IN 102 - Examen
24 janvier 2017
François Pessaux

Énoncé

ENSTA ParisTech

— **Lisez attentivement les consignes et tout le sujet avant de commencer.**

— Les documents (polys, transparents, TDs, livres ...) sont autorisés.
— Vous avez bien entendu accès à la commande `man` qui permet d'obtenir des informations sur les commandes Unix, mais aussi les fonctions de la bibliothèque standard de C (section 3 – ex : `man 3 printf`).
— Sont **absolument interdits** : le WEB, le courrier électronique, les messageries diverses et variées, le répertoire des camarades, le téléphone (même pour avoir l'heure puisque vous l'avez sur votre ordinateur).

— Votre travail sera (en partie) évalué par un mécanisme automatique. Vous **devez** respecter les règles de **nommage** des fichiers et autres **consignes** qui vous sont données.
— **Sauf indications contraires**, vos programmes doivent **gérer** les cas **d'erreur** pouvant survenir. Il vous est demandé de **respecter** la convention C concernant la **valeur retournée** par vos `main`'s ($0 \equiv$ OK, $\neq 0 \equiv$ KO).
— Lorsqu'il vous est demandé que votre programme réponde en affichant « **Yes** » ou « **No** », il ne doit **rien** afficher d'autre, et **pas** « Oui » ou « Yes. » ou « no » ou « La réponse est : no ».
Seuls les **messages d'erreurs** sont autorisés en plus et leur contenu est libre. Donc pensez à retirer vos affichages de test / debug.
— L'exercice 4 n'est pas à faire, il sert juste à vérifier que vous avez bien suivi la consigne de lire le sujet comme il vous l'est toujours demandé. Néanmoins, si vous trouvez la solution, vous serez gratifié d'un point supplémentaire.
— **Indentez** votre code afin que sa lecture ne soit pas un calvaire pour le correcteur ! En `Python` vous y étiez techniquement obligés, en C vous y êtes moralement obligés. La **lisibilité** de vos programmes sera **prise en compte** dans l'évaluation.

— **À la fin de l'examen**, vous devrez créer une **archive** contenant **tous** les fichiers **sources** que vous avez écrits (`.c`, `.h`). Le nom de cette archive devra avoir la structure suivante :
`nom_prenom.zip` ou `.tgz` (selon l'outil d'archivage que vous utilisez).
Par exemple, Donald Duck nommera son archive `duck_donald.zip`.
— Vous devrez **copier** cette archive dans répertoire de rendu se trouvant à `~pessaux/in102rendus/`
Par exemple, le canard ci-dessus remettra son examen en invoquant la commande : `cp -vi duck_donald.zip ~pessaux/in102rendus/`.
— **N'oubliez pas** d'effectuer cette copie sinon nous devrons considérer que vous n'avez rien rendu !

— Le sujet comporte **6** pages et l'examen dure **3** heures.
— Le barème est **volontairement** approximatif.

1 Conversion binaire → décimal ($\sim 15\%$)

Écrivez un programme qui prend en **ligne de commande** un argument représentant l'écriture en binaire (base 2) d'une valeur entière **positive** et affiche en retour cette valeur en décimal (base 10).

Note : On ne cherchera **pas** à gérer les débordements dans le cas où l'argument représente une valeur ne tenant pas sur un `int` non signé.

Note : Par contre, **on s'assurera** que la chaîne reçue en argument ne comporte bien **que des chiffres binaires**.

Nommage : Le fichier source de ce programme devra s'appeler `from_bin.c`.

Format de sortie : Uniquement la valeur numérique provenant de la conversion **suivie d'un retour à la ligne**.

Ex. tests :

- `./from_bin.x 101` ⟶ 5
- `./from_bin.x 10010011` ⟶ 147
- `./from_bin.x 11111111111111111111111111111111` ⟶ 4294967295
- `./from_bin.x 1110010` ⟶ 114
- `./from_bin.x` ⟶ ≪Error : expecting only one argument. ≫

2 Concaténation de chaînes de caractères ($\sim 30\%$)

On souhaite écrire un programme qui prend en **ligne de commande** des chaînes représentant les mots d'une phrase à reconstruire par juxtaposition. Comme dans toute phrase, les mots devront être séparés par un espace. Cette phrase finale devra être **affichée**.

Attention : Vous devez **explicitement** construire la chaîne de caractères représentant la **phrase complète** ! En aucun cas vous ne vous contenterez d'afficher chaque mot un par un directement depuis la tableau des arguments de votre `main` !

Nommage : Le fichier source de ce programme devra s'appeler `str.c`.

Format de sortie : Uniquement la phrase finale **suivie d'un retour à la ligne**.

Ex. tests :

- `./str.x` ⟶ (rien à afficher)
- `./str.x Ceci est un test` ⟶ `Ceci␣est␣un␣test`
- `./str.x Ceci est un test.` ⟶ `Ceci␣est␣un␣test.`
- `./str.x é 6 h -` ⟶ `é␣6␣h␣-`

Rappel : Vous avez à votre disposition les fonctions de la bibliothèque standard de C (nécessitant `string.h`) :

- `strcpy (char *dest, char *src)` qui copie la chaîne `src` dans la chaîne `dest` (note : pas vraiment nécessaire pour cet exercice).
- `strcat (char *dest, char *src)` qui ajoute en fin de la chaîne `dest` une copie de la chaîne `src`.
- `int strlen (char *str)` qui retourne la ≪longueur≫ de la chaîne `str`.

3 Mise en forme irréductible d'une fraction ($\sim 25\%$)

Il vous est donné, dans le fichier `fract.h`, des types permettant de représenter des fractions, comme dans le TD n°4 d'IN102.

```
------------- fract.h -------------
```

```
#ifndef __FRACT_H__
#define __FRACT_H__

enum sign_t { S_pos, S_neg };

struct fraction_t {
  enum sign_t sign ;
  unsigned int num ;
  unsigned int denom ;
};

#endif
```

Vous devrez réaliser un programme qui permet de saisir au **clavier** une fraction, calcule sa forme **irréductible** et **affiche** cette dernière.

- L'utilisateur devra rentrer :
  1. un signe (**caractère** '+' ou '-'),
  2. le numérateur,
  3. le dénominateur.

  chacun séparé par un (des) espaces (ou retour à la ligne, c'est la même chose pour `scanf`).
- À l'issue de cette saisie, une **structure de fraction** doit être créée et passée en **argument** à une **fonction** (`irreducible` par exemple) qui calcule la fraction réduite et la **retourne**.
- La fonction principale affichera alors la fraction réduite.

Note : La mise en forme irréductible passe par le calcul du **PGCD** des numérateur et dénominateur. Le calcul de ce PGCD est spécifié par :

$$PGCD(a,b) = \begin{cases} a & \text{si } a = b \\ PGCD(a-b,b) & \text{si } a > b \\ PGCD(a,b-a) & \text{si } a < b \end{cases}$$

Nommage : Le fichier source de ce programme devra s'appeler `fract.c`.

Format de sortie : Uniquement la fraction réduite, sous la forme `signe numérateur/dénominateur` **suivie d'un retour à la ligne**.

Ex. tests :

- `./fract.x`
  `-`
  `2`
  `3`
  $\longrightarrow$ `-2/3`
- `./fract.x`
  `+ 45 125`
  $\longrightarrow$ `9/25`

4 Stockage en mémoire ($\sim 20\%$)

Est-ce que deux vaches mortes (ou deux bœufs morts) tiennent sur 32 bits ? Vous argumenterez votre réponse.

Nommage : Vous répondrez dans un fichier **texte** nommé `stockage.txt`.

5 Transposition d'une matrice carrée ($\sim 30\%$)

Vous devez écrire un programme calculant et affichant la **transposée** d'une matrice **5×5** d'**entiers**. Les coefficients de la matrice seront rentrés au clavier par l'utilisateur.

Attention : vous ne devrez **pas** utiliser de matrice **temporaire**, autrement dit, vous modifierez **directement** la (**seule**) matrice présente dans votre programme. De plus, votre programme doit être **trivialement modifiable** pour fonctionner avec des matrices 15×15, 40×40, 100×100...

Note : afin de vous éviter de saisir 25 valeurs pour tester votre programme, 3 fichiers texte de données vous sont fournis (`mat1.dat`, `mat2.dat`, `mat3.dat`). Vous pourrez injecter leur contenu à partir du mécanisme de redirection (`<`) du terminal (c.f. exemples ci-dessous en cas de trou de mémoire de `MO101`).

Nommage : Le fichier source de ce programme devra s'appeler `transp.c`.

Format de sortie : Uniquement la matrice transposée, chaque ligne séparée de la suivante par **un retour à la ligne**, avec aussi **un retour à la ligne** pour la dernière ligne. Chaque ligne sera composée de ses coefficients séparés par **1 espace**.

Ex. tests : (regardez le contenu des fichiers `*.dat`)

— `transp.x < mat1.dat` ⟶

```
0 5 10 15 20
1 6 11 16 21
2 7 12 17 23
3 8 13 18 23
4 9 14 19 24
```

— `transp.x < mat2.dat` ⟶

```
44 0 17 12 39
35 4 35 1 43
32 20 49 6 14
41 21 28 28 23
39 20 15 19 23
```

— `transp.x < mat3.dat` ⟶

```
44 49 47 20 43
46 12 47 49 20
3 5 10 9 46
0 11 33 20 4
20 7 13 34 18
```

—— Fin du sujet ——