

Compte Rendu du Projet Cowsay

Aleksandr Shmigelskii, Gabriel Mella, Daniel Bass
IMA-05

Enseignant de TD/TP : Jean-Loup Haberbusch

Table des matières

1	Introduction	2
2	Préliminaires	3
3	Bash	4
3.1	cow_kindergarten	4
3.2	cow_primaryschool	5
3.3	cow_highschool	6
3.4	cow_college	7
3.5	cow_university	8
3.6	smart_cow	9
3.7	crazy_cow	10
4	C	11
4.1	newcow	11
4.2	wildcow	12
4.3	reading_cow	13

1. Introduction

2. Préliminaires

Option	Signification / Effet	Exemple d'usage
-b	Borg mode : la vache aura un aspect "cyborg".	<code>cowsay -b "Je suis un Borg"</code>
-d	Dead mode : la vache a des yeux « XX ».	<code>cowsay -d "Aïe. Je ne me sens pas bien"</code>
-g	Greedy mode : la vache a des yeux « \$\$ ».	<code>cowsay -g "J'adore l'argent"</code>
-p	Paranoïd mode : la vache a des yeux « @@ ».	<code>cowsay -p "Je suis surveillé..."</code>
-s	Stoned mode : la vache a des yeux « ** ».	<code>cowsay -s "Coucou..."</code>
-t	Tired mode : la vache a des yeux « - ».	<code>cowsay -t "Je suis épuisée..."</code>
-w	Wired mode : la vache a des yeux « OO ».	<code>cowsay -w "Je ne tiens plus en place"</code>
-y	Youthful mode : la vache a des yeux « .. ».	<code>cowsay -y "Je suis toute jeune"</code>
-e <i>eyes</i>	Personnalise les yeux (2 caractères).	<code>cowsay -e ^o "Regarde mes yeux"</code>
-T <i>tongue</i>	Personnalise la langue (1 ou 2 caractères).	<code>cowsay -T "U" "Ma langue est sortie"</code>
-f <i>cowfile</i>	Utilise un autre dessin ASCII (fichier .cow).	<code>cowsay -f small "Vraiment petite!"</code>
-r	Choisit une vache au hasard (fichier .cow).	<code>cowsay -r "Je suis une vache aléatoire."</code>
-l	Liste les vaches définies dans le chemin COWPATH	<code>cowsay -l</code>

TABLE 1 – Principales options de `cowsay`

3. Bash

3.1. cow_kindergarten

Fonctionnalité Ce script fait « dire » à la vache les nombres de 1 à 10 de manière animée :

- `clear` efface l'écran avant chaque itération pour simuler une animation.
- `cowsay $i` affiche le chiffre courant (`i` parcourant `{1..10}`).
- `sleep 1` introduit une pause d'une seconde entre chaque affichage.
- À la fin, `cowsay -T U "J'ai terminé!"` fait tirer la langue à la vache.

Exemple d'exécution

```
$ ./cow_kindergarten.sh

  ---
< >
  ---
      \   ^__^
      \  (oo)\_____
         (__)\\       )\\/\\
            ||----w |
            ||     ||

...
  ---
< 9 >
  ---
      \   ^__^
      \  (oo)\_____
         (__)\\       )\\/\\
            ||----w |
            ||     ||

-----
<J'ai terminé!>
-----
      \   ^__^
      \  (oo)\_____
         (__)\\       )\\/\\
            U ||----w |
               ||     ||
```

Commentaires

- Utilisation d'une boucle `for i in {1..10}` appelant `cowsay` à chaque itération.
- **Effet d'animation** : `clear` + `sleep 1` suffisent, pas besoin d'outils externes.

Le code source est fourni dans l'archive (`scripts/cow_kindergarten.sh`).

3.2. cow_primaryschool

Différences principales par rapport à cow_kindergarten

- Le nombre d'itérations est désormais fixé par l'argument \$1.
- Vérification qu'un seul argument est fourni et qu'il est strictement positif :
 - si \$# -ne 1, on affiche un message d'usage et on quitte ;
 - si \$1 -le 0, on affiche via cowsay « Veuillez fournir un nombre entier positif supérieur à 0 » puis on quitte.
- Boucle while [\$CMPT -le \$N] remplace la boucle fixe {1..10}.

Exemples d'exécution

```
$ ./cow_primaryschool.sh 5
```

```
---
< 5 >
---
  \  ^__^
    \ (oo)\_____
      (__)\       )\/\
         ||----w |
         ||     ||
```

```
$ ./cow_primaryschool.sh -5
```

```
-----
/ Veuillez fournir un nombre entier \
\ positif supérieur à 0.              /
-----
  \  ^__^
    \ (oo)\_____
      (__)\       )\/\
         ||----w |
         ||     ||
```

```
$ ./cow_primaryschool.sh 5 7
```

```
Usage: ./cow_primaryschool.sh <nombre n>
```

Commentaires

- Le test [\$1 -le 0] intercepte les valeurs non-positives et utilise cowsay pour un message d'erreur plus lisible.
- En cas d'erreur, le script quitte immédiatement sans exécuter la boucle principale.
- Le test [\$1 -le 0] couvre les entiers non positifs, mais si \$1 n'est pas numérique, Bash renvoie une erreur de syntaxe (operand expected), mais nous avons supposé que seuls des chiffres seraient transmis.

Le code complet est disponible dans l'archive (scripts/cow_primaryschool.sh).

3.3. cow_highschool

Différences principales par rapport à cow_primaryschool

- Au lieu d'énoncer simplement i , la vache énonce son carré i^2 grâce à :
`cowsay $((CMPT * CMPT))`
- La structure générale (vérification d'argument, boucle, `clear`, `sleep`) reste identique.

Exemples d'exécution

```
$ ./cow_highschool.sh 10
```

```
-----
< 36 >
-----
      \   ^__^
       \  (oo)\_____
          (__)\\       )\/\
              ||----w |
              ||     ||
```

```
$ ./cow_highschool.sh -5
```

```
-----
/ Veuillez fournir un nombre entier \
\ positif supérieur à 0.              /
-----
      \   ^__^
       \  (oo)\_____
          (__)\\       )\/\
              ||----w |
              ||     ||
```

```
$ ./cow_highschool.sh
```

```
Usage: ./cow_highschool.sh <nombre n>
```

Commentaires

- Le calcul du carré utilise l'arithmétique intégrée de Bash (`$((...))`).
- La validation écrite `[$1 -le 0]` couvre les valeurs non-positives, mais un argument non numérique génère une erreur Shell (« operand expected ») non gérée.

Le script complet se trouve dans l'archive (`scripts/cow_highschool.sh`).

3.4. cow_college

Fonctionnalité Ce script énonce les termes de la suite de Fibonacci strictement inférieurs à n :

- Vérification de l'argument : un entier > 1 est requis (`[$1 -le 1]`).
- Initialisation de deux variables `FIB1=1, FIB2=1`.
- Boucle `while [$FIB1 -lt $N]` :
 - Affichage de `cowsay $FIB1`.
 - Calcul du terme suivant via `NEW=$((FIB1+FIB2))`, décalage `FIB1=$FIB2, FIB2=$NEW`.
 - `sleep 1 + clear` pour l'animation.
- Fin marquée par `cowsay -T U "J'ai terminé!"`.

Exemples d'exécution

```
$ ./cow_college.sh 10
```

```
---
< 1 >
---
  \  ^__^
   \ (oo)\_______
      (__)\       )\/\
         ||----w |
         ||     ||
```

...

```
---
< 8 >
---
  \  ^__^
   \ (oo)\_______
      (__)\       )\/\
         U ||----w |
         ||     ||
```

```
$ ./cow_college.sh 1
```

```
-----
/ Veuillez fournir un nombre entier \
\ positif supérieur à 1.              /
-----
```

```
  \  ^__^
   \ (oo)\_______
      (__)\       )\/\
         ||----w |
         ||     ||
```

```
$ ./cow_college.sh
```

```
Usage: ./cow_college.sh <nombre n>
```

Commentaires

- On ne stocke que deux variables `FIB1, FIB2`.
- La condition `while [$FIB1 -lt $N]` garantit de n'afficher que les termes strictement inférieurs à N , et stoppe avant le premier terme $\geq N$.
- Validation minimale : un argument non numérique déclenchera une erreur de shell non gérée.

Le code complet est disponible dans l'archive (`scripts/cow_college.sh`).

3.5. cow_university

- Fonctionnalité** Ce script énonce tous les nombres premiers strictement inférieurs à n :
- Vérification de la présence d'un argument unique et de sa positivité (`[$1 -le 1]`).
 - Boucle `CMPT=2` à `CMPT<$N` :
 - Initialisation de `isPrime=1` (on suppose premier).
 - Boucle interne `while [$i -lt $CMPT]` testant `$CMPT % $i`.
 - Si un diviseur est trouvé, `isPrime=0` et `break`.
 - Si `isPrime==1`, appel de `cowsay $CMPT`, `sleep 1`, `clear`.
 - Fin de l'exercice marquée par `cowsay -T U "J'ai terminé!"`.

Exemples d'exécution

```
$ ./cow_university.sh 20
```

```
---
< 2 >
---
      \  ^__^
      \  (oo)\_______
          (__)\       )\/\
              ||----w |
              ||     ||
```

```
3 ... 19
```

```
      \  ^__^
      \  (oo)\_______
          (__)\       )\/\
              ||----w |
              ||     ||
```

```
$ ./cow_university.sh -5
```

```
-----
/ Veuillez fournir un nombre entier \
\ positif supérieur à 1.              /
-----
      \  ^__^
      \  (oo)\_______
          (__)\       )\/\
              ||----w |
              ||     ||
```

Commentaires

- Algorithme naïf de test de primalité en $O(n^2)$, testant tous les diviseurs jusqu'à `CMPT-1`.
- Interruption précoce dès qu'un diviseur est trouvé (`break`) pour limiter les calculs.
- Validation minimale : un argument non numérique déclenche une erreur de shell non gérée.

Le script complet est disponible dans l'archive (`scripts/cow_university.sh`).

3.6. smart_cow

Fonctionnalité Le script évalue une expression arithmétique simple (addition, soustraction, multiplication, division) passée en argument et affiche le résultat dans les yeux de la vache :

- Vérification qu'un seul argument (la chaîne d'expression) est fourni.
- **Subshell silencieux + redirection** (`(res=$((expr))) 2>/dev/null`) pour tester la validité de l'expression sans polluer l'écran.
- Inspection du code de retour (`$?`) : si $\neq 0$, message d'erreur `cowsay "Expression invalide : $expr"` et sortie.
- Re-calcul du résultat hors subshell (`res=$((expr))`) pour récupérer la valeur.
- Détermination de la forme des yeux selon la longueur du résultat :
 - 1 chiffre \rightarrow `eyes="$res"`
 - 2 chiffres \rightarrow `eyes="$res"`
 - >2 chiffres \rightarrow `eyes="??"` + message d'excuse.
- Affichage final `cowsay -e "$eyes" "$msg"`.

Exemples d'exécution

```
$ ./smart_cow.sh "3+11"
```

```
< Le résultat de 3+11 est 14 >
```

```
-----
\      ^__^
\    (14)\_______
      (__)\\       )\/\
           ||----w |
           ||     ||
```

```
$ ./smart_cow.sh "3+11+"
```

```
< Expression invalide : 3+11+ >
```

```
-----
\      ^__^
\    (oo)\_______
      (__)\\       )\/\
           ||----w |
           ||     ||
```

```
$ ./smart_cow.sh 3 + 11
```

```
Usage: ./smart_cow.sh "<expression>"
```

Commentaires et difficultés

- Capturer l'erreur d'arithmétique Bash nécessite un subshell et `2>/dev/null`, car `$((...))` renvoie un code $\neq 0$ mais affiche aussi un message sur stderr.
- Refaire le calcul hors subshell est la solution la plus simple pour récupérer `$res`.
- Pas de test explicite sur les caractères de l'expression : si on entre une chaîne non arithmétique, elle est évaluée à 0 sans message d'erreur.
- Gestion des cas « yeux trop petits » et format dynamique des yeux selon la longueur du résultat.

Le script complet est disponible dans l'archive (`scripts/smart_cow.sh`).

3.7. crazy_cow

Fonctionnalité Ce script applique une suite d'opérations arithmétiques successives, de gauche à droite, à partir d'un nombre initial. Il affiche à chaque étape une vache avec le résultat intermédiaire dans les yeux et change son comportement si un seuil est dépassé.

- Le script attend un argument initial suivi d'un ou plusieurs couples <opérateur> <valeur>.
- Il utilise `shift` pour traiter dynamiquement les arguments deux par deux.
- À chaque itération :
 - vérification de l'opérateur (+, -, *, /, %);
 - vérification que la valeur est bien un entier;
 - tentative de calcul (`result $op val`) avec gestion d'erreur comme dans `smart_cow`.
- En cas de dépassement du seuil (`THRESHOLD=100`), la vache devient folle :
 - affichage d'un message de délire (`eyes = ??, @@, etc.`),
 - puis mort finale avec l'option `-d`.
- Si le résultat est :
 - négatif → yeux `XX`,
 - nul → yeux `??`,
 - normal → yeux `oo`.

Exemple d'exécution

```
$ ./crazy_cow.sh 5 + 2 - 100 + 3 \* 10
...
< Résultat après -93 + 3 : -90 >
      \   ^__^
       \  (XX)\_____

$ ./crazy_cow.sh 50 + 40 + 30
...
< Aaaaarg, c'en est trop ! Je meurs... >
      \   ^__^
       \  (xx)\_____

$ ./crazy_cow.sh 5 + 2 - 100 = + 3 "*" 10
...
< Opérateur invalide : = >
      \   ^__^
       \  (00)\_____

$ ./crazy_cow.sh 5 + 2 - 100 + + 3 "*" 10
< Oups, impossible de calculer : -93 + + >
      \   ^__^
       \  (oo)\_____
```

Commentaires et difficultés

- Cette version visait à proposer un script plus original, avec un traitement dynamique des arguments via `shift` dans une boucle.
- Nécessité d'échapper l'astérisque `*` en ligne de commande (`*` ou `"*"`), car sinon le shell tente de l'expanser (globbing) en cherchant les fichiers du répertoire courant.
- Vérification élémentaire des erreurs d'entrée via un sous-shell et redirection `2>/dev/null` pour ne pas afficher les messages Bash.

Le script complet est fourni dans l'archive (`scripts/crazy_cow.sh`).

4. C

4.1. newcow

Fonctionnalité Cette version du programme `newcow.c` permet à l'utilisateur d'adapter dynamiquement l'apparence de la vache ASCII grâce à plusieurs options :

- `-e` ou `-eyes` : personnalise les yeux (exactement 2 caractères attendus),
- `-T` ou `-tongue` : personnalise la langue (1 ou 2 caractères autorisés),
- `-tail <n>` : ajoute `n` répétitions du motif `\ /` pour allonger la queue,
- `-legs <n>` : ajoute `n` lignes aux jambes verticales.

Exemples d'exécution

```
$ ./newcow
  \      ^__^
   \    (oo)\_____
      (__) \        )\/
           ||----w |
           ||     ||

$ ./newcow -e "0o" -T "u" --tail 4 --legs 3
  \      ^__^
   \    (0o)\_____
      (__) \        )\\/\//\//
           u ||----w |
           ||     ||
           ||     ||
           ||     ||
           ||     ||
```

Commentaires

- L'apparence de la vache est construite ligne par ligne à l'aide de `printf()` — chaque ligne du dessin est imprimée manuellement, ce qui permet une insertion dynamique de variables (yeux, langue, etc.).
- L'analyse des options repose sur des fonctions standards de `<string.h>` :
 - `strcmp(const char *s1, const char *s2)` : compare deux chaînes. Utilisée ici pour vérifier si un argument correspond à une option (ex : `-e`, `-tail`, etc.).
 - `strlen(const char *s)` : retourne la longueur d'une chaîne (hors `\0`). Cela permet de valider que les options `-e` contiennent bien exactement 2 caractères et que `-T` ne dépasse pas 2.
 - `strcpy(char *dest, const char *src)` : copie le contenu de la chaîne `src` dans `dest`. Utilisée pour copier la valeur saisie pour les yeux ou la langue dans les variables locales `eyes` et `tongue`.
- `atoi(const char *str)` (issue de `<stdlib.h>`) : convertit une chaîne représentant un nombre entier (ex : `"3"`) en une valeur `int`. Cela est nécessaire pour transformer les arguments de `-tail` et `-legs` en entiers utilisables dans des boucles.
- Le paramètre `-tail` ajoute une séquence répétée `\ /` simulant une queue plus longue. Cette répétition est obtenue via une boucle `for` dynamique.
- Le paramètre `-legs` augmente la hauteur du dessin par ajout de lignes supplémentaires verticales sous le corps.

Pour exécuter, il suffit de compiler le programme (`gcc newcow.c -o newcow`) puis de le lancer dans un terminal avec : `./newcow`

Le script complet est fourni dans l'archive (`src/newcow.c`).

4.2. wildcow

Fonctionnalité Ce programme écrit en C simule une vache « animée » qui :

1. Cligne des yeux plusieurs fois tout en regardant vers la gauche.
2. Se retourne, puis avance progressivement vers la droite.
3. Broute de l’herbe en baissant la tête.
4. Se redresse, cligne à nouveau, et reste immobile.

L’ensemble est affiché directement dans le terminal à l’aide de commandes ANSI, sans dépendance externe.

Comportement détaillé

- La vache est affichée ligne par ligne grâce à des fonctions `affiche_vache()`, `affiche_vache_mirror()` et `affiche_vache_mirror_graze()`.
- Lors de l’animation, deux « états » des yeux sont utilisés (“oo” pour ouverts et “-” pour fermés), avec alternance visuelle.
- L’animation de la marche utilise un indicateur `leg_frame` pour modifier les jambes toutes les itérations.
- Lorsqu’elle broute, la vache abaisse sa tête et une vague d’animation simule un mouvement de la queue grâce à un tableau de chaînes pré-définies.

Fonctions techniques importantes

- `void update()` : efface l’écran et replace le curseur en haut à gauche (`\033[H\033[J`) pour simuler une animation image par image.
- `void gotoxy(int x, int y)` : positionne le curseur à la ligne `x`, colonne `y` à l’aide des séquences ANSI (American National Standards Institute) (`\033[%d;%dH`).
- `fflush(stdout)` : force l’affichage immédiat du contenu du buffer standard (`stdout`). Sans cela, les `printf()` peuvent être retardés ou ignorés par le système tant que la ligne n’est pas « pleine ». C’est indispensable dans le cadre d’une animation.
- `usleep()` : utilisé à la place de `sleep()` pour permettre des pauses en fractions de seconde. Cela rend l’animation fluide et plus réactive. `sleep()` ne prend en charge que des durées entières en secondes.

Originalité

- Le déplacement est réalisé par incrémentation de la colonne d’affichage.
- La queue de la vache est animée avec une « onde » grâce à un tableau de motifs ASCII cycliques.
- Les mouvements sont construits en simulant un nombre d’étapes précises, permettant un contrôle fin de la scène (ex. : position `last_col` mémorisée pour la phase de broutage).

Note sur les performances L’utilisation de `const char *` pour les chaînes fixes (yeux, queue) évite des copies mémoire et signale explicitement au compilateur que les pointeurs ne doivent pas être modifiés. Cela améliore la lisibilité et protège de certaines erreurs de manipulation mémoire.

Pour exécuter l’animation, il suffit de compiler le programme (`gcc wildcow.c -o wildcow`) puis de le lancer dans un terminal avec : `./wildcow`

Le script complet est fourni dans l’archive (`src/wildcow.c`).

4.3. reading_cow

Fonctionnalité Ce programme simule une vache qui « apprend à lire », caractère par caractère. Chaque nouveau caractère apparaît d’abord dans sa bouche, puis est ajouté dans la bulle de texte. Deux modes sont possibles :

- Si un fichier est donné en argument, le texte est lu depuis ce fichier.
- Sinon, une phrase est saisie par l’utilisateur dans le terminal (lecture depuis `stdin`).

Structure du code

- `update()` : efface le terminal (séquence ANSI `\033[H\033[J`).
- `print_bubble()` : dessine dynamiquement la bulle de texte autour des caractères déjà lus.
- `print_cow()` : affiche la vache avec le caractère en train d’être lu dans la bouche.
- `animate()` : cœur du programme qui anime la lecture à l’aide d’une boucle.

Taille des buffers

- On utilise deux tableaux statiques : `char src[1024]` et `char buffer[1024]`.
- La taille maximale est contrôlée par `total < (int)sizeof(src)` pour éviter tout débordement.
- Ce choix simple évite l’usage de la mémoire dynamique tout en permettant des textes relativement longs.

Exemple d’exécution

- Depuis un fichier texte :

```
$ ./reading_cow fich.cow
-----
< bonjo >
-----

      \  ^__^
       \ (oo)\_______
          (__)\       )\/\
             u ||----w |
               ||     ||
```
- Depuis une saisie manuelle (entrée standard) :

```
$ ./reading_cow
Tapez la chaîne de caractères: Bonjour!
...
```

Commentaires et difficultés

- La lecture d’un fichier permet de traiter des chaînes contenant des retours à la ligne (`\n`), mais ceux-ci apparaissent « bizarrement » dans l’animation.
- À l’inverse, la lecture depuis `stdin` est limitée à une seule ligne (lecture jusqu’à `\n`).
- La fonction `fflush(stdout)` garantit que le dessin est bien affiché à chaque étape, sans attendre que le buffer soit rempli.
- Enfin, `sleep(1)` donne le rythme d’animation, mais aurait pu être remplacé par `usleep()` pour plus de fluidité si on voulait aller plus loin.

Le fichier source est fourni dans l’archive (`src/reading_cow.c`), ainsi qu’un exemple de fichier texte `fich.cow` pour le test.