TP 2 : `struct`, `enum`, et `union`

Programmation en C (LC4)

Semaine du 13 février 2012

1 Manipulation de rationnels, ou comment utiliser struct

Il existe principalement deux types numériques en `C` les entiers (`char`, `short`, `int`, `long`, `long long`, etc.) et les flottants (`float`, `double`). Nous allons définir un nouveau type pour représenter des rationnels sous forme de fraction.

Exercice 1

1. Dans un fichier `types_numeriques.h`, définir une structure `fraction` avec deux champs entiers `denominateur` et `numerateur`.
2. Dans un fichier `fractions.c`, qui importe la définition de la structure `fraction` à l'aide de la directive `#include "types_numeriques.h"`, implémenter une fonction :

   `void affiche_fraction(struct fraction f)` qui affiche une fraction sur la sortie standard, suivie d'une valeur approchée de cette fraction.
3. Définir un tableau `ex_fractions` contenant les fractions : $\frac{1}{1}$, $\frac{1}{2}$, $\frac{2}{4}$, $\frac{15}{10}$, $\frac{-9}{3}$, $\frac{8}{-20}$, $\frac{-5}{-10}$, $\frac{1}{-3}$. Testez votre fonction `affiche` sur chacune de ces fractions.
4. Implémenter une fonction `struct fraction reduit(struct fraction f)` qui renvoie la fraction irréductible [1] correspondant à la fraction passée en argument. Pour cela vous écrirez une fonction `long long pgcd(long long a, long long b)` qui renvoie le plus grand diviseur commun à deux entiers.

   Testez votre fonction sur les fractions du tableau `ex_fractions`.
5. Implémenter les fonctions :
   - `struct fraction add_frac(struct fraction f1, struct fraction f2)`
     qui renvoie la somme des deux fractions passées en argument
   - `struct fraction sub_frac(struct fraction f1, struct fraction f2)`
     qui renvoie la différence des deux fractions passées en argument
   - `struct fraction mult_frac(struct fraction f1, struct fraction f2)`
     qui renvoie le produit des deux fractions passées en argument
   - `struct fraction div_frac(struct fraction f1, struct fraction f2)`
     qui renvoie le quotient des deux fractions passées en argument
   - `int frac_eq(struct fraction f1, struct fraction f2)`
     qui renvoie si deux fractions sont égales.
   - `float to_float(struct fraction f)`
     qui renvoie un entier approximant la fraction passée en argument
   - `struct fraction of_int(int i)`
     qui renvoie la fraction irréductible correspondant à l'entier passé en argument.

   Exportez ensuite leur définition dans le fichier `fractions.h`. Ce fichier devra contenir la directive `#include "types_numeriques.h"`.
6. Ajouter la directive `#include "fractions.h"` dans le fichier `fractions.c`. A-t-on toujours besoin de la directive `#include "types_numeriques.h"` dans `fractions.c` ?

1. Si le dénominateur est 0, la forme irréductible peut être $\frac{1}{0}$, $\frac{0}{0}$ ou $\frac{-1}{0}$.

7. La *Méthode de Héron* permet de calculer des approximations rationnelles des racines carrées d'entiers. Elle se base sur le fait que la suite U_n définie par :

$$\begin{aligned} U_0 &= 1 \\ U_{n+1} &= \frac{U_n + \frac{A}{U_n}}{2} \end{aligned}$$

tend vers $\sqrt{A}$ quand n tend vers $+\infty$.
Dans un fichier `heron.c`, écrire une fonction `struct fraction heron(int a, int n)` qui renvoie le n-ième terme de la suite de Héron pour un entier a. Vous utiliserez le type `fraction` pour faire tous les calculs. Quel(s) fichier(s) "`.h`" faut-il importer ?

8. Écrire une fonction main [2] qui demande à l'utilisateur deux entiers a et n, et qui affiche les n premiers termes de la suite de Héron paramétrée par a. Que constatez-vous ?
`heron.c`

9. (**bonus**) Écrire une fonction `struct fraction heron_mieux(int a, int n)` qui renvoie le n-ième terme de la suite de Héron paramétrée par a et initialisée avec $\lfloor\sqrt{a}\rfloor$.

2 Types numeriques abstraits (avec enum, union)

En `C`, lorsqu'on effectue une division sur des entiers, le résultat renvoyé est un entier. Ici, on cherche à faire en sorte que la division de deux entiers nous renvoie une `fraction`.

Exercice 2

1. Dans le fichier `types_numeriques.h` définir un type union nommé `valeur` qui peut être soit un `long long`, un `float` ou un `struct fraction`.
2. Afin de nous "souvenir" quel est le type d'un objet de type `union valeur`, nous allons l'encapsuler dans une structure `num`, qui a deux champs : un champ `val` qui contiendra la `valeur` et un champ `t` qui contiendra le type de la valeur. Quels sont les types de ces deux champs ? Écrire la définition du type `num` [3].
3. Dans un fichier `nums.c` [4] écrire une fonction `affiche_num` qui affiche un `num`.
4. Écrire une fonction `struct num retype(struct num n)` qui retype l'argument :
   - Si sa valeur est un `struct fraction`, et que celle-ci a un dénominateur égal à 1, alors il est reconverti en un `num` dont la valeur est un `long long`. Sinon, la fraction est réduite. Enfin, si le dénominateur ou la valeur absolue du numérateur est supérieure à 2.10^9, alors il est converti en un `num` dont la valeur est un `float`. Cela en vue d'éviter un débordement d'entier [5].
   - Si sa valeur est un `float` tel que `(float) ((int) f) == f` alors, il est reconverti en un `num` dont la valeur est un entier.
   - Si sa valeur est un `long long`, alors on ne fait rien.
5. Dans `nums.c` écrire les fonctions d'addition, de soustraction, de multiplication et de division sur les `num`. Exportez ces définition dans `nums.h`.
6. Modifiez votre fichier `heron.c` [6] pour qu'il utilise les fonctions sur les nums au lieu des fonctions sur les fractions.

2. dans le fichier `heron.c`, qui devra être compilé avec `gcc heron.c fractions.o -o heron` sachant que `fractions.o` aura été obtenu avec la commande `gcc -c fractions.c`

3. `num` doit vous évoquer "type numérique abstrait", rien à voir avec `enum` qui doit vous évoquer le terme anglais "enumeration"

4. Qui sera compilé avec `gcc nums.c fractions.o -o nums`

5. On rappelle qu'un `long long` est codé sur 64 bits, il prend donc des valeurs entre -2^{63} et $2^{63}-1$, si tous les entiers sont inférieurs (en valeur absolue) à $2.10^9 \approx 2^{31}$, alors on est assuré que le produit de deux entiers sera inférieur (en valeur absolue) à 2^{62} et que la somme de deux tels produits sera inférieur (en valeur absolue) à 2^{63}, qu'ainsi nous n'aurons pas de débordements d'entiers.

6. Il devra maintenant être compilé avec `gcc heron.c nums.o fractions.o -o heron`

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 follow the directions. If you are a first-time user, type `C-h t` for a **tutorial**.

remove help window	`C-x 1`
scroll help window	`C-M-v`
apropos: show commands matching a string	`C-h a`
describe the function a key runs	`C-h k`
describe a function	`C-h f`
get mode-specific information	`C-h m`

Error Recovery

abort partially typed or executing command	`C-g`
recover files lost by a system crash	`M-x recover-session`
undo an unwanted change	`C-x u`, `C-_` or `C-/`
restore a buffer to its original contents	`M-x revert-buffer`
redraw garbaged screen	`C-l`

Incremental Search

search forward	`C-s`
search backward	`C-r`
regular expression search	`C-M-s`
reverse regular expression search	`C-M-r`
select previous search string	`M-p`
select next later search string	`M-n`
exit incremental search	`RET`
undo effect of last character	`DEL`
abort current search	`C-g`

Use `C-s` or `C-r` again to repeat the search in either direction. If Emacs is still searching, `C-g` cancels only the part not matched.

© 2010 Free Software Foundation, Inc. Permissions on back.

Motion

entity to move over	**backward**	**forward**
character	`C-b`	`C-f`
word	`M-b`	`M-f`
line	`C-p`	`C-n`
go to line beginning (or end)	`C-a`	`C-e`
sentence	`M-a`	`M-e`
paragraph	`M-{`	`M-}`
page	`C-x [`	`C-x ]`
sexp	`C-M-b`	`C-M-f`
function	`C-M-a`	`C-M-e`
go to buffer beginning (or end)	`M-<`	`M->`

scroll to next screen	`C-v`
scroll to previous screen	`M-v`
scroll left	`C-x <`
scroll right	`C-x >`
scroll current line to center of screen	`C-u C-l`

Killing and Deleting

entity to kill	**backward**	**forward**
character (delete, not kill)	`DEL`	`C-d`
word	`M-DEL`	`M-d`
line (to end of)	`M-0 C-k`	`C-k`
sentence	`C-x DEL`	`M-k`
sexp	`M-- C-M-k`	`C-M-k`

kill **region**	`C-w`
copy region to kill ring	`M-w`
kill through next occurrence of *char*	`M-z` *char*
yank back last thing killed	`C-y`
replace last yank with previous kill	`M-y`

Marking

set mark here	`C-@` or `C-SPC`
exchange point and mark	`C-x C-x`
set mark *arg* **words** away	`M-@`
mark **paragraph**	`M-h`
mark **page**	`C-x C-p`
mark **sexp**	`C-M-@`
mark **function**	`C-M-h`
mark entire **buffer**	`C-x h`

Query Replace

interactively replace a text string	`M-%`
using regular expressions	`M-x query-replace-regexp`

Valid responses in query-replace mode are

replace this one, go on to next	`SPC`
replace this one, don't move	`,`
skip to next without replacing	`DEL`
replace all remaining matches	`!`
back up to the previous match	`^`
exit query-replace	`RET`
enter recursive edit (`C-M-c` to exit)	`C-r`

Multiple Windows

When two commands are shown, the second is a similar command for a frame instead of a window.

delete all other windows	`C-x 1`	`C-x 5 1`
split window, above and below	`C-x 2`	`C-x 5 2`
delete this window	`C-x 0`	`C-x 5 0`
split window, side by side	`C-x 3`	
scroll other window	`C-M-v`	
switch cursor to another window	`C-x o`	`C-x 5 o`
select buffer in other window	`C-x 4 b`	`C-x 5 b`
display buffer in other window	`C-x 4 C-o`	`C-x 5 C-o`
find file in other window	`C-x 4 f`	`C-x 5 f`
find file read-only in other window	`C-x 4 r`	`C-x 5 r`
run Dired in other window	`C-x 4 d`	`C-x 5 d`
find tag in other window	`C-x 4 .`	`C-x 5 .`
grow window taller	`C-x ^`	
shrink window narrower	`C-x {`	
grow window wider	`C-x }`	

Formatting

indent current **line** (mode-dependent)	`TAB`
indent **region** (mode-dependent)	`C-M-\`
indent **sexp** (mode-dependent)	`C-M-q`
indent region rigidly *arg* columns	`C-x TAB`
insert newline after point	`C-o`
move rest of line vertically down	`C-M-o`
delete blank lines around point	`C-x C-o`
join line with previous (with arg, next)	`M-^`
delete all white space around point	`M-\`
put exactly one space at point	`M-SPC`
fill paragraph	`M-q`
set fill column to *arg*	`C-x f`
set prefix each line starts with	`C-x .`
set face	`M-o`

Case Change

uppercase word	`M-u`
lowercase word	`M-l`
capitalize word	`M-c`
uppercase region	`C-x C-u`
lowercase region	`C-x C-l`

The Minibuffer

The following keys are defined in the minibuffer.

complete as much as possible	`TAB`
complete up to one word	`SPC`
complete and execute	`RET`
show possible completions	`?`
fetch previous minibuffer input	`M-p`
fetch later minibuffer input or default	`M-n`
regexp search backward through history	`M-r`
regexp search forward through history	`M-s`
abort command	`C-g`

Type `C-x ESC ESC` to edit and repeat the last command that used the minibuffer. Type `F10` to activate menu bar items on text terminals.

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`
regexp 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 *c*	`\`*c*
alternative ("or")	`|`
grouping	`\( ... \)`
same text as *n*th group	`\`*n*
at word break	`\b`
not at word break	`\B`

entity	**match start**	**match end**
line	`^`	`$`
word	`\<`	`\>`
buffer	`\‘`	`\’`
class of characters	**match these**	**match others**
explicit set	`[ ... ]`	`[^ ... ]`
word-syntax character	`\w`	`\W`
character with syntax *c*	`\s`*c*	`\S`*c*

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 *n*th menu item by number (1–9)	*n*
follow cross reference (return with `l`)	`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 regexp	`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/software/emacs/#Manuals`