,
- we cannot force a full recompilation of the program (for this point however, note the option -B of make).

We usually add the following rules:
• all (usually the first one of the file), that group all the executable / every final object to be built,
• clean that removes all intermediary files,
• fclean that removes everything that can be generated.

For instance, on our example, one would get:

```
all: test

test: test.o fact.o
   gcc -o test test.o fact.o

test.o: test.c fact.h
   gcc -o test.o -c test.c

fact.o: fact.c
   gcc -o fact.o -c fact.c

.PHONY: clean fclean all

clean:
   rm *.o

fclean: clean
   rm test
```

Note that the .PHONY tells to the make command that the targets clean, fclean and all are not files to be built.

### 2.3 Macro and abbreviations

To make the writing of a makefile easier, one can use some macro in the following way:

```
macro_name = what we want
```

When we use the command make, all the instance as $(macro_name) in the makefile will be replace with what we want.

The common macro one uses are the following:

- CC for the name of the compiler we use,
- CFLAGS for the options of compilation,
- LDFLAGS for the options of the link edition step,
- CPPFLAGS provides options for the preprocessor,
- FFLAGS gives the extra flags to the Fortran compiler...

There are actually a lot of implicit variables used by make. You can find a partial list here: [http://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html](http://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html)

These macro enable to modify the makefile easily, by just changing the macros; for instance, if one want to change the compiler in the makefile below, the only line to change is the first one. One can also define any other macro ; for instance, in the example below, I use a macro EXEC to list the set of executables I want to compile ; this is mainly so that if this set changes, I do not need to change it twice in the makefile (targets all and fclean):
CC = gcc
RM = rm -f
CFLAGS = -ansi -Wall -pedantic -c
LDFLAGS =
EXEC = test

.PHONY: clean fclean all

all: $(EXEC)

test: test.o fact.o
   $(CC) -o $@ test.o fact.o $(LDFLAGS)

test.o: test.c fact.h
   $(CC) -o test.o test.c $(CFLAGS)

fact.o: fact.c
   $(CC) -o fact.o fact.c $(CFLAGS)

clean:
   $(RM) *.o

clean: clean
   $(RM) $(EXEC)

There exist a few macro already defined:

- $@ stands for the name of the target,
- $< gives the name of the first dependency,
- $^ is the list of all dependencies,
- $* designs the target file without any suffix.

One thus may write our makefile as:

CC = gcc
RM = rm -f
CFLAGS = -ansi -Wall -pedantic -c
LDFLAGS =
EXEC = test

.PHONY: clean fclean all

all: $(EXEC)

test: test.o fact.o
   $(CC) -o $@ $^ $(LDFLAGS)

test.o: test.c fact.h
   $(CC) -o $@ $< $(CFLAGS)

fact.o: fact.c
$(CC) -o $@ $< $(CFLAGS)

clean:
 $(RM) *.o

fclean: clean
 $(RM) $(EXEC)

Finally, one can create generic rules to manage a given suffix. This is done in the following way (here we also define rules to create debug files):

% .o: %.c
  commands

This gives:

CC = gcc
RM = rm -f
CFLAGS = -ansi -Wall -pedantic -c
LDFLAGS =
DEBUGFLAGS = $(CFLAGS) -g
EXEC = test.out
EXECDB = test.db

.PHONY: clean fclean all exec debug

exec: $(EXEC)

debug: $(EXECDB)

all: exec debug

%.out: %.o
  $(CC) -o $@ $^ $(LDFLAGS)

%.db: %.do
  $(CC) -o $@ $^ $(LDFLAGS)

% .o: %.c
  $(CC) -o $@ $(CFLAGS) -c $<

% .do: %.c
  $(CC) -o $@ $(DEBUGLAGS) -c $<

test.out: test.o fact.o

test.o: test.c fact.h
fact.o: fact.c

test.db: test.do fact.do

test.do: test.c fact.h
fact.do: fact.c
clean:
    $(RM) *.o *.do

fclean: clean
    $(RM) $(EXEC) $(EXECDB)

Here the .do extension stands for the object files with information for the debugger, and .db for the files to use gdb
3 C language

Contents

3.1 What's C ? ........................................ 35
   3.1.1 ANSI C: a very small and simple language .......... 35
   3.1.2 A low-level programming language .................. 36
   3.1.3 A first example ................................ 36
   3.1.4 Warning ...................................... 36
   3.1.5 Principle of compilation ........................... 37
3.2 More details ...................................... 38
   3.2.1 Operation priorities ............................... 38
   3.2.2 Identifiers .................................... 38
   3.2.3 Control structures ............................... 39
3.3 Preprocessor ...................................... 40
3.4 Structures ........................................ 41
   3.4.1 Table ........................................ 41
   3.4.2 Struct ........................................ 42
   3.4.3 Enumeration ................................... 43
   3.4.4 union ......................................... 43
3.5 Pointers .......................................... 43
   3.5.1 What is a pointer ? ............................... 43
   3.5.2 Pointer arithmetic ............................... 45
   3.5.3 Dynamic allocation ............................... 46
   3.5.4 Pointers and tables ............................. 48
   3.5.5 Pointers and structures ......................... 48
   3.5.6 Linked list ................................... 49
3.6 Function and variables ............................ 50
   3.6.1 Variables ..................................... 50
   3.6.2 the main function ............................... 51
   3.6.3 Pointers on function ............................. 52

3.1 What’s C ?

3.1.1 ANSI C: a very small and simple language

- 32 key words:
  - char
doubleenum
floatint
longshortsigned
structunionunsigned
void
continue
defaultdo
elseforgoto
if

switch
while
auto
register
static
extern
typedef
return
sizeof
That’s all!

3.1.2 A low-level programming language

C language

• enables to manage data at a processor level,
• does not manage memory,
• does not have any instruction to manage character chains, structures...
• does not have any Input/Output function

To manage these points, we need libraries.

3.1.3 A first example

```c
/* This is a comment */
#include <stdio.h>

int main(int argc, char**argv)
{
    printf("Hello world\n");
    return 0; /* i.e. EXIT_SUCCESS */
}
```

• `include` is a directive for the preprocessor to use what is needed for the function `printf` of the standard library;

• Instructions end by a semi-colon;

• The function `main` is needed to produce an executable (that will first run `main`). It is defined by the header of the function: return type, name, argument; the braces contain the instructions of the function;

• The function `printf` is declared in `stdio.h`.

3.1.4 Warning

In C, the coder is suppose to know what he is doing. Error test and type checking are only made during the compilation phase; this means that nothing of the sort is tested during the execution of the program (are we using an index that is bigger than the size of a table, is the conversion of type used ok...). Moreover, the compiler is lazy: if something has a tiny sense, it will be allowed; therefore, a program can run correctly for some data and give an error for other ones.
3.1.5 Principle of compilation

1. **Editing the source file:** text file containing the program; requires a text editor (vi, emacs...).

2. **Preprocessing phase:** the source file is processed by a *preprocessor* that makes only purely text transformations (replacing character chains, including other source file...).

3. **Compilation:** The file generated by the preprocessor is converted in *assembly language*, i.e. a sequence of instructions associated to the functionality of the microprocessor (make an addition...).

4. **Assembler:** transforms the assembly language in an *object* file understandable for the processor.

5. **Link edition:** In order to use function libraries already written, a program is separated in several source files. Once the source code is assembled, one must *link* the object files together. This creates an executable.

Usually, we suffix the files in the following way:

- .c for the source files,
- .i for the files created by the preprocessing phase,
- .s for the assembly language files,
- .o for the object files,
- .a for the object files that correspond to pre-compiled libraries,

The UNIX C compiler is called *cc*; we will rather use the one from the GNU project, *gcc*. A basic use is:

```
$ gcc [options] file.c [-l]libraries
```

If no option gives a name for it, the output will be an executable file `a.out`. If one gives libraries (let say `-llibrary`), they will be searched (precisely here, the file `liblibrary.a`) in the directory containing the precompiled libraries - usually `/usr/lib`. For instance, if we want to link the program with the mathematic library, we specify the option `-lm`, that will use the corresponding `libm.a` object file.

The main options of the compiler are the following:

- `-c` does not make the link edition; this produces an object file,
- `-E` makes only the preprocessing phase,
- `-g` will give the symbolic informations used by the debugger,
- `-I(directory)` specifies a directory where to find the needed header files (the current directory is always included by default),
- `-L(directory)` specifies the directory where to find libraries,
- `-o filename` gives the name of the file to produce (default is `a.out`),
- `-O, -O1, -O2, -O3` gives the level of optimization required; without such option, the aim of the compiler is to reduce the compilation cost. With this option, will try to reduce the size and the executive time of the produced file. The number specifies the level of optimization,
- `-S` runs only the preprocessor and the compiler; therefore, this produces an assembly language file,
• \(-v\) prints the executed commands during the compilation,
• \(-W\) prints more warnings,
• \(-Wall\) prints all warnings.

3.2 More details

3.2.1 Operation priorities

| 16 | ( ) [ ] -> . | L |
| 15 | ++ -- (prefix) | R |
| 14 | ! ~ ++ -- (prefix) - (unary) (type) | R |
|   | * (indirection) & (address) sizeof | R |
| 13 | (multiplication) / % | L |
| 12 | + - | L |
| 11 | << >> | L |
| 10 | < <= > >= | L |
|  9 | == != | L |
|  8 | & (bitwise and) | L |
|  7 | ^ | L |
|  6 | | | L |
|  5 | && | L |
|  4 | || | L |
|  3 | ?: | R |
|  2 | = += -= *= /= %= >>= <<= &= ^= |= | R |
|  1 | , | L |

You can use this table in exercise 3.1

3.2.2 Identifiers

These are memory containing data (variables...) or code to execute (function). One should choose “goods” names for it. There are some mandatory rules:

• an identifier must not begin with an integer,
• it must not contain an operator,
• it cannot be a key word,
• small and capital letters are different,

There are also some conventions one should follow:

• non ASCII characters should not be used (portability),
• identifiers beginning by a _ are supposed to be only for the OS programs
• identifiers should be easily understandable : usually, one use short names for local variable, more complete one for global variables (surname_table for instance), and capital letters are kept for constant values.

Example 3.1.

• 5, foo-1, 3knight, foo. are not correct,
• _foo_bar, tête, ___A_ _ are correct but should not be used,
• a is perfectly fine (usually for a local variable).
3.2.3 Control structures

**Conditional instruction.** The syntax is the following:

```c
if (expression)
    instruction1;

or

if (expression)
    instruction1;
else
    instruction2;
```

This evaluates the expression. If the value is different from 0, instruction1 is executed; otherwise, it is instruction2 (if it exists).

**Multiple choice instruction.** The syntax is the following:

```c
switch (expression)
    case cte1:
        instruction1;
        break;
    case cte2:
        instruction1;
        break;
    .
    .
    case cteN:
        instructionN;
        break;
    default:
        instruction;
```

This evaluates expression, compares it with cte1, cte2,... If expression is equal to cteK, instructionK is executed; the break instruction stops the execution of the switch, so that the next constants are not checked (this is a common but optional instruction). If none of the constant is equal to expression then instruction is executed (if it exists; the default part is optional).

**Remark:** the constants must be different!

**Condition-controlled loop** There exist three kinds of iterative instructions:

- **The instruction** while:
  ```c
  while (expression)
      instruction;
  ```

- **One can also use the do** while **instruction, in the following way:**
  ```c
  do
      instruction;
  while (expression);
  ```
Here instruction is executed at least once.

• Finally, there is the for loop:

```c
for (expression1; expression2; expression3)
  instruction;
```

which is equivalent to

```c
expression1;
while (expression2)
{
  instruction;
  expression3;
}
```

**Controlling loops iterations** The break and continue enable to respectively terminate a loop body and skip the current loop instruction. More precisely, the break instruction stops the first for, while, do or switch body structure. On the other hand, the continue instruction stops the current iteration of the current for, while or do loop, and starts the next one.

### 3.3 Preprocessor

• inclusion (# include <file.h>, # include "file.h"),

• text substitution: # define A 12, # undef A...
  
  Macro with parameters:

  ```c
  # define max(a,b) a>b?a:b
  
  max(3,4)
  ```

  after the preprocessing phase (gcc -E), we have

  ```c
  3>4?3:4
  ```

  (that will be evaluated as 4).

  **Warning!**

  ```c
  # define SQR(x) x*x
  
  SQR(a+1)
  ```

  will compute a+1*a+1, thus 2a + 1 ≠ a^2 + 2a + 1.
  One can have some really weird results:

  ```c
  # define max(a,b) a>b?a:b
  
  max(x=y, ++z)
  ```

  after the preprocessing phase (gcc -E), we have
\[ x = y > + + z \? x = y : + + z; \]

which is the same than

\[ x = ( y > + + z \? x = y : + + z) \]

different than

\[ (x = y) > + + z \? x = y : + + z \]

**Conclusion:** Put brackets around variables (\# define \texttt{max(a,b)} \texttt{(a)>(b)?(a):(b)})

- condition structures:
  - \# if \# endif
  - \# if \# else \# endif
  - \# ifdef \# endif
  - \# ifdef \# else \# endif
  - \# ifndef \# endif

Two examples:

\[
\begin{align*}
\# \text{ifdef} \ ERR \\
\# \text{define} \ SQR(x) \ x*x \\
\# \text{else} \\
\# \text{define} \ SQR(x) \ ((x)*(x)) \\
\# \text{endif}
\end{align*}
\]

One can define a macro while computing: `gcc -D ERREUR file.c`

\[
\begin{align*}
\# \text{ifndef} \ MYMACRO \\
\# \text{error} \ "MYMACRO \text{ unknown}" \\
\# \text{endif}
\end{align*}
\]

(# error stops the compilation.)

### 3.4 Structures

#### 3.4.1 Table

This code:

\[
\begin{align*}
\# \text{include} <stdio.h> \\
\# \text{define} \ N \ 12 \\
\# \text{define} \ M \ 15
\end{align*}
\]

```c
void mytabprint(int tab[], int size) {
    int i;
    for (i=0; i<size; i++)
        printf("%d ",tab[i]);
    putchar('\\n');
}
```
int main()
{
    int tab[N]={24, 32, 2, 4, 8, 16};
    mytabprint(tab,N);
    int tab2[M];
    mytabprint(tab2,M);
    return 0;
}

will print:

./a.out
24 32 2 4 8 16 0 0 0 0 0 0
-1075251788 134513128 -1075251800 -1215882668 0 -1216005304 1 0 1
-1215883016 -1216094220 -1216351431 -1217219675 -1075251880 -1217320331

3.4.2 Struct

struct complex {
    float re;
    float im;
};

struct complex alpha;
alpha.re=4.5;
alpha.im=0.2;

One can initialize a variable directly:

struct adress{
    int num;
    char street[40];
    char code[8];
    char city[20];
};

struct people{
    char name[20];
    char first_name[25];
    int age;
    struct adress adr;
};

struct people john_doe = {"Doe", "John", 45, {39, "London street", "N2J 1B9", "Waterloo"}};

printf("%s\n", john_doe.adr.code);

will print N2J 1B9.
One can create a "new type" (it is more a new name for another type):
typedef type synonym

For instance:

```c
struct complex {
    float re;
    float im;
};

typedef struct complex complexe

complexe alpha;
alpha.re=4.5;
alpha.im=0.2;

This also works:

typedef struct complex {
    float re;
    float im;
} complex alpha = {4.5, 1.2};

3.4.3 Enumeration

denum color_e {CLUB, DIAMOND, HEART, SPADE, TRUMP, EXCUSE};

We can create an anonymous enumeration:

denum {JACK=11, KNIGHT, QUEEN, KING};

Here KNIGHT is equal to 12, QUEEN to 13...

typedef enum colors_e { RED, GREEN, BLUE = 5, YELLOW } colors paint_color;

Here the declaration is done directly. RED is equal to 0, GREEN to 1, BLUE to 5 and YELLOW to 6.

3.4.4 union

This enables to store in a same union different types: an object can be of one of the types defined in the structure union.

union day {
    char letter;
    int number;
};

3.5 Pointers

3.5.1 What is a pointer ?

int i, j;
i=3;
j=i;
If the compiler put the variable \( i \) at the address 343211400 and the variable \( j \) at the address 343211404, we have:

<table>
<thead>
<tr>
<th>object</th>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td>343211400</td>
<td>3</td>
</tr>
<tr>
<td>( j )</td>
<td>343211404</td>
<td>3</td>
</tr>
</tbody>
</table>

\[
\text{int } i=3; \\
\text{int } *p; \\
p=&i;
\]

In this other program, the situation is:

<table>
<thead>
<tr>
<th>object</th>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td>343211400</td>
<td>3</td>
</tr>
<tr>
<td>( p )</td>
<td>343211404</td>
<td>343211400</td>
</tr>
</tbody>
</table>

```c
#include <stdio.h>

int main()
{
    int i=3; 
    int *p; 
    p=&i; 
    printf("\*p = %d\n", *p); 
    return 0; 
}
```

will print \( \ast p = 3 \). Here we have:

<table>
<thead>
<tr>
<th>object</th>
<th>address</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td>343211400</td>
<td>3</td>
</tr>
<tr>
<td>( p )</td>
<td>343211404</td>
<td>343211400</td>
</tr>
<tr>
<td>( \ast p )</td>
<td>343211400</td>
<td>3</td>
</tr>
</tbody>
</table>

One can note that \( i \) and \( \ast p \) are identical: same adress and same value. If you change \( \ast p \), you also change \( i \)!

Let consider the situation of the two next programs:

```c
int main()
{
    int i=3, j=6; 
    int *p1, *p2; 
    p1=&i; 
    p2=&j; 
    *p1=*p2; 
    return 0; 
}
```

```c
int main()
{
    int i=3, j=6; 
    int *p1, *p2; 
    p1=&i; 
```
Before the last affectation, for both programs, the variables are:

<table>
<thead>
<tr>
<th>object</th>
<th>adress</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>343211400</td>
<td>3</td>
</tr>
<tr>
<td>j</td>
<td>343211404</td>
<td>6</td>
</tr>
<tr>
<td>p1</td>
<td>343211984</td>
<td>343211400</td>
</tr>
<tr>
<td>p2</td>
<td>343212000</td>
<td>343211404</td>
</tr>
</tbody>
</table>

After the affectation *p1=*p2; of the first program, we have:

<table>
<thead>
<tr>
<th>object</th>
<th>adress</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>343211400</td>
<td>6</td>
</tr>
<tr>
<td>j</td>
<td>343211404</td>
<td>6</td>
</tr>
<tr>
<td>p1</td>
<td>343211984</td>
<td>343211400</td>
</tr>
<tr>
<td>p2</td>
<td>343212000</td>
<td>343211404</td>
</tr>
</tbody>
</table>

whereas after the affectation p1=p2; of the second program, we have:

<table>
<thead>
<tr>
<th>object</th>
<th>adress</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>343211400</td>
<td>3</td>
</tr>
<tr>
<td>j</td>
<td>343211404</td>
<td>6</td>
</tr>
<tr>
<td>p1</td>
<td>343211984</td>
<td>343211404</td>
</tr>
<tr>
<td>p2</td>
<td>343212000</td>
<td>343211404</td>
</tr>
</tbody>
</table>

### 3.5.2 Pointer arithmetic

A pointer is an integer, therefore, one can make some operations on them: one may

- add an integer to a pointer; the result is a pointer with the same type as the original pointer.
- subtract an integer to a pointer; the result is a pointer with the same type as the original pointer.
- make the difference between two pointers with the same type. The result is an integer.

(one cannot add two pointers).

Of course, one can use the increment `++` and decrement `--` operations.

For instance,

```c
#include <stdio.h>

int main()
{
    int i;
    int *p1, *p2;
    p1=&i;
    p2=p1+1;
    printf("p1 = %ld \t p2 = %ld\n", p1, p2);
    return 0;
}
```

45
will print \( p_1 = 354321840 \) \( p_2 = 354321844 \).

But if the pointers point on double:

```c
#include <stdio.h>

int main()
{
    double i;
    double *p1, *p2;
    p1=&i;
    p2=p1+1;
    printf("p1 = %ld \t p2 = %ld\n", p1, p2);
    return 0;
}
```

will print \( p_1 = 354321800 \) \( p_2 = 354321808 \).

In particular, pointers are useful when dealing with tables:

```c
#include <stdio.h>
#define N 5
int tab[N]={3,1,7,9,4};
int main()
{
    int *p;
    printf("Increasing order:\n");
    for (p=&tab[0]; p <= &tab[N-1]; p++
        printf("%d ", *p);
    printf("\nDecreasing order:\n");
    for (p=&tab[N-1]; p >= &tab[0]; p--
        printf("%d ", *p);
    return 0;
}
```

3.5.3 Dynamic allocation

When we want to use a pointer, specially when we want to use the indirection operator *, we have to initialize it. One can affect to the pointer the address of another variable. One can also directly give a value to *p, but in such case we have to reserve a memory space of the correct size. The address of this memory space will be the value of p. This is called dynamic allocation, and is with the malloc function (from stdlib.h):

```c
malloc(number_of_octets)
```

This function returns a pointer of type char * that points to an object of size number_of_octets. If we want to use something else than an object of type char, we can do the following:

```c
#include <stdlib.h>
int *p;
p=(int*) malloc(sizeof(int));
```

Remark 3.1. To any malloc call should correspond a free call later on, in order to avoid memory leaks. In particular, you might want to check your programs involving dynamic allocation with the valgrind command.
In the following example:

```c
#include <stdlib.h>
int main()
{
    int i=3;
    int *p;
    p=(int*) malloc(sizeof(int));
    *p=i;
    return 0;
}
```

before the dynamic allocation, we have:

<table>
<thead>
<tr>
<th>object</th>
<th>adress</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>343211400</td>
<td>3</td>
</tr>
<tr>
<td>p</td>
<td>343211404</td>
<td>0</td>
</tr>
</tbody>
</table>

Here, *p has no sense; if we try to use the variable *p, one will get a segmentation fault. The dynamic allocation gives a value to p, and reserve at this adress a memory space equal to 4 octets. After that, we have:

<table>
<thead>
<tr>
<th>object</th>
<th>adress</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>343211400</td>
<td>3</td>
</tr>
<tr>
<td>p</td>
<td>343211404</td>
<td>546318724</td>
</tr>
<tr>
<td>*p</td>
<td>546318724</td>
<td>(int)</td>
</tr>
</tbody>
</table>

At this stage, *p is defined but not initialized. Its value can be any integer (the last one stored at this adress). At the end of the program, we thus have:

<table>
<thead>
<tr>
<th>object</th>
<th>adress</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>343211400</td>
<td>3</td>
</tr>
<tr>
<td>p</td>
<td>343211404</td>
<td>546318724</td>
</tr>
<tr>
<td>*p</td>
<td>546318724</td>
<td>3</td>
</tr>
</tbody>
</table>

Note that the following program is different:

```c
main()
{
    int i=3;
    int *p;
    p=&i;
}
```

Here we have:

<table>
<thead>
<tr>
<th>object</th>
<th>adress</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>343211400</td>
<td>3</td>
</tr>
<tr>
<td>p</td>
<td>343211404</td>
<td>343211400</td>
</tr>
<tr>
<td>*p</td>
<td>343211400</td>
<td>3</td>
</tr>
</tbody>
</table>

Here i and *p are the same variable: any modification on one of them modifies the second. This is not the case in the previous example.
3.5.4 Pointers and tables

In C, a table is a constant pointer: if one has

```c
int tab[10];
```

tab is a constant pointer, with for value the address of the first element of the table. One can access to the element of index `i` of `tab` with `tab[i]`, or also `*(tab+i)` (these two ones are equivalent).

One can create a table with `n` elements where `n` is a variable of the program as follows:

```c
#include <stdlib.h>
main()
{
    int n;
    int *tab;
    ...
    tab=(int *)malloc(n*sizeof(int));
    ...
    free(tab);
}
```

If we use the function `tab = (int*)calloc(n,sizeof(int));` instead of `malloc`, all the elements of `tab` are also initialized to 0.

One always have to allocate the memory of such table, and if we do not want any memory problem later on, one has to free the memory once used.

One can create a matrix with `k` lines and `n` columns as follows:

```c
#include <stdlib.h>
main()
{
    int k, n,i;
    int **tab;
    ...
    tab=(int **)malloc(k*sizeof(int*));
    for (i=0; i<k ; i++)
        tab[i]=(int *)malloc(n*sizeof(int));
    ...
    for (i=0; i<k ; i++)
        free(tab[i]);
    free(tab);
}
```

It is also possible to have a different number of elements for each line.

```c
for (i=0; i<k ; i++)
    tab[i]=(int *)malloc((i+1)*sizeof(int));
```

3.5.5 Pointers and structures

In C, structures have an address. Therefore, if we want to modify one given as an input of a function, it is better to use a pointer on this structure. Also, if the structure takes a lot of memory space, it is better to use a pointer to this structure than the structure itself when we use it as a parameter of an intermediary function: as copies of the parameter are made when the function is called, it is better to copy a pointer (4 octets) than the structure (specially if it contains for instance a big table).

If `p` is a pointer on a structure, one can access to a member of this structure by the following expression:
The following line is equivalent:

\[ p->\text{member} \]

For instance, one could replace \( \text{tab[i].date} \) by \( \text{tab[i]}->\text{date+} \).

### 3.5.6 Linked list

It is very often useful to create a structure with one member that is a pointer to a structure with the same model, as for instance:

```c
struct cell
{
    int value;
    struct cell *next;
};
```

This enables to create a linked list as a pointer that points to such a structure:

```c
typedef struct cell *list;
```

With such a list, it is easy to insert an element:

```c
list insert(int elt, list l)
{
    list l2;
    l2=(list) malloc(sizeof(struct cell));
    l2->value=elt;
    l2->next=l;
    return l;
}
```

One example: the following program create a list of integers

```c
#include <stdlib.h>
#include <stdio.h>

struct cell
{
    int value;
    struct cell *next;
};

typedef struct cell *list;

list insert(int elt, list l)
{
    list l2;
    l2=(list) malloc(sizeof(struct cell));
    l2->value=elt;
    l2->next=l;
    return l;
}
```
main()
{
  list l, p;
  l = insert(1, insert(2, insert(3, insert(4, NULL))));
  printf("\n The list is:\n");
  p = l;
  while (p != NULL)
  {
    printf("%d \t", p->value);
    p = p->next;
  }
}

A last example of structure: one can manage a binary tree in the following way:

struct node
{
    int value;
    struct node *left;
    struct node *right;
};

typedef struct node *tree;

3.6 Function and variables

3.6.1 Variables

• **Local variables:** they are defined inside a function, and are destroyed at the end of the function call.

• **Global variables:** these are the variables declared outside of any function. One make declaration in headers files, but we define them only once, in a .c file.

• **Static variables:** they take a place in the memory that will never change during all the program. The memory allocated is part of the data memory. These variables are initialized at 0 by default. This is the only way to keep a local variable from being lost at the end of a function call.

• **Constant variables:** a variable defined with const cannot be modified. One use it quite often for character chain.
  - `const char *p` is a pointer on a constant character.
  - `char * const p` is a constant pointer on a character.

• **volatile variable:** such variable cannot be modified by any optimisation of the compiler. This is used whenever we consider a variable that may be modified outside the program.

Some examples:

```c
int n = 10, k=20;
void fonction();
void fonction()
{```
int n = 0;
n++;
printf("call %d\n",n);
printf("global %d\n",k);
return;
}

main(){
    int i;
    for (i = 0; i < 3; i++)
        fonction();
    printf("outside the function %d\n",n);
}

will print

call 1
global 20
call 1
global 20
call 1
global 20
outside the function 10

#include <stdio.h>

int n = 10;
void fonction();
void fonction(){
    static int n;
    n++;
    printf("call %d\n",n);
    return;
}

main(){
    int i;
    for (i = 0; i < 5; i++)
        fonction();
    printf("outside the function %d\n",n);
}

will print

call 1
call 2
call 3
call 4
call 5
outside the function 10

3.6.2 the main function

The structure of the main function is the following:
int main(int argc, char* argv[]);

which can also be written as

int main(int argc, char** argv);

- `argc` is the number of arguments given as an input to the executable;
- `argv` is a table of pointers. `argv[0]` will be the name of the executable, `argv[1]` the first parameter given to the executive, `argv[2]` the second...

One example:

```c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    int a, b;
    if (argc != 3)
        { 
            printf("\nError : bad number of arguments");
            printf("\nUsage: %s int int\n",argv[0]);
            exit(EXIT_FAILURE);
        } 
    a = atoi(argv[1]);
    b = atoi(argv[2]);
    printf("\nThe product of %d and %d is : %d\n", a, b, a * b);
    exit(EXIT_SUCCESS);
}
```

After compilation, if we run

$ a.out 9 4

we will have the result:

The product of 9 and 4 is : 36

Here, the `atoi` function, from the standard library, converts a character chain into an integer.

### 3.6.3 Pointers on function

```c
int operation(int, int, int(*)(int, int));

int operation(int a, int b, int (*f)(int, int)){
    return((*f)(a,b));
}
```
First exercises in C

Contents

Exercise 3.1: Priority of operations ........................................... 53
Exercise 3.2: Introduction to the C language ............................... 53
Exercise 3.3: first programs in C ............................................... 55
Exercise 3.4: around floating point numbers ............................... 57
Exercise 3.5: the gdb debugger ................................................. 58
Exercise 3.6: around the preprocessor ...................................... 59

Exercise 3.1. [Priority of operations] Assume that we have:

\[ A = 20 \quad B = 5 \quad C = -10 \quad D = 2 \quad X = 12 \quad Y = 15 \]

Evaluate the valid expressions in the following expressions:

\[ 5 \times X + 2 \times 3 \times B/4 \quad A == B = 3 \quad A+ = X + 2 \]
\[ A! = C* = -D \quad A% = D + + \quad A% = + + D \]
\[ (X + +) * A + C \quad A - A == B - B \quad !(X - D + C) || D \]
\[ A&\&B||0 \quad C = 12 -- \quad (1 << (2 << 1)) == ((1 << 2) << 1) \]

If an expression is not valid, add some brackets so that it becomes valid.

Exercise 3.2. [Introduction to the C language]

1. Editing a file.

Emacs provides a c mode that makes easier the writing of c code (files .c or .h). In particular, it provides the following commands (C- designs the <Ctrl> button, M- the <Meta> one, that is either <ESC>, <Echap> or <Alt>):

- TAB puts the current line with the correct indentation,
- M-; creates a commentary,
- M-x compile starts the compilation in the current directory (by default, calling make),
- C-x ` goes to the next compilation error,
- M-x gdb runs the debugger (gdb by default),

Try these commands all along this practical work!

2. Preprocessor. In the following, we will look at the several steps of the compilation process on the following file:

```c
/* ------------------------------
A first program
-------------------------------- */

#include <stdio.h>
#include <stdlib.h>
```

53
#define MAX 12

int foo = 3;
int tom = 4;
const int bar = 5;

extern int gee;

int main()
{
    int bzu; /* declaration and definition of bzu */
    int biz=3,buz;
    const int cte = 0;
    static raf = 3;
    bzu = MAX;
    printf("Hello : %d\n", foo + bar + gee + bzu);

    exit(EXIT_SUCCESS);
}

• Save this code in a file phases.c
• Check the option -E of gcc in its manual.
• Create a file phases.i containing the result of the preprocessor on the program phases.c.
• Compare the two files:
  – How many lines do they contain ?
  – What did the comments become ?
  – What became the line #include <stdio.h> ?
  – What happened for the line bzu = MAX; ? You can check that with the Macro expand region command (C-c C-e) of the C mode of emacs.

3. Generating the assembly code
• Check the option -S of gcc.
• Use it to generate the assembly code associated to phases.c,
• You can have a look at it ; try to identify the different sections .data, .rodata, and .text. You can add some declarations to phases.c or modify it to see the differences in the produced assembly codes...

4. Object file generation
• How do we generate the object file associated to phases.c ?
• Compare the result of the nm command on the object file with the section you identified in the assembly code.
• What does nm says about the symbols printf, exit, and gee ?
• Where are defined these symbols ?
• Which program is supposed to look for these symbols ?
• What changes in the result of the nm command if one suppress the use of gee in the function call printf of phases.c ?
• How do you interpret the result of the `strings` command on the object file?

5. A first compilation and execution.

We want to create a program that compute factorial 10 and prints the result. To that purpose, we will use the following function, that computes factorial n in an iterative way (copy it in a file `fact.c`)

```c
unsigned int
fact (unsigned int n)
{
    unsigned int i = 1, res = 1;

    while (i < n)
        res = res * i++;
    return res;
}
```

• Compile this source file with the following command:

```bash
$ gcc -g -Wall -ansi -pedantic -o fact fact.c
```

- We use the gcc compiler of the GNU project,
- Options `-ansi` and `-pedantic` precise that we want strictly ANSI C programs. Option `-Wall` makes gcc print every warning message. You will always use these 3 options for your compilation in this lecture.
- Option `-g` creates precious informations for the debugger. In case you are considering the speed of your program, it is better to avoid it (once everything works well), since it might slow down a little your program.
- Option `-o` enables us to name the executable file (here `fact`, default `a.out`).

If there is no error, the compiler prints nothing and generates the executable `fact. Otherwise, it prints errors and/or warnings (in case of errors, no executabale is created).

• Here you will have a couple of errors. One operator is missing, and there is no `main` function. Correct the program, compile and run it (the result should be 362880).

**Exercise 3.3.** [first programs]

• Write a program that read (`scanf`) two integers and print their gcd.

```c
Correction:

#include <stdio.h>

main()
{
    int a, b, r;
    printf("Give an integer ");
    scanf("%d", &a);
    printf("Give another one ");
    scanf("%d", &b);
    while ((r = (a % b)) != 0)
    {
        a = b;
        // Correct: Replace with b = a % b
    }
    printf("gcd: %d\n", r);
}
```

55
• The hamming weight of a number is the number of non zero bytes its binary form contains.

1. Write a program that computes the Hamming weight of an integer (type int) in a number of operations proportional to the size of the input.

   **Correction:**
   ```c
   # include <stdio.h>

   main()
   {
   int n,s,weight = 0;
   printf("Write an integer ");
   scanf("%d",&n);
   for (s=n; s > 0; s>>=1)
      weight+=y&1;
   printf("The Hamming weight of %d is %d\n",n,weight);
   }
   ```

2. How to get a number of operations proportional to the Hamming weight of the input ?

   **Correction:**
   ```c
   # include <stdio.h>

   main()
   {
   int n,s,weight = 0;
   printf("Write an integer ");
   scanf("%d",&n);
   s=n;
   while (n != 0)
   {
      weight++;
      n = n & ( n - 1 );
   }
   printf("The Hamming weight of %d is %d\n",s,weight);
   }
   ```

• What does the following code ?

   ```c
   main(){int x,p=0;scanf("%lu",&x);for(;x>0;p++){x&=x-1;printf("%d\n",p);} }
   ```

   **Correction:** This computes the Hamming weight of a given integer. Here, the stdio library is not declared, but this still compiles correctly because it is automatically checked by the preprocessor nowadays. The scanned number (%lu) is not declared as an integer but as an unsigned long, but this is also interpreted quite well. Therefore, this program works and gives the expected result. But writing it as in the previous question is definitely more than welcomed.

• Write a program that, given a value $x$ with type double, computes the evaluation of the polynomial $P(X) = a_nX^n + \cdots + a_1X + a_0$ at $X = x$. The degree $n$, the coefficients $a_i$ and $x$ will
be inputed by the keyboard. To compute the evaluation of the polynomial, we will use the Horner scheme (we first compute $a_n x + a_{n-1}$, then $a_n x^2 + a_{n-1} x + a_{n-2} \ldots$).

**Correction:**

```c
#include <stdio.h>

int main()
{
    int n, i;
    float res, a, x;

    printf("Degree of the polynomial ? ");
    scanf("%d", &n);
    printf("Value of x ? ");
    scanf("%f", &x);
    for( i = n ; i >= 0 ; i-- )
    {
        printf("Give the coefficient of index %d\n",i);
        scanf("%f", &a);
        if (i==n)
            res = a;
        else
            res=res*x+a;
    }
    printf("The value of the polynomial in x=%f is \%f\n",x,res);
}
```

**Exercise 3.4. [Floating point numbers]**

- Without using any computer: does the following program compile? If yes, do you think that the executable will work correctly? To do what?

```c
int main()
{
    float x=1;
    while (x!=x+1)
    {
        x=x+1;
    return 0;
}
```

- You can try the following one on your computer to check if your intuition was good:

```c
#include <stdio.h>
#define CTE 3000000.134513
int main()
{
    float x=CTE;
    printf("%f\n%f\n",x,x+1);
    while (x != (x+1))
    {
        x=x+CTE;
    }
```
Correction: This is indeed an algorithm: the while loop will end, because at some point, the value of \( x \) will be big enough so that \( x = x + 1 \).

Exercise 3.5. [The GDB debugger] We consider the following code:

```c
#define len 10

int main ()
{
    int tab[len];
    unsigned int i;

    for (i=len-1; i>=0; i--)
        tab[i]=i;
    return 0;
}
```

1. By reading it, what do you think this code does? Compile it and run the executable. What’s happening?

2. To understand what’s going on, we will use the debugger gdb. To use it, we need to add the option `-g` of gcc when compiling. Then one runs `$ gdb executable_name`. One can also use the command `M-x gdb` of Emacs. This leads to the gdb prompt:

```
(gdb)
```

Here are some gdb commands we can use at this point:

- `(gdb) run` (shortcut `r`) runs the executable,
- `(gdb) print i` (shortcut `p`) prints the variable `i`,
- `(gdb) display i` prints the value of `i` at each step used,
- `(gdb) whatis M` gives the type of the variable `M`,
- `(gdb) call function(arguments)` prints the result of the function call for given values,
- `(gdb) break 9` (shortcut `b`) makes a break when reaching the line 9 of the program. One can add conditions to the break command. For instance:
  ```
  (gdb) break 9 if i=0 || j!=3
  ```
- `(gdb) b file.c:9` makes a break when reaching the line 9 of file.c,
- `(gdb) continue` (shortcut `c`) run the program up to the next break,
- `(gdb) next` (shortcut `n`) enables to run a program step by step after reaching a break,
- `(gdb) quit` (shortcut `q`) quits gdb.

Run gdb and follow the variable `i`. When `i` reaches 0, what happens next? Why?

Correction: By running gdb, one can see that the segmentation fault appears at the line 9 (`tab[i]=i;`). One can put a break here (`b 9`), follow the variable `i` (`display i`), and...
run the program again (r or run), and continue step by step (continue or c each time). This enables us to see that the value i goes from 0 to 4294967295. This is because i is an unsigned int.

To solve the problem, we just need to declare i as an int.

**Exercise 3.6. [Some Macros]**

1. **Define with macros two "constants" TRUE and FALSE**

   **Correction:**
   
   ```c
   # define FALSE 0
   # define TRUE !(FALSE)
   
   or
   
   # define FALSE (1==0)
   # define TRUE (1==1)
   ```

2. **Write a macro DIGORCAP(c) that test if a character given as parameter is a digit or a capital letter.**

   **Correction:**
   
   ```c
   # define ISDIG(c) ((c>='0')&&(c<='9'))
   # define ISCAP(c) ((c>='A')&&(c<='Z'))
   # define DIGORCAP(c) (ISDIG(c)||ISCAP(c))
   ```

3. **What does the following algorithm:**

   ```c
   # include <stdio.h>
   # define ISDIG(c) (((c)>='0'&&((c)<='9'))
   
   int main()
   {
       unsigned int res=0;
       int c;
       while (c=getchar(), ISDIG(c))
       {
           res*=10;
           res+=c-'0';
       }
       printf("%d",res);
       return 0;
   }
   ```

   *What is the difference with this one (you can compile it with the -E option to see more):*

   ```c
   # include <stdio.h>
   # define ISDIG(c) (((c)>='0'&&((c)<='9'))
   
   int main()
   ```
{ unsigned int res=0; int c; while (ISDIG(c=getchar()))
{ res*=10; res+=c-'0'; }
printf("%d",res);
return 0; }

Which is the “good” one?

**Correction:** The first algorithm converts an integer given as characters input on stdin (ending by a non digit character) to an int. The second one does not make the same thing because of the getchar function given as a parameter to the macro ISDIG, so that it is evaluated twice. Conclusion: do not use a function as a parameter of a macro, except if you’re sure of what you want to do (that includes incrementation).
First project: the exponential function

Send your archive by email before Monday 22, 9am

In this exercise, we will compute an approximation of the exponential function by using its truncated Taylor expansion. We recall that we have \( \exp(X) = \sum_{i=0}^{\infty} \frac{X^i}{i!} \)

1. This exercise will use several intermediary functions. Therefore, we are going to separate them and use header files. To be able to test the intermediary functions one by one, we will use a menu. During this exercise, you will modify the following code so that you can run the different functions you will create.

File menu.c:

```c
#include <stdio.h> /* for printf and scanf */
#include <stdlib.h> /* exit(), EXIT_SUCCESS, EXIT_FAILURE */

#include "fact.h"

/* one can define some constants */
#define CONSTANT 12

void menu()
{
    printf("menu :
");
    printf("1 - Compute a power of a value
");
    printf("2 - Compute a factorial
");
    printf("3 - exit
");
}

int main(void)
{
    int n,k;
    float x;
    /* one may want to add a loop here */
    menu();
    scanf("%d",&n);
    switch (n)
    {
        case 1:
        {
            printf("Value of x ? ");
            scanf("%f",&x);
            printf("Value of n ? ");
            scanf("%d",&k);
            printf("Not implemented yet");/*printf("%f",power(x,k))*/
            break;
        }
```

61
case 2:
{
    printf("Value of n ? ");
    scanf("%d",&k);
    printf("Not implemented yet");/*printf("%d",fact(k))*/
    break;
}
case 3:
{
    printf("Good bye\n");
    break;
}
default:
{
    printf("Sorry, this option does not exist\n");
    exit(EXIT_FAILURE);
}
exit(EXIT_SUCCESS);

Remember to use a makefile to manage the compilation.

2. Preliminary functions:
   a) Write a function factorial, that given an integer n, compute n!. We will use a recursive algorithm.

   **Correction:**
   ```c
   int fact(int n)
   {
       return n<=1 ? 1 : n*fact(n-1);
   }
   ```

   b) Write a function power, that given a value \(x\) and an integer \(n\), compute \(x^n\). We will use an iterative algorithm.

   **Correction:**
   ```c
   float power(float x, int n)
   {
       float res=1;
       for (;n>=1;n--)
           res*=x;
       return res;
   }
   ```

3. Write a function that, given a value \(x\) and an integer \(n\), returns the evaluation at \(x\) of the truncated Taylor series at order \(n\) of the exponential function. For instance, if one gives \(x = .12\) and \(n = 1\), this will return 1.12. For this first program, we will use the functions power and fact of the two previous questions.

   **Correction:** File exp_basic.h:
   ```c
   # ifndef EXP_BASIC_H
   ```
# define EXP_BASIC_H

extern float exp_basic(float, int);

# endif /* EXP_BASIC_H */

File exp_basic.c:

#include "fact.h"
#include "power.h"

float exp_basic(float x, int n){
  float res=1;
  int i;
  for (i=1; i<=n; i++)
    res+=power(x,i)/fact(i);
  return res;
}

a) Try it for \( x = 20 \), and for increasing order of truncations (for instance, for 12, 20, 30 and 40). What happens? Can you explain such results?

**Correction:** For the first values, one gets the correct mathematical approximation. But at some point (normally 13), the result is not good anymore; and for higher value, one gets negative values, and then inf (which means \( \infty \)), and sometimes nan (which means “not a number”). One can check that by running the factorial function for \( n = 12, 13 \) and some higher values. Same for the power function, for instance for \( x = 20 \) and \( n = 29 \) and 30.

b) Without changing the algorithm, what can be changed to solve the matter?

**Correction:** The “problem” comes from the representation of the numbers on the computer: an integer is (usually) stored on 32 bytes. As 13! is greater than \( 2^{32} \), when we try to represent the result, the first byte of 13! is lost, and we do not get the correct result. To solve that matter, one can compute the factorial as a float, or better again a long double (same for the power function).

4. We will now consider the running time of this program. To do that, one may adapt the following code to our context:

    # include <time.h>
    ...
    clock_t initial_time; /* Initial time in micro-seconds */
    clock_t final_time; /* Final time in micro-seconds */
    float cpu_time; /* Total time in seconds */
    ...
    initial_time=clock();
    fct ();
    final_time=clock();
    cpu_time=(final_time - initial_time)*1e-6;
    printf (“%f”,cpu_time);

a) Modify the menu so that one option computes an approximation of an exponential, and also gives the running time of this computation.
b) To plot the time computations, we will use the gnuplot function. This enables to display data contained in a file. Discrete data contained in a columns of a file example.dat will be used in the following way: the command

```
plot "example.dat" i j title 'example'
```

will plot the data of the i-th (as abscissa) and j-th (as ordinate) columns. Columns must be separated by a space; lines beginning by # are ignored. Create (in a separate file) a C program that computes an approximation of \( \exp(3) \) with an approximation order of \( 2^k \), \( k = 1 \cdots 14 \), and print the running time for these computation on the screen, together with the corresponding approximation order.

```
Correction:

#include <stdio.h> /* for printf and scanf */
#include <stdlib.h> /*exit(), EXIT_SUCCESS, EXIT_FAILURE*/
#include <time.h> /* to manage the running times */

#include "exp_basic.h"

clock_t initial_time; /* Initial time in micro-seconds */
clock_t final_time; /* Final time in micro-seconds */
float cpu_time; /* Total time in seconds */

int main()
{
    int i,j=1;
    long double res;
    for (i=0; i<=14; i++)
    {
        initial_time=clock();
        res=exp_basic2(3,j);
        final_time=clock();
        cpu_time=(final_time - initial_time) * 1e-6;
        printf("%d %Lf %f\n",i,res,cpu_time);
        j<<=1;
    }
    exit(EXIT_SUCCESS);
}
```

c) Store these data in a file time1.dat, and use the gnuplot function to plot them. One could use a file containing the following lines as an input to the function gnuplot:

```
set term postscript landscape
set output "time1.ps"
plot "time1.dat" using 1:2 title 'basic algorithm'
```

(The first two lines are only useful if one want to store the plotting).

5. One can see that the approximation order we can use is not very high in practice. Find a new clever algorithm to compute the Taylor series, and generate some timings for the computation of \( \exp(3) \) with an approximation order of \( 2^k \), \( k = 1 \cdots 24 \). Plot the result together with the previous running time from the naive algorithm.

```
Correction: File exp_better.h:

#include <stdio.h> /* for printf and scanf */
#include <stdlib.h> /*exit(), EXIT_SUCCESS, EXIT_FAILURE*/
#include <time.h> /* to manage the running times */

#include "exp_basic.h"

clock_t initial_time; /* Initial time in micro-seconds */
clock_t final_time; /* Final time in micro-seconds */
float cpu_time; /* Total time in seconds */

int main()
{
    int i,j=1;
    long double res;
    for (i=0; i<=24; i++)
    {
        initial_time=clock();
        res=exp_better2(3,j);
        final_time=clock();
        cpu_time=(final_time - initial_time) * 1e-6;
        printf("%d %Lf %f\n",i,res,cpu_time);
        j<<=1;
    }
    exit(EXIT_SUCCESS);
}
```
# define EXP_BETTER_H

extern long double exp_better(long double, int);

# endif /* EXP_BETTER_H */

File exp_better.c:

#include "exp_better.h"

float exp_better2(float x, int n){
    float res=1,c=x;
    int i;
    for (i=2; i<=n+1; i++)
    {
        res+=c;
        c*=(x/i);
    }
    return res;
}

File time2.c:

#include <stdio.h> /* for printf and scanf */
#include <stdlib.h> /*exit(), EXIT_SUCCESS, EXIT_FAILURE*/
#include <time.h> /* to manage the running times */

#include "exp_better.h"

clock_t initial_time; /* Initial time in micro-seconds */
clock_t final_time; /* Final time in micro-seconds */
float cpu_time; /* Total time in seconds */

int main()
{
    int i,j=1;
    long double res;
    for (i=0; i<=24; i++)
    {
        initial_time=clock();
        res=exp_better(3,j);
        final_time=clock();
        cpu_time=(final_time - initial_time)*1e-6;
        printf("%d %Lf %f\n",i,res,cpu_time);
        j<<=1;
    }
    exit(EXIT_SUCCESS);
}

Correction: You can find here:

$ wget http://www.lifl.fr/~poteaux/fichiers/teaching/exponential.tar.gz
an archive containing all the programs for this exercise, given with a makefile. You can extract it via the tar command ($ tar xzf exponential.tar.gz). Then, you can use the $ make command to see how to create the executive files, or also directly the plot files. In particular, you can have a look at the makefile to see that it is not used only to compile the c programs.
Structures and tables

Contents

| Exercise 3.7: dealing with big positive integers | 67 |
| Exercise 3.8: sorting function | 67 |
| Exercise 3.9: numerical computations | 70 |

Exercise 3.7. [Dealing with big positive integers]

1. Create a new type that represent a big positive integer (GPI) as a vector of GPI_SIZE digits which are unsigned long

   **Correction:**

   ```c
   # define GPI_SIZE 7
   typedef struct GPI_s
   { 
     unsigned long digit[GPI_SIZE];
   } GPI;
   ```

2. Write a function that add two GPI (one have to take into account the carry for the digits).

   **Correction:**

   ```c
   GPI gpi_add(GPI a, GPI b)
   {
     int i;
     int carry=0;
     GPI c;
     for (i=0; i<GPI_SIZE; i++)
     {
       c.digit[i] = a.digit[i] + b.digit[i] + carry;
       carry = (c.digit[i] > a.digit[i]) ||
          ((c.digit[i] == a.digit[i]) && (carry ==0)) ? 0 : 1;
     }
     return c;
   }
   ```

Exercise 3.8. [sorting function] Write an algorithm that sort elements of a table, which will have a constant number of values (therefore defined by something like double tab[N] where we define N in the preprocessive directives). The value of the table will be generated randomly by the function rand. It is initialized by the srand function from the number of seconds since January 1st, 1970 as:

   ```c
   srand((unsigned int) time(NULL));
   ```

To use not too big values, one should take the random numbers modulo an integer of our choice.
The algorithm should print something like:
Correction: There are a lot of sorting algorithms. Here I present two of them. At the i-th step of the algorithm, the first i elements of the table are sorted. The quicksort algorithm takes a pivot at each step and reorder the table so that all the elements greater than the pivot are after it; then, we apply the same algorithm to the two sublists. The second one is really faster in practice. You can see that by running value for high values of N (but then one should comment the tables printings).

File main.c:

```c
#include "main.h"
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "sort.h"
#define N 40 /* size of the table */
#define M 100 /* random integers will be taken in [0,M-1] */
clock_t initial_time; /* Initial time in micro-seconds */
clock_t final_time; /* Final time in micro-seconds */
float cpu_time; /* Total time in seconds */

void mytabprint(double tab[], int size)
{
    int i;
    for (i=0; i<size; i++)
        printf("%f ",tab[i]);
    putchar('\n');
}

int main()
{
    double tab1[N];
    double tab2[N];
    int i;
    /* initialization of the random function */
srand((unsigned int) time(NULL));
    /* creation of the table */
    for (i=0; i<N; i++)
    {
        tab1[i]=rand()%M;
        tab2[i]=tab1[i];
    }
    printf("Initial table\n");
    mytabprint(tab1,N);
    putchar('\n');
    ```
initial_time = clock()
insertion_sort(tab1, N);
final_time = clock();
cpu_time = (final_time - initial_time) * 1e-6;
printf("Insertion sorting\n");
mytabprint(tab1, N);
printf ("It took %f seconds to compute it\n\n", cpu_time);
initial_time = clock();
quicksort(tab2, 0, N-1);
final_time = clock();
cpu_time = (final_time - initial_time) * 1e-6;
printf("Quick sorting\n");
mytabprint(tab2, N);
printf ("It took %f seconds to compute it\n\n", cpu_time);
return 0;
)

File sort.h:

#include "sort.h"

void swap(double t[], int i, int j)
{
    double tmp = t[i];
    t[i] = t[j];
    t[j] = tmp;
}

void insertion_sort(double tab[], int size)
{
    int i, j;
    for (i = 0; i < size; i++)
        for (j = size-1; j > i; j--)
            if (tab[j] < tab[j-1])
                swap(tab, j, j-1);
}

void quicksort(double t[], int left, int right)
{
    int i, sep = left+1;
    if (left < right)
Exercise 3.9. [Numerical computations]
Create a table tab of 20000 elements containing random numbers with the following rule:

- tab[4*i] contains a float between 0 and 1112465223
- tab[4*i+1] contains a float between 0 and 5
- tab[4*i+2] contains a float between 0 and 5000
- tab[4*i+3] contains a float between 0 and 1000000

1. Add the elements of the table as a float and print the result.
2. Sort the table and make the sum again. Do you get the same result? Why?
3. Make the two same computation by using long double and interpret the result.

Correction: Here we are using the quicksort function of the last exercise. This gives the following file:

```c
#include <stdlib.h>
#include <stdio.h>

#define N 20000
#define A 5001
#define S 6
#define B 1000001
#define VB 1112465224

#include "sort.h"

int main()
{
    int i;
    float res=0;
    float tab[N];
    srand((unsigned int) time(NULL));
    for (i=0; i<N; i+=4)
    {
```
By running it, we see that we do not get the same result before or after sorting. The second result is greater. This is due to the representation of float numbers, and more precisely here how the addition of two floats numbers with different exponent parts is dealt.
Exercise 3.10. [pointers and matrices]

Let assume we have this declaration:

```c
int b[3][5];
```

If we suppose that the allocation of `b` is such that the adress of the three “slides” of `b` are successive, what is `b` after the following C code ?

```c
int b[3][5];
int *a = b, i;
for (i=0 ; i<15 ; *a++ = i++)
    ;
**b = 15;   *(b+1) = 16;       *(b[0]+1) = 17;
*(b+8) = 18;   *(b[1]+2) = 19;   *(b+1)+5 = 20;
*(b[2]+3) = 21;   *(b+2)+2 = 22;
```

Exercise 3.11. [strlen]

A string is a one dimension table with object of type char that ends by the null character ‘\0’. By using pointers, write a function that computes the number of character of a chain (we do not count the null character).

**Correction:**

```c
unsigned int chain_length(char* c)
{
    unsigned int n=0;
    while (*c != '\0')
    {
        c++;  
        n++;
    }
    return n;
}
```

This could also be written as:

```c
unsigned int chain_length(char* c)
{
    unsigned int n;
```
for (n=0, *c, c++)
    n++;
return n;
}

Exercise 3.12. [circular shift]
Write a function that takes an integer as an input (i.e. we provide its address) and transforms it in its circular shift

Correction:

void circular_shift(unsigned long int *x)
{
    unsigned long int a;
    a = *x & 1;
    *x = ( *x >> 1 ) + ( a << (8*sizeof(unsigned long) -1) );
    return;
}

Exercise 3.13. [cross product]
Write a function that computes the cross product of two vectors B and C and store it in a third vector A, with the following structure:

void cross_product(int *A, int *B, int *C);

Call this function as cross_product(int *A, int *A, int *B); for two vectors A and B. Is the result correct?

Correction:

void cross(int *A, int *B, int *C)
{
    int *D;
    if (A==B || A==C)
    {
        D=(int*) malloc(3*sizeof(int));
        D[0]=B[1]*C[2]-C[1]*B[2];
        D[2]=B[0]*C[1]-C[0]*B[1];
        A[0]=D[0];
        A[1]=D[1];
        free(D);
    }
    else
    {
        A[2]=B[0]*C[1]-C[0]*B[1];
    }
}

If we do not use the if condition, then if we give the same pointers as a first parameter and one of the two last, A[0] will be computed correctly, but A[1] and A[2] will be altered by the previous
computation. Note also that when using a temporarily pointer D, an instruction as \( A=D \) will not work, since the pointer \( A \) used in the function is a copy of the one given as parameter.

Exercise 3.14. [\texttt{strcat}] Write a function that concatenates two strings.

**Correction:**

```c
char* concatenate_chain(char* c1, char* c2)
{
    char *res, *c;
    int i, n1=chain_length(c1), n2=chain_length(c2);
    res=malloc(sizeof(char)*(n1+n2+1));
    /* The +1 is for the \0 character at the end */
    c=res;
    for (i=0; i<n1; i++)
        *c++=c1[i];
    for (i=0; i<n2; i++)
        *c++=c2[i];
    return res;
}
```

Exercise 3.15. [sparse polynomials]

We will represent a polynomial as a list of monomials; a monomial is given as a coefficient (assumed integer here) and a degree. We will only store non zero coefficients. For instance, the polynomial \( x^7 + 5 \times x^3 - 4 \times x + 2 \) is represented as a list of 4 monomials as follows:

![Diagram of polynomial representation]

1. Provide a definition of the two types `monom_t` and `polynom_t`.
2. Write a function that increment every monomial’s degree of a given polynomial (multiplication by \( x \)).
3. Give a function that evaluate a polynomial at a given value,
4. Write a function that prints on the standard output the representation of a polynomial,
5. Write a function that free the memory allocated for a polynomial,
6. Provide the code of a function that read on the standard input a polynomial given by its coefficient list: the first integer represent the coefficient of the highest degree monomial, the number of given coefficients is equal to its degree plus one; for instance, the polynomial of the example above would be given as
   
   \[
   1 \ 0 \ 0 \ 0 \ 5 \ 0 \ -4 \ 2
   \]
7. Write a function that adds a monomial to a polynomial (remember that zero coefficients are not stored),
8. Provide the definition of a function that adds two polynomials.


4 Fortran language

Contents

4.1 First practical works in Fortran ........................................... 75
Exercise 4.1: Newton’s method ............................................. 75
Exercise 4.2: Computing areas and volumes for a given radius ............. 76
Exercise 4.3: Loop order for matrices ..................................... 76
Exercise 4.4: Dynamic allocation in Fortran ................................. 77
Exercise 4.5: Structures in Fortran: sparse matrices ...................... 78
4.2 Project: Strassen matrix multiplication in fortran ....................... 79

4.1 First practical works in Fortran

Exercise 4.1. [Newton’s method]
Compute the solution of the equation \( x - e^{-x} = 0 \) by using the Newton’s method. We recall the algorithm:

Given \( x_0 \) a starting point and \( \varepsilon \) a precision we want on the result (we will ask them via the \texttt{read} function), one compute:

\[
x_{n+1} = x_n - \frac{x_n - e^{-x_n}}{1 + e^{-x_n}} \quad \text{as long as} \quad \left| \frac{x_{n+1} - x_n}{x_n} \right| \geq \varepsilon
\]

You can use the \texttt{exp} and \texttt{abs} functions here.

Correction:

```fortran
program root
    implicit none
    ! -- variables
    real :: x,y,diff,eps
    ! -- reading data
    print*, 'give me the initial value'
    read*, x
    print*, 'give me the precision'
    read*, eps
    ! -- Newton's method
    newton:do
        ! -- iteration
        y=x-(x-exp(-x))/(1.0+exp(-x))
        ! -- difference between the last two values
        if (x/=0.0) then
            diff=abs((x-y)/x)
        else
            diff=abs(y)
        endif
        ! -- updating the variables
        x=y
    enddo
endprogram root
```

75
Exercise 4.2. [circle and sphere]

1. Write a program that read a radius on the standard input and prints on the standard output the area ($\pi r^2$) and the volume ($4/3\pi r^3$) of the corresponding circle and sphere. Compile and try it.

2. Add a infinite do loop that, each time, asks the user if he wants to continue. If the answer is yes, we ask for a new radius and print the corresponding date ; if not, the program ends.

Correction:

```fortran
program radius
    implicit none
    ! -------- variables
    character(len=3)::answer
    real :: r , pi
    ! --- computing pi
    pi = acos(-1.0)
    ! --- loop
    loop : do
        ! --- reading data
        print*,'Give the radius'
        read*,r
        ! --- Printing area and volume
        print*,'area of the circle = ',pi*r**2
        print*,'volume of the sphere = ',4./3.*pi*r**3
        ! --- question
        print*,' another one ? ( yes / no ) ?'
        read*, answer
        if ( answer=='no' ) then
            exit loop
        end if
    end do loop
end program radius
```

Exercise 4.3. [multidimensional tables and memory]

Write a program that initialize two tables $a$ and $b$ with size $5000 \times 5000$.

1. First compute the product element by element of $a$ and $b$ by using an external loop on lines,

2. Do the same computation by inversing the order of the loops,

3. Print on the screen the computational time of these two loops by using the `cputime` command.

Correction:

```fortran
program variable_order
```

76
implicit none
! -------- variables
integer :: i,j
integer,parameter :: N=5000
real,dimension(N,N)::a,b,c
real::t1,t2
! --- data
call random_number(a)
call random_number(b)
! --- computing and printing timing, order i,j
call cpu_time (t1)
do i=1,N
   do j=1,N
      c(i,j)=a(i,j)*b(i,j)
   end do
end do
call cpu_time(t2)
print*, 'order i,j'
pint*, t2-t1
! --- computing and printing timing, order j,i
call cpu_time(t1)
do j=1,N
   do i=1,N
      c(i,j)=a(i,j)*b(i,j)
   end do
end do
call cpu_time(t2)
print*, 'order j,i'
pint*, t2-t1
! --- computing and printing timing, intrinsec timing
call cpu_time(t1)
c=a*b
call cpu_time(t2)
pint*, 'intrinsec function'
pint*, t2-t1
end program variable_order

Running this program, one sees a big difference: first loop is really slower; this is due to cache access and how memory is handled in Fortran.

Exercise 4.4. [dynamic allocation]
Write a program that fill a matrix of $n$ lines and $m$ columns (that will be read on the standard input, thus not known in advance) in the following way:

- even lines are filled with 1,
- odd lines are filled with increasing integers (1, 2, 3,...).

For instance, this would be the corresponding $3 \times 4$ matrix:

$$
\begin{pmatrix}
1 & 2 & 3 & 4 \\
1 & 1 & 1 & 1 \\
5 & 6 & 7 & 8 \\
\end{pmatrix}
$$
The program will print on the screen the matrix line by line.

**Correction:**

```fortran
program alloc_dyn
    implicit none
    ! -------- variables
    integer :: m,n,i,j,k=1
    integer,dimension(:,,:),allocatable::A
    ! --- dimensions
    print* , 'Give the two dimensions'
    read*,m,n
    ! --- allocation
    allocate (A(m,n))
    ! --- even lines
    do i=2,m,2
        A(i,:)=1
    end do
    ! --- odd lines
    do i=1,m,2
        do j=1,n
            A(i,j)=k
            k=k+1
        end do
    end do
    ! --- affichage
    print*,'A='
    do i=1,m
        print*,A(i,:)
    end do
end program alloc_dyn
```

**Exercise 4.5.** [sparse matrix representation]

In this exercise, we will store only non zero elements of matrices. To that purpose, a matrix A will be stored in the following structure:

```fortran
type element
    real :: coef
    integer :: indl,indc
end type element

A matrix will be stored in a file A.mat as follows:

```

| 9 | -1. 1 4 |
|---|---|---|
| 4 | 2 1 |
| 9 | 2 2 |
| 10 | 1 1 |
| 1 | 3 1 |
| -3 | 4 2 |
| 3 | 2 3 |
| 8 | 4 4 |
```

A matrix will be stored in a file A.mat as follows:
5. 3 3

The first integer represents the number of non zero elements. Then, each line is made of 3 values, respectively coef, indl and inde in the structure.

1. Write a procedure that read in a file a matrix stored as above,
2. Write a function \( \text{trace}(B) \) that computes the trace of a matrix \( B \),
3. Write a function that \( \text{mtvect}(A, x) \) that computes the product of a matrix and a vector.
4. The power iteration algorithm computes the highest eigenvalue (in module) of a matrix as follows:
   a) start with \( q_0 \) such that \( \|q_0\| = 1 \); initialize \( k \) to 0
   b) do:
      • \( k = k + 1 \),
      • \( x_k = A \cdot q_{k-1} \)
      • \( q_k = \frac{x_k}{\|x_k\|} \),
   while \( \|x_k - x_{k-1}\| \leq \varepsilon \)
   where \( \| \cdot \| \) is the euclidean norm.
5. Output \( \lambda_k = q_k^T \cdot A \cdot q_k \)

Implement such a program.

4.2 Project: Strassen matrix multiplication in fortran

Send your archive by email before Monday 29, 1pm

Let consider the following matrix multiplication:
\[
\begin{pmatrix}
W & X \\
Y & Z
\end{pmatrix} =
\begin{pmatrix}
A & B \\
C & D
\end{pmatrix}
\begin{pmatrix}
E & F \\
G & H
\end{pmatrix},
\]
where all the matrices \( A, B, \ldots, Z \) are supposed square and on the same size \( n \). Volker Strassen showed in 1969 [1] that \( W, X, Y, Z \) can be computed via the following formulae:

\[
\begin{align*}
P_1 &= (A + D)(E + H), \quad P_5 &= (A + B)H, \\
P_2 &= (C + D)E, \quad P_6 &= (C - A)(E + F), \\
P_3 &= A(F - H), \quad P_7 &= (B - D)(G + H), \\
P_4 &= D(G - E),
\end{align*}
\]

\[
W = P_1 + P_4 - P_3 + P_7, \quad Y = P_2 + P_4, \\
X = P_3 + P_5, \quad Z = P_1 + P_3 - P_2 + P_6.
\]

One can notice that this reduces the number of matrix multiplication of size \( n \) to 7 instead of 8 (but more additions). This enables to reduce the matrix multiplication complexity from \( O(n^3) \) for the naive algorithm to \( O(n^{\log_2^7}) \subset O(n^{2.81}) \) for the strassen algorithm (basically by applying recursively the division by two of the size explained above).

To simplify the process, we will only consider matrix that have a size equal to a power of 2. The aim of this project is to write an implementation of the Strassen’s algorithm in FORTRAN.

For all these algorithms, you should write them with an additional parameter that provides the “computer” size of the matrix, so that one can use them by providing the address of any element of the matrix.
(this will enable to use recursively the Strassen algorithm directly on the submatrices). More generally, we will try to be cautious on the memory space the program will use (it is possible to use only two $\frac{n}{2} \times \frac{n}{2}$ additional matrices for each strassen call made with $n \times n$ matrices).

Finally, you will separate your code in (at least) four files:

- one will contain a module for dealing with the matrix multiplication itself,
- one will contain any function you might use for printing matrices (also in a module), or to check if the result is fine (a function testing if two matrices are equal could be helpful to check your result on “big” matrices),
- the last two ones will contain your main programs, that will use the previous module to provide respectively some printable tests (on small examples) and some timings.

Obviously, you will provide (and use !) a makefile to use separated compilation for your program. It is possible that you use more files or functions… adapt your makefile accordingly.

1. Write a subroutine that computes the classical “naive” matrix multiplication,

2. Implement the above Strassen strategy in another subroutine; test it on some example to check that everything works correctly (compare the result with your “naive” algorithm), and add a target test to your makefile that run such examples,

3. The Strassen’s method is particularly not efficient on small matrices (because it uses a lot of additions). Thus, your Strassen algorithm will have to use the naive algorithm when the dimension of the matrix will be less of some threshold. Run your code with different threshold, and guess experimentally what is the good threshold (this will depend on the computer you use). Explain your choice in the report.

4. Use some random matrices with increasing sizes (only use powers of 2) to get some timings on the previous algorithms,

5. Create some graph showing the timings of these three approach,

6. Finally, to show the experimental complexity of the last mixed algorithm, show the timings by using a logarithmic scale; what is the slope you get? Is it the expected one?

Files to be sent. Send by email an archive containing the following files:

- a makefile,
- your source code (only fortran files, no .mod…),
- a small report in pdf explaining how you dealt with question 3 and your timing experimentations.
5 POSIX

5.1 Filesystem

5.1.1 File descriptors

When a process needs to use a file, it is designed by an integer. This is the index of a table that contains some information describing some data related to the file. The first three elements are:

- the standard input \texttt{stdin} (index 0); by default the keyboard,
- the standard output \texttt{stdout} (index 1); by default the screen,
- the error output \texttt{stderr} (index 2); also the screen by default.

We will now describe some functions using these file descriptors. You can find some help about them in the second section of the inline manual ($\texttt{man 2 open}$,$\texttt{man 2 write}$...).

5.1.2 The open and close functions

The prototype of this function is

\begin{verbatim}
int open(char *name, int mode,...);
\end{verbatim}

It belongs to the \texttt{<fcntl.h>} library. This opens a file \texttt{name} (this includes its path; remember that if none is given the file is searched in the current directory), returning its file descriptor (it will be the first empty box in the file descriptor table), or $-1$ in case of error. The \texttt{mode} argument can be one of the following (among others):

- \texttt{O_RDONLY}: read only,
- \texttt{O_WRONLY}: write only,
- \texttt{O_RDWR}: read and write,
- \texttt{O_APPEND}: write at the end of the file (more precisely, that puts the offset at the end of the file),
- \texttt{O_CREAT}: create the file if it does not exist yet.
• **O_TRUNC**: if the file exists, it is truncated to 0 characters

One can give several of the previous options, but if none of the first three is given, read only is assumed. The options are combined by using the | character. For instance:

```c
#include <fcntl.h>

#define FILENAME "test.txt"

int fd=open(FILENAME,O_CREAT|O_WRONLY|O_APPEND);
```

will open the file `test.txt` in the current directory (creating it if it does not exist), and enable writing inside (starting at the end of the file). Note that without the O_APPEND parameter, one will overwrite the date in your file when writing in it.

**Remark 5.1.** These constants are actually well chosen power of two. This explains that we can combine them by using the binary OR |. In particular, one can check if an option is encoded in an integer (given as an input if the open function by example) corresponds to a given option by using the binary AND &

**Example 5.1.** (O_CREAT | O_WRONLY) & O_CREAT is equal to O_CREAT

When using the O_CREAT flag, one should also provide the permission that the file will have if created as a third parameter. For instance, the following code:

```c
#include <fcntl.h>
#include <sys/stat.h>

#define FILENAME "test.txt"

int fd=open(FILENAME,O_CREAT|O_WRONLY|O_APPEND,S_IRUSR|S_IWUSR);
```

will create (if it does not exist already) the file `test.txt`, with the permissions of reading and writing in the file for the user only. You can read the O_CREAT section of the manual for more details about this ($ man 2 open). Note that the macros S_IRUSR (and others) are defined in the sys/stat.h library.

Finally, one can close an opened file with the close function, from the <unistd.h> library: the call

```c
int close(int fd)
```

will close the file associated to the file descriptor `fd`. It returns 0 in case of success and −1 otherwise.

### 5.1.3 Reading and writing in a file

The library <unistd.h> contains two functions read and write that enable respectively to read and write some given octets in a file. Their prototypes are:

```c
ssize_t read(int fd, void *buf, size_t n_byte);
ssize_t write(int fd, const void *buf, size_t n_byte);
```

(the types ssize_t and size_t are defined in the library <sys/types.h>). `fd` is the file descriptor where we want to read (resp. write), `buf` is a pointer that gives the address of the bytes where we want to store (resp. from which we want to read) the data, and `n_byte` is the number of bytes we want to read (resp. write) at once. Here are three examples where we use these functions:

**Example 5.2.**
# include <fcntl.h>
# include <sys/stat.h>
# include <unistd.h>

# define FILENAME "test.txt"
# define NUMBER 80

int main()
{
    int i,fd;
    char a='0'-10,*p=&a;
    fd=open(FILENAME,O_CREAT|O_RDWR,S_IRUSR|S_IWUSR);
    for (i=0;i<NUMBER;a++,i++)
        write(fd,p,1);
    a='\n';
    write(fd,p,1);
    close(fd);
    fd=open(FILENAME,O_CREAT|O_RDWR,S_IRUSR|S_IWUSR);
    for (i=0;i<5;i++)
    {
        read(fd,p,1);
        a+=5;
        write(fd,p,1);
    }
    a='\n';
    write(fd,p,1);
    close(fd);
    return 0;
}

Example 5.3.

# include <unistd.h>
# define STDIN 0
# define STDOUT 1
# define NUMBER 80

int main()
{
    int i;
    char a,*p=&a;
    for (i=0;i<NUMBER;i++)
    {
        read(STDIN,p,1);
        a++;
        write(STDOUT,p,1);
    }
    a='\n';
    write(STDOUT,p,1);
    return 0;
}

Example 5.4.

# include <unistd.h>
# define STDIN 0
# define STDOUT 1
# define NUMBER 1600
# define STEP 50

int main()
{
    int i;
    char *p,a='\n';
    p=malloc(STEP);
    for (i=0;i<NUMBER;i+=STEP)
    {
        read(STDIN,p,STEP);
        write(STDOUT,p,STEP);
    }
    write(STDOUT,&a,1);
    free(p);
    return 0;
}
Feel free to try them (even if they are not very useful). Finally, note the existence of the function

\[
\text{off_t lseek(int fd, off_t move, int fromwhere);}\
\]

This function, part of the `<unistd.h>` library, moves the current offset of the file corresponding to the file descriptor `fd` of `move` bytes (this is a `long int`) from the position `fromwhere`, that can be:

- `SEEK_SET`, the beginning of the file,
- `SEEK_CUR`, the current offset,
- `SEEK_END`, the end of the file.

### 5.1.4 The `strace` command

The `strace` command enables to get the set of system calls made by a process, together with the set of signals it receives. We will not detail everything here (you can use `man` if you want to understand the different functions called; note that not all the functions are documented). By default, its output is sent on the error output `stderr`.

In particular, it enables to see the impact of the buffer used by the functions of the library `<stdio.h>`. For instance, you can run `strace` on the executable produced by the following code:

#### Example 5.5.

```c
#include <stdio.h>

int main()
{
    int i;
    printf("Hello");
    putchar('\n');
    printf("Hello again ");
    printf("world\n");
    putchar('H');
    putchar('e');
    putchar('l');
    putchar('l');
    putchar('o');
    for (i=0;i<500;i++)
        putchar('3');
    putchar(' ');
    for (i=0;i<50;i++)
    {
        putchar('2');
        putchar(' ');
    }
    for (i=0;i<3000;i++)
        putchar('p');
    printf('n');
    return 0;
}
```

To make the difference between the standard output and the error one, you can run `strace` by redirecting the error output to a file. For instance:

```bash
$ strace ./trace.out 2>trace.txt
```
5.1.5 Getting information on a node

Two values might be used to get informations on a node of a file-system:

- Values of type `struct dirent` returned by the system call `readdir()`;
- Values of type `struct stat` returned by the system call `stat()`, `lstat()` (in the case of a symbolic link, this provides informations on the file itself and not on the file designed by the symbolic link) or `fstat()` (informations on a file designed by a descriptor).

These structures contain the following fields: Note that these are the fields defined by the POSIX norm; a particular implementation can provide additional informations that we will try not to use.

```
struct dirent {
  ino_t  d_ino;  /* File number of entry */
  char   d_name[];  /* Name of entry */
}

struct stat {
  dev_t   st_dev; /* Device ID of device containing file */
  ino_t   st_ino; /* File serial number */
  mode_t  st_mode; /* Mode of file */
  nlink_t st_nlink; /* Number of hard links to the file */
  uid_t   st_uid; /* User ID of file */
  gid_t   st_gid; /* Group ID of file */
  dev_t   st_rdev; /* Device ID (if file is character or block special) */
  off_t   st_size; /* For regular files, the file size in bytes. For symbolic links, the length in bytes of the pathname contained in the symbolic link */
  time_t  st_atime; /* Time of last access */
  time_t  st_mtime; /* Time of last data modification */
  time_t  st_ctime /* Time of last status change */
  blksize_t st_blksize; /* A file system-specific preferred I/O block size for this file */
  blkcnt_t st_blocks; /* Number of blocks allocated for this file */
```

The following macros can be used on the `st_mode` field:

- `S_ISREG(m)` Test for a regular file.
- `S_ISDIR(m)` Test for a directory.
- `S_ISLNK(m)` Test for a symbolic link.
- `S_ISFIFO(m)` Test for a pipe or FIFO special file.
- `S_ISSOCK(m)` Test for a socket.
- `S_ISBLK(m)` Test for a block special file.
- `S_ISCHR(m)` Test for a character special file.

5.2 Coding in POSIX

5.2.1 POSIX constraints

The POSIX norm provides some constraints on sizes. In particular, there are minimal values that must ensure any implementation of the norm. On another hand, the POSIX norm imposes that these values are accessibles to applications. Among these macro-definitions provided by the files `<limits.h>` et `<unistd.h>`, one finds:

- `NAME_MAX` the maximal length size for the name of the system’s files (the required minimum is 14)
• **PATH_MAX** the maximal length size of a path of the file-system (the minimum value required is 255).

### 5.2.2 POSIX source code

In order to tell the compiler that we want to compile a C code that fits to the POSIX norm, the macro \_XOPEN_SOURCE has to be greater or equal to 500. This can be done with the following variables in the makefile:

```bash
CC     = gcc
CFLAGS = -Wall -ansi -pedantic
CFLAGS += -D_XOPEN_SOURCE=500
CFLAGS += -g
```

If you want to see more, you can have a look in the file `/usr/include/features.h`
## Exercise 5.1: Our which command

The Unix command `which` searches commands in the user’s path. The environment variable `PATH` contains a list of directories separated by `:`. When one runs a command from the shell, an executable file corresponding to that name is searched in this list of directories. This will be the file ran; if no file is found, this leads to an error message.

The `which` command prints the full path of this command.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/home/ens/poteaux/bin
$ which ls
/bin/ls
$ echo $? 0
$ which foo
$ echo $? 1
```

1. **Listing the research repertories.** We will first write a function `filldirs` that fills a table `dirs` that contains the list of directory paths in the `PATH` variable. It will end with a NULL pointer.

### Correction:

- The table `dirs` contains pointers towards the beginning of the repertory names in the variable `PATH`.
- We replace `:` characters by `\0` so that these pointers are indeed strings (warning: this destroys the variable! One could allocate memory to make a copy of the variable to solve this matter).
- The variable `dirs` must be declared outside the function to be useful (or the function must return the pointer).
- We need to allocate memory for the variable `dirs` according to the number of directories. This memory has to be freed in the main function.
- The `assert` call in the code below should detail more properly the errors... finir proprement...

```c
/* null terminated array of dirname in PATH */
static char **dirs;
```
/* fill dirs with content of $PATH
   warning: "kill" $PATH */
static void filldirs(void)
{
    char *path;
    int length;
    int i;
    char *p;

    path = getenv("PATH");
    assert(path != NULL);

    /* count number of dir
       and null terminate dirnames of PATH */
    for (length=1, p=path; *p; p++)
        if (*p == ':')
        {
            length++;
            *p = '\0';
        }

dirs = malloc((length+1) * sizeof(char *));
    assert(dirs);

    for (i=0; i<length; i++)
    {
        dirs[i] = path;
        path += strlen(path)+1; /* next dirname */
    }
dirs[length] = (char *)0;
}

2. A which function. We will now write the function which that prints the full path corresponding to the command given as a parameter. If the command cannot be found in the list of directories of the variable PATH. The function will return a boolean value according to the success or not of the search. One will use the system call

    #include <unistd.h>
    int access(const char *path, int mode);

    that checks the permissions of a given file according to a mask (i.e. a combination with the bit operator |) of the macro F_OK (existence), R_OK (reading), W_OK (writing) and X_OK (executable).

Correction:
- Note that the size used for the pathname variable is defined in <limits.h>,
- man 3 snprintf will provide you details on that function (very useful to deal with string manipulations)

    /* return a success of a failure */
static int which(const char *command)
{
    char **dirname;
    char pathname[PATH_MAX+1];
    int status;

    for (dirname = dirs; *dirname; dirname++)
    {
        snprintf(pathname, PATH_MAX, "%s/%s", *dirname, command);
        status = access(pathname, X_OK);
        if (!status)
        {
            printf("%s
", pathname);
            return TRUE;
        }
    }
    printf("%s: Command not found\n", command);
    return FALSE;
}

3. **The which command.** Finally, write your own version of the which command that ends with a success if and only if all the given commands have been found in the directories listed in the variable PATH.

**Correction:**
- We just read the parameters and call the which function for each of them.
- Here again, the assert function should be replaced by something better...

```c
int main (int argc, char *argv[])
{
    int status = EXIT_SUCCESS;
    int i;

    assert(argc > 1);

    filldirs();

    for (i=1 ; i< argc ; i++)
    {
        if (! which(argv[i]))
            status = EXIT_FAILURE;
    }

    exit(status);
}
```

**Exercise 5.2.** [Finding files]

We want to implement our own (basic) version of the function find. Our program mfind will search files given as a parameter only from the current directory.
1. Give the definition of the function

   ```c
   int valid_name(const char* name)
   ```

   that indicates if the path `name` has to be considered by the command `mfind` or not. To ensure that the program will terminate, we will not consider the path `.` and `..`.

2. Write a recursive function

   ```c
   int find_file(const char* path, const char* file_name)
   ```

   that prints on the standard output the path of the current node if its name is the same as the one of the input, and make a recursive call in the subdirectories if the current path is a directory.

3. Finally, provide the main function to define our `mfind` command.

**Exercise 5.3.** [fork and mutation]
Write the implementation of a command

   ```
   do cmd
   ```

   that run the command `cmd` in a child process

**Exercise 5.4.** [redirection]
Write the implementation of a command

   ```
   to file cmd
   ```

that run the command `cmd` one after by redirecting its standard output to the file `file` that will possibly be overwritten at the beginning of the execution of our command `to`.  

90
6 Final project

Send your archive by email before Wednesday, 15th of October, 11pm

Contents

6.1 A generic quicksort algorithm .............................................. 91
6.2 Disc usage ........................................................................ 92
6.3 A pipe function ................................................................. 93
6.4 Testing the different commands ........................................... 94

Remark 6.1. This project will ask the creation of several independant commands, and use them together at the end. Therefore, you should always work step by step, testing your code at each step. Note that it will be asked to provide some test targets in your makefile that will run some final or intermediary tests (the requirements are described all along the project). Every files should belong to the same directory, but you can create subdirectories if you feel the need. In any case, you will have only one makefile to compile all the code and answer to the test questions.

6.1 A generic quicksort algorithm

We remember that the quicksort algorithm is based on a partition of the table in two different areas. We assume having:

- a table we want to sort,
- a function $f$ to compare two elements of the table that returns:
  - 0 if the two elements are equivalent,
  - a positive value if the first element is considered greater than the second,
  - a negative value otherwise.

The idea to sort the table is to take the first element $x$ of the table (one could take something else), and to partition the table in two parts so that:

- any element $y$ in the first part of the table satisfies $f(y, x) < 0$
- any element $z$ in the second part of the table satisfies $f(x, z) \leq 0$

Then, we apply recursively this process to the two parts of the table to finish the sorting (recursivity ending on one element tables).

Note that this version of the algorithm is destructive (i.e. we change the initial table during the process).

As for the partition algorithm, we can follow the following process:

1. We take as a pivot one element of the table (for instance the first one),
2. We start with two pointers down and up, initially the first and last elements; as long as $up$ is greater than $down$, we do the following:
   - increment $down$ as long as $f(*down, x) < 0$,
• decrement up as long as \( f(x, \ast up) \leq 0 \),
• if down is strictly less than up, exchange them (i.e. their content).

3. Finally, one exchange the pivot and down (here also, we mean their content).

The questions are the following:

1. Give the definition of the following prototype:

   ```c
   void quicksort(void *base, int nelem, int size,
                  int(*compar)(const void *, const void *));
   ```

   where:
   • base is a pointer on the first element of the table we want to sort,
   • nelem the number of elements of the table we want to sort,
   • size the size, in bytes, of an element of the table,
   • compar is a pointer of the comparison function ; this function takes two pointers on the
     elements to compare and returns a value as the function \( f \) above.

   This function will be written in a separated file, and you will create the corresponding header file,
   in order to use it in the next questions of this section. As usual, you will use a makefile to deal
   with the separated compilation. This makefile will be the same for all the program you will have
   to create in this project. Further requirements for your makefile will be asked in the next questions
   of the project.

2. Using the quicksort algorithm above, produce a function msort that sort the lines given on the
   standard input and print the sorted lines on the standard output. This function will not take any
   parameter. One can limit ourself to files containing at most MAXLINE lines, with each line having
   at most NMAXCHAR characters. You might want to use the strcmp function from the string.h
   library to compar the different lines.

3. Add a target test-qsort to your makefile that will run some test to show the correctness of
   your algorithm. Be cautious on the dependencies of this target ; in particular, if needed, make
   should compile your code first. Also (and this is true for all the test targets that will be asked later
   on), your clean target should remove any intermediary file your test might create (if it creates
   any).

4. Add an option -num to your msort command that will sort lines (still of the standard input)
   beginning by an integer (followed by a space) according to the value of this integer (the work here
   is only to define the comparison function - additionally with dealing with the option)

### 6.2 Disc usage

The aim of this section is to implement our own version of the function du. This command prints on the
standard output the size used on the hard drive of a set of files (and the set of files in the given directories,
recursively). Two kind of files can be taken into account for a file:

• the apparent size, which is the number of bytes contained in a file (field st_size of the
  struct stat structure - see section 5.1.5),

• the real size, which is the number of disk blocks (field st_blocks).

As the command du, our program mdu will have two options available:
• -L to follow symbolic links,
• -b to print apparent sizes (by default, one prints real sizes).

For instance, one can use a global variable to memorize the state of this option:

```c
static int opt_follow_links = 0;
static int opt_apparent_size = 0;
```

You are encouraged to write your code step by step, checking its correctness each time. In particular, at first, we will assume that the option -L is not provided.

1. Give the definition of the function

```c
int valid_name(const char* name)
```

that indicates if the path name has to be considered by the command mdu or not. To ensure that the program will terminate, we will not consider the path . and ..

2. Write a recursive function

```c
int du_file(const char* pathname);
```

that returns the size of a given file (and its potential subdirectories). Note that in the case of a link, since we are not following it, we are interested by the size of the file itself, and not the file it points to. Here again, the function snprintf should be useful for the construction of the recursive paths.

3. We now consider the -L option. At first, you might want to define a variable number_links_followed, and stop following links when this variable is greater than a given macro (this will avoid any infinite loop in case of circular links).

4. A node that would be referenced several times in the hierarchy could be counted several times. How could we deal with it?

5. *(bonus)* Provide an implementation of a (at least partial) solution.

Here again, you will add some target test-du to your makefile that test your program; you can use the du command to check the result for instance.

**Remark 6.2.** For testing your program, note that your function ./mdu is supposed to be equivalent to du -s -B 512 (at least on the computers of the lecture room).

### 6.3 A pipe function

The aim of this section is to write a command

```c
pipe command1 to command2
```

that creates a pipe between commands command1 and command2 (i.e. by redirecting the standard output of command1 to the standard input of command2 via an anonymous pipe). As before, you will add a test-pipe target that uses this function (use it with native command so that the test is only on your pipe command; next section is about using all your commands together).

**Remark 6.3.** *In order to deal easily with the parameters of the exec* function you will use, you might want to replace the to parameter by a NULL pointer...
6.4 Testing the different commands

The point of this section is to use all the previous commands together. To that purpose, you will find here: 
http://www.lifl.fr/~poteaux/fichiers/teaching/test-prof-final-project.tar.gz

two different files:

- **partial-makefile** that contains a few macros and two targets that you will add to your makefile (pay attention to change the macro defining your command’s names correctly),

- **res-test-prof.txt** that provides the result I got with my implementations when running the command `make test-prof`. This is only to give you an idea on how the result should look like. It is quite probable that you will not get exactly the same result (in particular, you will get results close to mine when dealing with circular links only if you implemented the bonus question of section 6.2). It is also possible you get some slight differences due to the way you implemented your `du` command; this is no problem.

Finally, you are asked to add a target `test` to your makefile that will run all the test targets you have in your makefile (the one that are required in the previous section, the `test-prof` I provide, and any other you might have added (this is just a shortcut similar to the target `all` you use for the compilation of your programs; this `all` target should still be the first of your makefile).
7 Resources

7.1 Emacs

7.1.1 Some hints about Emacs

To open a file `toto.txt`, type `$ emacs toto.txt`. If this file does not exist, that will open a new window with the given name. If emacs recognise the type of the file (and the corresponding mod is installed), then the text will be coloured, and automatic indentations will be available (with the TAB key). For instance, a file beginning by `#! /bin/sh` will be recognised as a script. Same for a file with the extension `.sh`. A file with the extension `.c` will load the emacs C mode...

Remember that one can use a & to run it in the background, or the combination `CTRL+z` then `bg` afterwards. Also, if we use the option `-nw`, emacs is run inside the terminal.

Some shortcuts (here, `C-` stands for the CTRL key, and `M-` called meta - stands for either the ALT or ESC key):

**Running emacs**

- `C-x C-c` quit emacs,
- `C-x C-f` opens a new file,
- `C-x k` kills an opened file,
- `C-x C-v` opens a new file and close the current one,
- `C-x C-s` saves a new file,
- `C-x b` moves to another buffer,
- `C-x leftarrow` moves to the next left buffer,
- `C-x rightarrow` moves to the next right buffer.

**Moving into a file**

- `M-<` moves to the beginning of the document,
- `M->` moves to the end of the document,
- `C-s` searches forward,
- `C-r` searches backward,
- `C-l` centers the screen on the current line,
- `C-leftarrow` moves the cursor to the beginning of the current/previous block,
- `C-rightarrow` moves the cursor to the end of the current/next block,
- `C-l` centers the screen on the current line,
- `M-x goto-line l` moves to the l-th line of the file.
Editing

- **TAB** indents current line,
- **C-k** deletes the current line, starting from the cursor position,
- **C-_** cancels the last operation,
- **M-%** search and replace,
- **C-space** starts selecting a region from the cursor position,
- **M-w** copy,
- **C-w** cut,
- **C-y** paste,

Screen splitting

- **C-0** Remove the current window from the screens,
- **C-1** Make active window the only screen,
- **C-2** Split screen horizontally,
- **C-3** Split screen vertically,
- **C-x o** Move to next screen.

Running commands

- **C-g** cancels the current operation (to move back to the window),
- **M-$** check spelling of the word at cursor or the selected area,
- **M-!** run a shell command,
- **M-x shell** starts a shell within emacs,
- **M-x compile** starts the compiler for the active window,
- **M-x space** gives available commands,
- **C-h emacs helps,**
- **C-h t** runs the emacs tutorial,
8 Annexes

8.1 Inspirations of this lecture

This list will be detailed later on...

- First chapter: Mathieu Nebra\textsuperscript{1} and Harry Mangalam\textsuperscript{2}

- System and C: Philippe Marquet, Alexandre Sedoglavic, François Lemaire and François Boulier (several lectures in computer science given at the university Lille 1)

- Luc Mieussens\textsuperscript{4} for the chapter on Fortran.

\textsuperscript{1}http://fr.openclassrooms.com/informatique/cours/reprenez-le-controle-a-l-aide-de-linux
\textsuperscript{2}http://moo.nac.uci.edu/~hjm/ManipulatingDataOnLinux.html
\textsuperscript{3}http://moo.nac.uci.edu/~hjm/biolinux/Linux_Tutorial_2.html
\textsuperscript{4}http://www.math.u-bordeaux1.fr/~lmieusse/PAGE_WEB/ENSEIGNEMENT/cours_f90.pdf
Bibliography