Compte rendu projet Cowsay

de Valicourt - Nonis - Groupe BG

Partie Bash

1-2)

Les deux premier scripts, cow_kindergarten.sh et cow_primaryschool.sh, il s'agit juste de faire une boucle simple et d'afficher une dernière fois la vache avec la langue qui pend.

3)

cow_highschool.sh est globalement toujours aussi simple que les deux premiers, on va juste cette fois-ci afficher les nombres 1,2,4... jusqu'à n^2 avec n donné en argument, comme demandé.

un exemple d'affichage final avec la commande `./cow_highschool 7` :

```
 ----
< 49 >
 ----
     \  ^__^
      \ (oo)\_______
        (__)\       )\/\
         U  ||----w |
            ||     ||
```

4)

Pour cow_college.sh, on reste encore une fois sur une boucle dont on va à la fin calculer le chiffre suivant de la suite de fibonacci.

La boucle s'arrête dès que le nombre dépasse le chiffre donné en argument, et affiche encore la vache avec la langue qui pend, avec le dernier chiffre de la suite inférieur au chiffre donné.

5)

Dans cow_university.sh, on va faire une boucle qui parcours les entiers de 1 à n-1 avec n donné en argument du script, et pour chacun de ces entiers on va supposer qu'ils sont premiers, puis on va entrer dans une seconde boucle qui va tester si le reste de la division entière de l'entier avec tous les nombres de 0 à (cet entier - 1).

Si une seule des divisions donne en reste 0, alors cet entier ne sera pas affiché, si toutefois aucun des nombres ne le divise, alors on l'affiche, et on le stocke dans une variable pour plus tard l'afficher si c'était le dernier nombre premier de 0 à n-1.

*voici un **extrait** du script cow_university.sh* :

```
while [ $nb -lt $1 ]
do
    prem=1                              # On suppose que le nombre est
premier
    for i in $(seq 2 $(expr $nb - 1))   # Et on va ensuite tester de le
diviser avec tous les nombres de 2 à lui-même -1
    do
        if [ $(expr $nb \% $i) -eq 0 ]  # Si il est divisible par au moins
un des nombres alors il n'est pas premier
        then
            prem=0
        fi
    done

    if [ $prem -eq 1 ]                  # On l'affiche si il est premier
    then
        echo " ----"
        echo "< $nb >"
        echo " ----"
        echo "     \  ^__^"
        echo "      \ (oo)\_______"
        echo "        (__)\       )\/\\"
        echo "            ||----w |"
        echo "            ||     ||"
        sleep 1
        clear
        der_nb=$nb
    fi

    nb=$(expr $nb + 1)
done
```

6)

Pour smart_cow.sh, on a réalisé une boucle qui va parcourir la chaîne de caractère donnée en argument : on récupère le caractère avec `char=${1:$i:1}` avec `i` l'index du caractère dans la chaîne.

On suppose ensuite que le caractère n'est pas un nombre (donc un opérateur), puis on teste son égalité avec les caractères des chiffres (de "0" à "9"), s'il match avec l'un des chiffres alors on prends (le chiffre précédemment trouvé)*10 + le chiffre trouvé, on forme alors ainsi de suite le nombre donné.

```
nb_temp=0

for i in $(seq 0 $(expr ${#1} - 1))
do
    char=${1:$i:1}

    est_nb=false            # On suppose que le caractère n'est pas un
nombre
```

```
    for j in $(seq 0 9)
    do
        if [ "$char" = "$j" ]   # Si il match avec un caractère de 0 à 9,
        then
            nb_temp=$(expr $nb_temp \* 10 + $j)     # On "l'ajoute" au
début du nombre calculé
            est_nb=true
        fi
    done
...(suite)
```

Maintenant si le caractère n'est pas un chiffre alors c'est forcément un opérateur, on va alors stocker l'opérateur dans une variable, et le nombre précédemment trouvé dans une autre variable, et mettre à zéro la variable qui stock le chiffre dans la première boucle.

```
...
if [ "$est_nb" = false ]    # C'est nécéssairement c'est un opérateur
    then
        mode=$char      # On définit le mode comme l'opération mathématique
        nb1=$nb_temp    # Et on stock le premier nombre trouvé dans une
variable
        nb_temp=0
    fi
done
```

La boucle principale va continuer et stocker dans `nb_temp` le second nombre jusqu'à finir la chaîne de caractère entrée en paramètres.

Il ne reste donc plus qu'à faire notre calcul, et afficher le tout, mais avant ça il faut faire attention à la multiplication, représentée par `*`, car c'est un métacaractère donc il faut ajouter un antislash avant pour faire le calcul avec `expr`.

Il faut aussi faire attention au cas où l'utilisateur rentre un calcul avec une division par 0, si tel est le cas on affiche un message d'erreur sans afficher la vache avec le calcul effectué

```
if [ "$mode" = "*" ]    # On doit gérer le cas de la multiplication
différemment car * est un métacaractère
then
    res=$(expr $nb1 \* $nb_temp)
elif [ "$mode" = "/" -a $nb_temp = "0" ]    # On fait attention au cas de
la division par 0 !
then
    echo Erreur de division par 0 !
    exit 1
else
    res=$(expr $nb1 $mode $nb_temp)
fi

if [ ${#res} -eq 1 ]    # Juste un petit fix, si jamais le résultat est un
```

```
seul chiffre, on ajoute un 0 devant pour ne pas décaler le dessin ascii de
la vache
then
    res=$(echo 0$res)   # Une variante serait de mettre un espace au lieu
du 0 mais je trouve que ca rend mieux avec un 0 plutôt qu'avec un espace
fi
```

exemples d'exécution :

```
suka@suka:~/code/cowsay/bash$ ./smart_cow.sh "6+13"
 __________
< 6 + 13 = >
 ----------
     \  ^__^
      \ (19)\_______
        (__)\       )\/\
            ||----w |
            ||     ||
suka@suka:~/code/cowsay/bash$ ./smart_cow.sh "16*48"
 __________
< 16 * 48 = >
 -----------
     \  ^__^
      \ (768)\_______
        (__)\       )\/\
            ||----w |
            ||     ||
suka@suka:~/code/cowsay/bash$ ./smart_cow.sh "376-135"
 _____________
< 376 - 135 = >
 -------------
     \  ^__^
      \ (241)\_______
        (__)\       )\/\
            ||----w |
            ||     ||
suka@suka:~/code/cowsay/bash$ ./smart_cow.sh "57/11"
 __________
< 57 / 11 = >
 -----------
     \  ^__^
      \ (05)\_______
        (__)\       )\/\
            ||----w |
            ||     ||
suka@suka:~/code/cowsay/bash$ ./smart_cow.sh "35/0"
Erreur de division par 0 !
```

7)

Pour le dernier script bash crazy_cow.sh, on a décidé de partir sur un chiffrement de césar, en utilisant donc la table ascii. C'était compliqué mais on est parvenu à le faire fonctionner. Il faut simplement impérativement backslash les guillemets et les dollars sinon il risque d'y avoir des problèmes.

On vérifie premièrement que l'utilisateur exécute le script avec deux arguments : un entier pour le décalage, et la chaîne de caractère qu'il veut chiffrer. Puis pour chaque caractère de la chaîne, on va simplement le convertir en sa valeur dans la table ascii avec `ascii_val=$(printf "%d" "'$char")`.

On additionne cette valeur avec le décalage donné avec `val=$(expr $ascii_val + $decalage)`, puis on fais attention à ce qu'il soit dans l'intervalle des caractère imprimables de la table ascii (espace compris).

```
if [ $val -gt 126 ]                         # 126 étant la borne supérieure
    then
        val=$(expr 31 + $(expr $val - 126))
    elif [ $val -lt 32 ]                    # et 32 la borne inférieure
    then
        val=$(expr 127 - $(expr 32 - $val))
    fi
```

Ensuite on ajoute le caractère chiffré à la chaîne de caractère du message chiffré :

```
encrypted_val=$(printf \\$(printf '%03o' $val))
encrypted_msg=$(echo "$encrypted_msg$encrypted_val")    # sorte de
concaténation
```

Il ne reste plus qu'a faire la même opération sur toute la chaîne de caractère passée en paramètre et on obtient la chaîne chiffrée avec le décalage souhaité.

Pour bien finir le tout, lorsqu'on exécute le script avec des arguments, la vache va d'abord penser au texte brut, puis au bout de 2 secondes va dire le texte chiffré avec le décalade dans ses yeux *(voir le script pour plus de détails)*

```
suka@suka:~/code/cowsay/bash$ ./crazy_cow.sh 6 "Bonjour tout le monde !"
 _____________________________
( Bonjour tout le monde ! )
 -------------------------
    o   ^__^
     o  (--)\_______
        (__)\       )\/\
            ||----w |
            ||     ||

*clear*
 _____________________________
< Hutpu{x&zu{z&rk&sutjk&' >
 -------------------------
    \   ^__^
```

```
      \  (06)\_______
         (__)\       )\/\
             ||----w |
             ||     ||
```

À savoir que l'opération inverse est totalement possible :

```
suka@suka:~/code/cowsay/bash$ ./crazy_cow.sh -6 "Hutpu{x&zu{z&rk&sutjk&'"
  ____________________________
( Hutpu{x&zu{z&rk&sutjk&' )
 --------------------------
    o   ^__^
     o  (--)\_______
        (__)\       )\/\
            ||----w |
            ||     ||

*clear*
 ____________________________
< Bonjour tout le monde ! >
 --------------------------
    \   ^__^
     \  (-6)\_______
        (__)\       )\/\
            ||----w |
            ||     ||
```

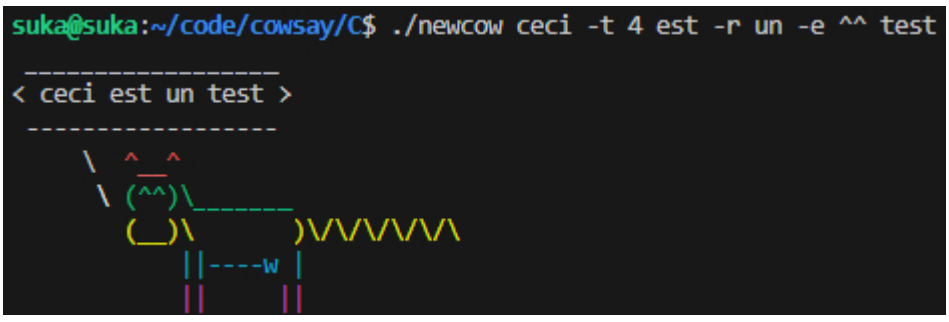
Partie C

1)

newcow.c est simplement une recréation basique de cowsay, 3 options sont disponibles :

1. --tail n (ou -t n, avec n un entier) qui rallonge la queue de la vache de n fois "/\".
2. --eyes oO (ou -e, oO avec oO deux caractères) qui va remplacer les yeux par les deux caractères donnés.
3. --rainbow (ou -r, qui s'utilise sans arguments) qui rend la vache multicolore

Tout argument donné qui n'est pas cité dans les trois points est considéré comme étant le texte à afficher par la vache. Exemple :



évidemment si l'on met toutes les options au début et le texte à la fin, ça fonctionne aussi

Pour gérer le texte à afficher et ne pas prendre les arguments, on a une variable skip qui enregistre l'index dans argv[] à éviter. C'est le cas notamment quand on fais `--tail 5` par exemple, le 5 n'est pas affiché car ce n'est pas le texte mais l'argument de l'option.

On finit ensuite par afficher la bulle du texte ainsi que la vache. Pour supporter les couleurs dans l'affichage de la vache, on initialise d'abord des chaînes de caractère contenant le code du formattage du texte avec le formattage de base, et si l'option --rainbow est donnée, alors on change ces chaînes au bon code couleur par rapport à leur nom.

2)

Notre wildcow.c est une animation basique de la vache qui court et saute au dessus d'un obstacle, notre version à l'avantage d'être un peu modulable sur les coordonnées.

Ici pour afficher la vache, on se place aux coordonnées x,y données, puis on affiche la première ligne de la vache, ensuite on se déplace aux coordonnées x,y+1, on affiche la seconde ligne, et ainsi de suite.

Dans afficher_vache, deux "type" sont disponibles, ce qui permet d'avoir deux sprites différent de la vache, pour la faire courir. On peut aussi modifier les yeux pour lui donner une expression.

```
void afficher_vache(int type, int x, int y, char* eyes){
    gotoxy(x, y);

    if (type == 1){
        printf("^__^");
        gotoxy(x, y+1);
        printf("(%c%c)\\_______", eyes[0], eyes[1]);
        gotoxy(x, y+2);
        printf("(__)\\       )\\/\\");
        gotoxy(x, y+3);
        printf("    ||----w |");
        gotoxy(x, y+4);
        printf("    ||     ||");
    } else if (type == 0){
        printf("^__^");
        gotoxy(x, y+1);
        printf("(%c%c)\\_______", eyes[0], eyes[1]);
        gotoxy(x, y+2);
        printf("(__)\\       )\\/\\");
        gotoxy(x, y+3);
        printf("    /\\----w \\");
        gotoxy(x, y+4);
        printf("   /  \\   /  \\");
    }
}
```

La fonction affiche_poteau fonction globalement de la même manière que afficher_vache et affiche un poteau aux coordonnées x,y données en paramètre

Enfin dans la fonction main, on commence par définir des coordonnées, elles sont modulables dans la mesure du possible (si on entre n'importe quoi alors on va avoir des bugs graphiques).

On entre ensuite dans une boucle qui s'arrête lorsque la vache est arrivée à gauche de l'écran. À chaque tour de cette boucle on va évidemment décrémenter la valeur x de la vache pour la faire aller de la droite vers la gauche, et on change son mode pour donner l'illusion qu'elle marche.

Les if, else if, else dans la boucle sont là pour modifier les coordonnées, yeux, mode de la vache afin de la faire sauter par dessus le poteau, tout est calculé en fonction des coordonnées du poteau, c'est de là que viens notre modulabilité.

3)

reading_cow.c est finalement assez basique, on reprend la fonction dessiner_vache et dessiner_bulle de newcow.c et ensuite on test d'ouvrir le fichier ou alors on prend stdin comme source. On initialise ensuite un tableau de caractère qui va stocker la phrase et l'index où on ajoutera les caractères.

Dans la boucle de lecture du fichier, on lit un caractère du fichier qu'on place en position i dans le tableau, on place ensuite en position i+1 un `\0` pour finir le texte, qui sera remplacé par le caractère suivant puisqu'on incrémente i à chaque tour de la boucle

```
while (!feof(f)){
    fscanf(f, "%c", &car);

    if (car == '\n')    // Gestion du cas où le caractère est un retour à
la ligne
        car=' ';        // On le remplace par un espace pour éviter les
bugs d'affichage

    update();
    dessiner_bulle(msg); dessiner_vache(car);
    sleep(1);

    msg[i]=car;
    msg[i+1]='\0';
    i++;
}
```

4)

On finit ce compte rendu avec tamagochi_cow.c, qui répond parfaitement aux attentes du sujet du projet. Ici la routine afficher_vache affiche désormais aussi une bulle de texte avec les trois états possible de la vache, les yeux de la vache changent aussi en fonction de son état et elle tire même la langue si elle est morte.

```
void dessiner_vache(int etat){
    char yeux[2]="^^";
    char langue=' ';
    if (etat == LIFESUCKS){
```

```
        strcpy(yeux, "--");
        printf(" ___________\n");
        printf("< Lifesucks >\n");
        printf(" -----------\n");
    } else if (etat == BYEBYELIFE){
        strcpy(yeux, "xx"); langue='U';
        printf(" ____________\n");
        printf("< Byebyelife >\n");
        printf(" ------------\n");
    } else if (etat == LIFEROCKS){
        printf(" ___________\n");
        printf("< Liferocks >\n");
        printf(" -----------\n");
    }
    printf("     \\  ^__^\n");
    printf("      \\ (%s)\\_______\n", yeux);
    printf("        (__)\\       )\\/\\\n");
    printf("         %c  ||----w |\n", langue);
    printf("            ||     ||\n");
}
```

stock_update et fitness_update fonctionne un peu de la même façon, on leur donne la valeur de lunchfood et vont générer une valeur de crop entre -3 et 3 avec `int crop = (rand() % 6) - 3;` et une valeur de digestion entre -3 et 0 avec `int digestion = -(rand() % 3);` ce qui fait l'aléatoire du jeu. On utilise ensuite les formules de calcul de la santé de la vache et du stock de lunchfood données dans le sujet du projet et on finit par vérifier que les valeurs restent bien dans l'intervalle [0;10]

Dans la boucle principale du jeu, on commence par afficher la vache en fonction de son état de santé actuel, puis on demande combien de lunchfood le joueur veut donner à la vache et on vérifie bien qu'il donne une valeur correcte : une valeur positive ou nulle et inférieure ou égale à la valeur du stock.

```
while (fitness != BYEBYELIFE && fitness != 10){
        update();
        if ((fitness >= 1 && fitness <= 3) || (fitness >= 7 && fitness <=
9))
            mode = LIFESUCKS;
        else if (fitness >= 4 && fitness <= 6)
            mode = LIFEROCKS;

        printf("Durée de vie : %d\n", duree_de_vie);
        dessiner_vache(mode);
        printf("stock : %d\n", stock);
        printf("Combien de lunchfood voulez vous donner à votre vache ?
\n");
        scanf("%d", &lunchfood);

        while (lunchfood > stock || lunchfood < 0){     // On gère le cas
où le joueur entre une valeur de lunchfood incorrecte.
            update();
            printf("Durée de vie : %d\n", duree_de_vie);
            dessiner_vache(mode);
```

```
            printf("stock : %d\n", stock);
            printf("Valeur de lunchfood incorrecte !!\nCombien de lunchfood
voulez vous donner à votre vache ?\n");
            scanf("%d", &lunchfood);
        }
...(suite)
```

Enfin on met à jour le stock de lunchfood et la santé de la vache et on recommence la boucle tant que la vache est encore en vie.

À la fin du jeu on affiche simplement la durée de vie de la vache (nombre de tours de la boucle) et on affiche la vache morte avecun message de game over