

TD 1 : Que peut-on faire d'un fichier source ? *

ED6 - Licence 3 - Université Paris Diderot

Séances du 5 et 12 janvier 2013

L'objectif de ces travaux dirigés est de s'initier à une manipulation efficace des fichiers sources, ces fichiers textes qui servent à stocker les programmes. À l'issue de cette séance, vous devrez être capables de :

- { Utiliser les principales commandes de l'éditeur de texte EMACS, efficacement à l'aide des raccourcis claviers.
- { Archiver une arborescence, décompresser une archive
- { Écrire un *script shell* UNIX pour extraire des informations d'un fichier source ou pour le transformer.

Le sujet de ce TD est principalement centré sur l'utilisation et la vérification de conformité d'un programme C à une *norme d'écriture de programmes*. De nombreuses équipes de développement utilisent une telle norme de façon à rendre le code source plus lisible en augmentant son homogénéité, sa simplicité et la qualité de la mise en page. La norme que nous allons utiliser aujourd'hui, et durant tout ce cours, est décrite dans l'annexe A. Bien sûr, une norme ne doit pas forcément être suivie à la lettre : il s'agit de se donner des règles générales qui s'appliquent dans 99% des cas, le pourcentage restant étant formé des exceptions qui les contournent !

En guise de support, nous allons utiliser un ensemble de fichiers sources écrits dans le langage C. Vous pouvez télécharger ces fichiers sur la page du cours : <http://upsilon.cc/zack/teaching/1213/ed6/>. Vous organiserez votre répertoire de travail pour ce TD de la façon suivante :

.	Répertoire de travail
-- dictionary	Bibliothèque de travail
'-- src	Fichiers sources téléchargés sur le site
'-- scripts	Sources des scripts (exercice 3 et exercice bonus)

1 Édition



PRÉFÉRER LE CLAVIER À LA SOURIS POUR LES COMMANDES COURANTES

Un raccourci clavier permet de gagner du temps à condition de ne pas avoir à se le remémorer : son utilisation doit être « automatique ». Il faut donc faire un effort pour les utiliser de façon systématique et les assimiler avec le temps. Vous trouverez en annexe une fiche regroupant les principaux raccourcis clavier d'EMACS. Faites l'effort de l'avoir toujours avec vous lorsque vous programmez, vous progresserez alors significativement !

Nous allons (re)faire un tour d'horizon de l'éditeur EMACS, en nous focalisant sur les raccourcis clavier utiles à la programmation. Mettez donc votre souris de côté ! Elle est interdite à partir de maintenant. Voici comment lire la description des raccourcis de la fiche :

- { C-x signifie qu'on doit appuyer sur la touche **C**ontrô**l** et la touche **x** simultanément.
- { M-x signifie qu'on doit appuyer sur la touche **M**eta et la touche **x**. Dans EMACS, la touche **Meta** fait référence en général à la touche **Alt** ou parfois **Echap**. Il s'agit d'une touche symbolique qu'on peut configurer à sa guise.

Un certain nombre de critères de la norme ne peuvent pas être traités automatiquement (ou bien difficilement). Leur vérification et leur correction sont donc faits manuellement. C'est ce que nous allons faire dans cette section.

Exercice 1

1. Récupérez l'archive <http://upsilon.cc/~zack/teaching/1213/ed6/tp1-files.tar>, décompressez l'archive et placez les fichiers dans `dictionary/src`.
2. Ouvrez l'ensemble des fichiers contenus dans `dictionary/src`. À l'aide des commandes de la section **Buffers** de la fiche, naviguez à travers ces différents fichiers et à l'intérieur de chacun d'eux sans utiliser les touches directionnelles du clavier (section **Motion**).

*Ce sujet de TD est (très) largement inspiré d'un sujet proposé par Yann Régis-Gianas



Dans EMACS, un *buffer* est une zone de travail. Il peut s'agir d'un fichier mais aussi d'une zone d'interaction entre l'utilisateur et l'éditeur (comme par exemple, le *buffer* contenant la liste des *buffers* ouverts). Un des *buffers* a un statut special : le *mini-buffer*. C'est une zone qui sert à gérer l'interaction courante entre l'éditeur et l'utilisateur.

3. Visualisez un fichier `index.c` et son interface `index.h` en divisant la fenêtre principale d'EMACS en deux fenêtres verticales à l'aide des raccourcis de la section **Multiple Windows**.
4. La fonction `index.create` n'est pas exportée dans le fichier `index.h`. Toujours à l'aide de raccourcis claviers et sans utiliser les touches directionnelles du clavier, placez-vous dans le fichier `index.c`, au début de l'en-tête de cette fonction et copiez-collez cette en-tête dans le fichier d'interface. Enregistrez vos modifications à l'aide du raccourcis idoine de la section **Files**



La plupart des éditeurs de texte possèdent un presse-papier (*clipboard*) qui sert de zone temporaire de stockage au mécanisme de copier/coller. EMACS fournit une **pile de presse-papiers**, appelée *kill-ring*. Elle permet de stocker un nombre non borné de portions de texte à copier/coller.

5. Effectuez la même transformation que dans la question précédente mais pour les deux fonctions `index.release` et `index.compare` en faisant deux opérations "copier" puis deux opérations "coller".
6. Dans le fichier `cast.h`, mettez en majuscule la macro `declare.as` à l'aide du raccourci clavier de la section **Case Change**.
7. Dans chacun des fichiers `.c`, remplacez toutes les occurrences de la macro `declare`

Exercice 2

1. Lisez la documentation de l'outil `tar`.
2. Produisez un fichier `dictionary.tar` contenant l'arborescence dont la racine est le repertoire `dictionary`.
3. Compressez l'archive obtenue a l'aide de l'outil `gzip`.
4. Déterminez la commande permettant d'effectuer les deux étapes précédentes en un unique appel a `tar`.
5. Modifiez un fichier source du repertoire `src`.
6. Décompressez l'archive.
7. Calculez la différence entre l'arborescence et l'archive a l'aide de `tar`.
8. Créez un fichier `README` dans le repertoire `dictionary`.
9. Mettez a jour l'archive pour prendre en compte la nouvelle arborescence.
10. Calculez a nouveau la différence entre l'arborescence et l'archive. Comment interprétez vous le resultat ?

3 Un peu de shell-scripting



LES ACTIVITÉS RÉPÉTITIVES DOIVENT ÊTRE AUTOMATISÉES

Il vaut mieux passer une heure a écrire un script qui automatisera une tâche quotidienne demandant 10 minutes si elle est effectuée manuellement. Au bout de seulement 6 jours, vous gagnez 10 minutes de temps de travail par jour et par tâche !

Exercice 3

1. Écrivez un script `backup.sh` qui effectue la sauvegarde d'un fichier `foo.ext` en copiant son contenu dans un fichier dont le nom suit le format : `.foo.ext.backup`. Que se passe-t-il si le fichier `.foo.ext.backup` existe déjà ?
2. Modifiez `backup.sh` pour qu'il effectue la sauvegarde de `foo.ext` dans le fichier `.foo.backup`. Si le fichier existe déjà, le script échoue (le statut de retour est 1).
3. Modifiez `backup.sh` pour qu'il effectue la sauvegarde de `foo.ext` dans le fichier `.foo-jour-mois-annee-heure-minute-seconde.backup`. Si le fichier existe déjà, le script échoue (le statut de retour est 1).

Indice : Consultez la page de manuel de la commande `date`.

4. Écrivez un script `clean-backup.sh` qui fait la liste de tous les fichiers du repertoire courant dont l'extension est `.backup` et les supprime apres en avoir eu confirmation de la part de l'utilisateur.

Exercice Bonus

1. Parmi les points de la norme, lesquels vous semblent vérifiables automatiquement par un script (simple) ?
2. Pour chacun des criteres de la norme que vous avez exhibes, écrivez un script nommé `check-C-norm-N-M.sh` qui vérifie que le critere `N-M` est bien vérifié. Si c'est bien le cas, le statut d'erreur sera 0, sinon ce sera 1. En cas de rejet, un message d'erreur de la forme `nom-du-fichier:numero-de-ligne:message` sera produit sur la sortie standard d'erreur. Vous stockerez ce script dans le repertoire `scripts`.
Pour cette question, la collaboration inter-etudiants est encouragée ! Adoptez un critere a chacun d'entre vous et échangez-vous vos scripts ! Bien sûr, vous devrez avoir compris le fonctionnement des scripts que vous utilisez.
3. Incluez l'ensemble des tests de criteres dans un script nommé `check-C-norm.sh`, stocker dans votre repertoire `scripts`, et dont le comportement correspond a la spécification décrite plus haut.
4. Écrivez un script `install.sh` qui copie l'ensemble des fichiers `.sh` dans votre arborescence personnelle `$(HOME)/usr/bin`.
5. Déterminez l'ensemble des criteres que vous pouvez corriger automatiquement dans un fichier non conforme a la norme.
6. Pour chacun des criteres exhibes par la question précédente, écrivez un script `fix-C-norm-N-M.sh` qui corrige le fichier pris en argument. On n'oubliera pas d'utiliser le script `backup.sh` pour sauvegarder l'état du fichier avant modification.
Pour cette question, la collaboration inter-etudiants est encouragée ! Adoptez un critere a chacun d'entre vous et échangez-vous vos scripts ! Bien sûr, vous devrez avoir compris le fonctionnement des scripts que vous utilisez.
7. Modifiez le script `check-C-norm.sh` en lui rajoutant une option `\-c` qui active la correction automatique des fichiers a l'aide des scripts précédents. Il devra vérifier que la correction des fichiers invalides est bien conforme a la norme.

A Norme d'écriture pour le langage C

A.1 Structure globale

1. Une ligne ne doit pas excéder 80 caracteres.
2. Un bloc (ensemble d'instructions entre accolades) ne doit pas dépasser 15 lignes.
3. On alignera l'ensemble des identifiants intervenant dans une liste de declarations.
4. Toute accolade ouvrante doit être suivie d'un retour a la ligne.
5. Tout point-virgule est suivi d'un retour a la ligne ou d'un espace.
6. Toute virgule est suivi d'un retour a la ligne ou d'un espace.
7. On sautera une ligne entre les declarations de variables et les instructions.
8. Le code doit être indente.

A.2 Expression

1. Un operateur binaire doit toujours être precede et suivi par un espace.
2. Il doit toujours y avoir un espace avant une parenthese.
3. Un operateur unaire est toujours colle a la sous-expression sur laquelle il s'applique.

A.3 Instruction

1. Il ne doit y avoir qu'une instruction par ligne.
2. Les affectations au sein des expressions sont interdites.
3. Le mot-cle **return** est toujours suivi d'une expression entre parentheses.

A.4 Déclaration

1. Les declarations multiples de pointeurs sont interdites.
2. Les étoiles servant a denoter les types de pointeur doivent être collees au type des donnees pointees.
3. Toute variable dont la valeur initiale est importante doit être initialisee au moment de sa declaration.
4. Les noms de type declares par l'utilisateur doivent se terminer par `_t`.
5. Les noms de structure declares par l'utilisateur doivent se terminer par `_s`.
6. Les noms d'enumeration declares par l'utilisateur doivent se terminer par `_e`.
7. Les macros doivent être definiés en lettres majuscules.
8. Les identifiants doivent être le plus informatifs possible.
9. On limitera le plus possible l'utilisation de variables globales.

A.5 Commentaires

1. Les commentaires doivent suivre le format du logiciel DOXYGEN decrit ici :
<http://www.stack.nl/~dimitri/doxygen/docblocks.html>
2. Une unique langue doit être utilisee dans les commentaires.
3. L'orthographe des commentaires doit être correcte.
4. Une documentation n'est pas une paraphrase du code : elle doit contenir une information qui n'est pas evidente a la lecture de ce dernier (condition d'utilisation de la fonction, propriete de la sortie, invariants des structures de donnees utilisees, ...).
5. Un fichier d'en-tête (.h) commence par un prologue decrivant l'utilisation des fonctions et des types qu'il declare.
6. Un fichier d'implementation (.c) commence par un prologue decrivant les details de fonctionnement du code.

A.6 Unités de compilation

1. Un fichier `foo.c` doit inclure un fichier `foo.h`.
2. Un fichier d'en-tête `foo.h` doit être protege contre la double inclusion par un identifiant du preprocesseur de la forme `FOO_H` (eventuellement pre fixe par le nom de la bibliotheque).
3. Seules les inclusions strictement necessaires aux prototypes des fonctions doivent apparaître dans les fichiers d'en-tête.
4. La liste des inclusions debute par l'inclusion des modules necessaires du systeme puis des modules de l'utilisateur.

GNU Emacs Reference Card

(for version 23)

Starting Emacs

To enter GNU Emacs 23, just type its name: **emacs**

Leaving Emacs

suspend Emacs (or iconify it under X)	C-z
exit Emacs permanently	C-x C-c

Files

read a file into Emacs	C-x C-f
save a file back to disk	C-x C-s
save all files	C-x s
insert contents of another file into this buffer	C-x i
replace this file with the file you really want	C-x C-v
write buffer to a specified file	C-x C-w
toggle read-only status of buffer	C-x C-q

Getting Help

The help system is simple. Type C-h (or F1) and folle

GNU Emacs Reference Card

Buffers

select another buffer	C-x b
list all buffers	C-x C-b
kill a buffer	C-x k

Transposing

transpose characters	C-t
transpose words	M-t
transpose lines	C-x C-t
transpose sexps	C-M-t

Spelling Check

check spelling of current word	M-\$
check spelling of all words in region	M-x ispell-region
check spelling of entire buffer	M-x ispell-buffer

Tags

find a tag (a definition)	M-.
find next occurrence of tag	C-u M-.
specify a new tags file	M-x visit-tags-table
regex search on all files in tags table	M-x tags-search
run query-replace on all the files	M-x tags-query-replace
continue last tags search or query-replace	M-,

Shells

execute a shell command	M-
run a shell command on the region	M-
filter region through a shell command	C-u M-
start a shell in window *shell*	M-x shell

Rectangles

copy rectangle to register	C-x r r
kill rectangle	C-x r k
yank rectangle	C-x r y
open rectangle, shifting text right	C-x r o
blank out rectangle	C-x r c
prefix each line with a string	C-x r t

Abbrevs

add global abbrev	C-x a g
add mode-local abbrev	C-x a l
add global expansion for this abbrev	C-x a i g
add mode-local expansion for this abbrev	C-x a i l
explicitly expand abbrev	C-x a e
expand previous word dynamically	M-/

Regular Expressions

any single character except a newline	.	(dot)
zero or more repeats	*	
one or more repeats	+	
zero or one repeat	?	
quote regular expression special character <i>c</i>	\c	
alternative ("or")		
grouping	\(... \)	
same text as <i>n</i> th group	\n	
at word break	\b	
not at word break	\B	
entity		match start
line	^	\$
word	\<	\>
buffer	\'	\'
class of characters		match these
explicit set	[...]	[^ ...]
word-syntax character	\w	\W
character with syntax <i>c</i>	\sc	\Sc

International Character Sets

specify principal language	C-x RET l
show all input methods	M-x list-input-methods
enable or disable input method	C-\
set coding system for next command	C-x RET c
show all coding systems	M-x list-coding-systems
choose preferred coding system	M-x prefer-coding-system

Info

enter the Info documentation reader	C-h i
find specified function or variable in Info	C-h S
Moving within a node:	
scroll forward	SPC
scroll reverse	DEL
beginning of node	. (dot)

Moving between nodes:

next node	n
previous node	p
move up	u
select menu item by name	m
select <i>n</i> th menu item by number (1-9)	n
follow cross reference (return with 1)	f
return to last node you saw	l
return to directory node	d
go to top node of Info file	t
go to any node by name	g

Other:

run Info tutorial	h
look up a subject in the indices	i
search nodes for regex	s
quit Info	q

Registers

save region in register	C-x r s
insert register contents into buffer	C-x r i
save value of point in register	C-x r SPC
jump to point saved in register	C-x r j

Keyboard Macros

start defining a keyboard macro	C-x (
end keyboard macro definition	C-x)
execute last-defined keyboard macro	C-x e
append to last keyboard macro	C-u C-x (
name last keyboard macro	M-x name-last-kbd-macro
insert Lisp definition in buffer	M-x insert-kbd-macro

Commands Dealing with Emacs Lisp

eval sexp before point	C-x C-e
eval current defun	C-M-x
eval region	M-x eval-region
read and eval minibuffer	M-:
load from standard system directory	M-x load-library

Simple Customization

customize variables and faces M-x customize

Making global key bindings in Emacs Lisp (example):

```
(global-set-key (kbd "C-c g") 'search-forward)
(global-set-key (kbd "M-#") 'query-replace-regexp)
```

Writing Commands

```
(defun command-name (args)
  "documentation" (interactive "template")
  body)
```

An example:

```
(defun this-line-to-top-of-window (line)
  "Reposition current line to top of window.
With ARG, put point on line ARG."
  (interactive "P")
  (recenter (if (null line)
                 0
                 (prefix-numeric-value line))))
```

The *interactive* spec says how to read arguments interactively. Type C-h f *interactive* for more details.

Copyright © 2010 Free Software Foundation, Inc.
For GNU Emacs version 23
Designed by Stephen Gildea

Permission is granted to make and distribute modified or unmodified copies of this card provided the copyright notice and this permission notice are preserved on all copies.

For copies of the GNU Emacs manual, see:

<http://www.gnu.org/rg/eF32-0-4/#Mnral>,