R1.04 - Systèmes - TP 8
Scripting Bash

[Bb]ash[1]

Le Bash

Vous connaissez **Bash** depuis plusieurs semaines (au moins) : c'est un shell, un logiciel qui permet de saisir des commandes à exécuter dans un Terminal.

Même si vous n'avez sans doute jamais eu besoin d'en lancer un à la main, un Bash se lance simplement avec la commande **bash**.

bash

bash est un langage, celui… du Bash, celui du shell.

Pourquoi le Bash a-t-il un langage ?

Taper des commandes c'est pratique (si si).

Pouvoir les enchaîner, les automatiser, c'est encore mieux !

Un shell, tel que **Bash**, est un outil d'admistrateur, ou au moins d'utilisateur averti.

Et un tel utilisateur a souvent besoin d'automatiser des tâches, des actions répétitives ! Il est pénible de devoir répéter inlassablement les mêmes séquences d'actions. On a bien d'autres choses à faire, plus intéressantes et utiles.

Ces automatismes se construisent grâce au langage **bash**, intégré au logiciel **Bash**[2].

Les programmes qu'on écrit en **bash** s'appellent des **scripts**.

Même si on a coutume de dire qu'un.e informaticien.ne peut passer une heure à mettre au point un script qui ne lui fera finalement gagner que deux minutes, c'est aussi une question de s'assurer qu'on n'oublie pas une action importante, pour la prochaine fois où

[1] Regex… encore ! :-)
[2] Il existe d'autres shells que le **bash**. Tous les shells possèdent leur langage et ces langages sont très proches les uns des autres, mais avec des nuances qui les rendent souvent incompatibles entre eux.

on aura besoin de refaire la même séquence, ça rassure. Ca permet aussi de passer la responsabilité de la tâche en question à d'autres personnes en toute tranquillité, pour vous et pour celui qui l'utilise[3], sans avoir besoin d'une tonne d'explications.

On va maintenant oublier cette subtile différence d'écriture entre **Bash** et **bash** et ne plus écrire que **bash**, pour désigner à la fois le shell et son langage 👍

Langage bash

Le **bash** est un shell créé à la fin des années 80.

Il a été conçu sur les bases d'un autre shell, le **bsh**, le Bourne shell[4], du nom de son créateur, Steve Bourne, en 1975.

On se rappellera que dans les premiers temps d'Unix, les barbus étaient assez peu enclins à rendre leurs outils *User Friendly*.

Bourne n'était pas barbu mais il avait fait barbu-2ème langue au Lycée : de fait, la syntaxe du langage **bsh** (Bourne Shell) était loin d'être simple.

Le **bash**, construit sur les bases du **bsh**, a donc hérité de cette syntaxe un peu spéciale…

Un utilisateur Linux se doit de connaître ce langage de scripting.

> Bonne nouvelle pour vous, nous allons simplement survoler le scripting bash pour étudier, dans un second temps, un autre langage de scripting plus facile à appréhender et avec lequel nous ferons des choses plus complètes et complexes.

Script vs Langage compilé

Vous connaissez déjà le langage C (et peut-être d'autres langages du même style), qu'on appelle un langage compilé.

Un **langage compilé** permet d'écrire des programmes qui doivent passer par une phase de transformation d'un code compréhensible par un être humain (le code C) en un code compréhensible par le processeur d'un ordinateur (le code machine). Cette phase de transformation est faite une fois pour toute, c'est la **compilation**.

Un **langage de script** permet d'écrire des programmes dans un code compréhensible par un être humain (le script) et qui sont convertis, en direct et à chaque exécution du script, en un code compréhensible par la machine (le code machine). C'est un

[3] Sous réserve que le travail d'automatisation ait été bien ficelé bien sûr !
[4] **bash** signifie **B**ourne **A**gain **SH**ell (Un autre shell Bourne)

interpréteur de script qui fait ce travail. Dans le cas du bash, c'est justement au shell bash que revient le rôle d'interpréteur de script.

L'avantage d'un script est qu'il est immédiatement exécutable, sans passer par une phase de compilation.

Un autre avantage est qu'il est lisible par n'importe qui, puisqu'il faut disposer du code source pour l'exécuter. Ainsi, si on dispose du script, on peut aussi l'adapter et regarder son contenu pour comprendre son fonctionnement.

Il en découle donc l'inconvénient qu'il est impossible de protéger le code d'un script, puisqu'il est clairement lisible par quiconque en dispose. Ainsi, un script contient rarement des algorithmes secrets. Pour cela on utilisera plutôt un langage compilé, tel que le C par exemple.

Comme pour la plupart des langages de programmation, bash contient aussi les éléments suivants :

- Des instructions et des fonctions
- Des structures de contrôle
- Des variables

Interpréteur bash

Le shell **bash** est un programme qui vous permet de saisir des commandes que le système va exécuter pour vous (**`ls`**, **`mkdir`**, **`cc`**, **`kill`**, etc.).

Vous vous demandez peut-être comment cela fonctionne ? La réponse est finalement simple, et vous venez de la lire un peu plus haut : le shell est un interpréteur de script, un script est un programme, et un programme est une succession d'instructions. Une commande est simplement… une de ces instructions !

Quand vous écrivez **`mkdir src/`** vous exécutez tout simplement un mini script d'une ligne, d'une instruction.

Hello World!

Vous commencez à connaître la musique : on fait ses premiers pas avec un langage en affichant un texte **`Hello World!`**, c'est la tradition, on ne va pas y déroger.

Voici comment on procède avec **bash**, tapez ceci dans votre shell :

```
echo Hello World
```

Voilà, c'est aussi simple que ça.

Variables

L'utilisation des variables est précisément le genre de choses à la syntaxe déroutante qu'on trouve dans le langage **bash** : affecter et lire une variable sont deux actions qui n'ont pas la même syntaxe !

Caractéristiques d'une variable

Une variable n'a pas de type, on y met ce qu'on veut (textes, chiffres, tableaux[5]).

Une variable porte un nom qui respecte les caractéristiques suivantes :

- Doit commencer par une lettre ou un **_** (underscore)
- Peut contenir des lettres minuscules et majuscules, mais min et MAJ ne sont pas équivalentes.
- Peut contenir des lettres chiffres
- Peut contenir des **_** (underscores)
- Ne peut contenir aucun autre symbole

Affectation

Pour affecter une variable, on utilise la syntaxe suivante :

```
nom=toto
```

Pour affecter une variable, il ne doit y avoir **AUCUN ESPACE NI AVANT NI APRES** le **=**

Lecture

Pour lire une variable, on préfixe son nom d'un **$** :

```
echo $nom
```

[5] On n'étudiera pas les tableaux.

Déclaration

Contrairement à des langages comme le C, il n'y a pas besoin d'une déclaration préalable d'une variable pour pouvoir l'utiliser (affecter ou lire).

Affecter une variable provoque sa création.

Lire une variable non affectée ne provoque pas d'erreur et c'est simplement une valeur vide (chaîne vide) qui est alors utilisée dans l'expression. Voir un exemple un peu plus loin.

Concaténations

Le contenu d'une variable peut être concaténé (à l'affichage par exemple) avec le contenu d'une autre variable ou d'un texte de la façon suivante :

```
nom=peuplu
prenom=jean
echo $prenom $nom
```

L'espace entre les deux variables est respecté à l'affichage. Mais ceci :

```
echo $prenom$nom
```

juxtapose les contenus des deux variables, sans aucun espace. Ceci fait de même :

```
echo Nom:$nom
```

La partie **`Nom:`** est un simple texte, pas une variable (il n'y a pas de **`$`** devant).

Essayez de placer un **`$`** devant (collé à) **`Nom`** :

```
echo $Nom:$nom
```

Que se passe-t-il ? Relisez les paragraphes **Déclaration** et **Caractéristiques d'une variable** pour comprendre et faites-vous expliquer par l'enseignant.e si besoin.

Tout comme on a concaténé un texte directement devant le contenu d'une variable, on peut tenter de faire de même après la variable, ainsi :

```
echo $prenomNeymar
```

Obtenez-vous le texte **JeanNeymar** ? Pourquoi ? Relisez encore les paragraphes **Déclaration** et **Caractéristiques d'une variable** pour comprendre, ou faites-vous vraiment expliquer ça par l'enseignant.e si vous n'avez toujours pas compris.

Pour aider bash à bien identifier le nom d'une variable, en cas d'ambiguïté, on peut (i.e. on doit) écrire ainsi :

```
echo ${prenom}Neymar
```

Les **{** et **}** permettent d'isoler le nom d'une variable, notamment lors de concaténation de texte à la suite du nom.

Guillemets et Apostrophes

Vous avez pu voir précédemment qu'on a pu afficher des chaînes de caractères sans les placer entre des **"** (guillemets) ou des **'** (apostrophes).

Tant qu'il n'y a pas d'ambiguïté, bash ne requiert pas d'encadrer les chaînes, comme un langage tel que le C l'obligerait.

Mais ça ne veut pas dire qu'il ne faut pas le faire car l'absence de ces **"** et **'** a des effets qu'il faut bien connaître.

Différence entre " et '

La principale différence est la façon dont les variables sont interprétées (ou pas). Testez ceci :

```
echo "$nom $prenom"
```

par rapport à ceci :

```
echo '$nom $prenom'
```

Vous comprenez la différence ? Avec les **"**, les variables à l'intérieur de la chaîne continuent à être interprétées, alors qu'avec des **'** ce n'est plus le cas, le texte est affiché en l'état.

Q.1 Affectez une variable **prix** avec la valeur **12** (ce sont des dollars) et affichez **12$US** en utilisant la variable **prix**.

Une autre différence est la façon dont les espaces sont interprétés. Sans **"** ni **'**, les espaces successifs multiples sont réduits à un seul et unique espace. Exemple :

```
echo "Nom:   $nom          $prenom"
```

par rapport à ceci :

```
echo Nom:   $nom          $prenom
```

Pour terminer, si vous avez besoin d'afficher un **"** ou un **'**, vous pouvez soit encadrer votre chaîne ainsi :

```
echo "J'adore laisser secher les sushis"
```

ou ainsi :

```
echo 'Mon nom est "Bond"... "James Bond"'
```

Ou alors préfixer le **"** ou le **'** d'un ****, ainsi :

```
echo J\'adore laisser secher les sushis
```

Fichier script

Pour le moment, vous n'avez exécuté que des commandes directement dans le shell bash. Il est souvent utile de pouvoir exécuter une série de commandes et les taper à la main, surtout quand c'est récurrent, peut s'avérer fastidieux.

C'est là qu'entrent en jeu les scripts shell, qui sont des fichiers textes permettant de stocker ces séquences pour les rejouer quand on le souhaite.

Tapez ceci dans un fichier nommé **script1** :

```
#!/bin/bash
nom=Bond
prenom=James
echo Mon nom est $nom...
echo $prenom $nom
```

Pour pouvoir exécuter ce script, il doit être… exécutable ! Bravo, et ça se fait ainsi (une seule fois, juste quand il est créé) :

```
chmod +x script1
```

Il ne reste plus qu'à l'exécuter ainsi :

```
./script1
```

Notez la présence de la 1ère ligne du script. C'est ce qu'on appelle le Shebang[6]. Cette ligne indique simplement le nom et le chemin menant à l'interpréteur de script, ici le bash.

Placez toujours cette ligne, sans espace nulle part, en 1ère ligne de vos scripts bash.

Exécution de commandes

Vous pouvez évidemment exécuter aussi n'importe quelle commande dans un script. Exemple à saisir dans un script nommé **script2** :

```
#!/bin/bash
dossier=un_dossier_de_test
echo "Creation d'un dossier \"$dossier\""
mkdir $dossier
ls -l
```

[6] Prononcez "chibang". She = Sharp (dièse en anglais, le **#**), Bang = **!** (comme une exclamation)

Exécutez votre **script2** (attention, n'oubliez pas de le rendre exécutable avant tout).

Sorties

STDOUT

On l'a déjà vu, pour écrire sur la sortie standard, comme le **printf()** en C, il suffit de faire un :

```
echo "C'est fin, ca se mange sans fin"
```

STDERR

Pour écrire sur la sortie des erreurs, comme le **fprintf(stderr, ...)** en C, il suffit de faire un :

```
>&2 echo "Erreur"
```

C'est une forme d'écriture qui utilise les redirections vues dans un précédent TP. Retenez surtout la syntaxe, sans chercher à en comprendre toute la finesse ! 😳

Entrée - STDIN

Pour lire sur **STDIN** (par défaut il s'agit du clavier), on peut utiliser la commande interne du bash :

```
read nom_variable
```

Exemple de **script3** à saisir et tester :

```
#!/bin/bash
echo -n "Quel est votre prenom ? "
```

```
read prenom
echo Bonjour $prenom, je suis enchante de vous rencontrer.
```

Vous noterez le **-n** ajouté au **echo**. Il permet d'afficher quelque chose en conservant le curseur à la fin du texte au lieu de le ramener sur le début de la ligne suivante. C'est comme si on avait omis un **\n** dans un **printf()** du C.

Notez aussi l'absence de **$** devant le nom de la variable lors d'un **read**. C'est ainsi car c'est une sorte d'affectation de la variable. Encore une bizarrerie du bash.

Paramètres

Comme avec les commandes, passer des paramètres aux scripts peut s'avérer très utile.

On peut accéder aux paramètres à l'aide de variables spéciales, dont le nom est **$<numéro_du_paramètre_sur_la_ligne>**.

Exemple de **script4** à saisir :

```
#!/bin/bash
echo Bonjour $1 $2, vous etes l\'agent $3
```

et à tester ainsi :

```
./script4 James Bond 007
```

Paramètre 0

Q.2 La variable **$0** existe aussi. Que contient-elle ? Faites un essai pour le découvrir.

Structures de contrôle

Tests

La syntaxe des tests est une autre bizarrerie du bash. Retenez la syntaxe sans chercher à y trouver une quelconque logique.

Pour effectuer un test, il y a deux syntaxes possibles.

Saisissez et testez les deux syntaxes suivantes, dans **script5** et **script6** :

Syntaxe 1 - commande **test**

```
#!/bin/bash
read -p "Quel est votre nom ? " nom
if test "$nom" = ""
then
   echo Bonjour inconnu.e
else
   echo Bonjour $nom
fi
```

Notez au passage cette nouvelle syntaxe du **read** qui permet, avec un **-p**[7] et une chaîne de caractères, d'effectuer un **echo** de cette chaîne avant de lire la réponse au clavier.

Syntaxe 2 - avec **[ ]**

```
#!/bin/bash
read -p "Quel est votre nom ? " nom
if [ "$nom" = "" ]
then
   echo Bonjour inconnu.e
else
   echo Bonjour $nom
fi
```

Que faut-il retenir ?

La syntaxe avec **test** ou avec **[ ]** est toujours :

```
if ...
then
   ...
else
   ...
fi
```

[7] **p** comme **prompt** (inviter en anglais)

- Le **then** est obligatoire, c'est une sorte de **{** du C.
- Le **fi** termine le **if** (c'est **if** à l'envers[8]), c'est une sorte de **}** du C.
- Le **else** est facultatif.
- Il y a toujours un espace devant et derrière l'opérateur.
- Il y a toujours un espace après **[** et avant **]** dans la syntaxe utilisant **[ ]**

Les opérateurs

C'est là qu'on rigole...

- **=** : pour l'égalité de chaînes de caractères (et non pas **==** !!!)
- **!=** : pour la différence de chaînes de caractères
- **-eq**[9] : pour l'égalité de valeurs numériques 😕
- **-ne**[10] : pour la différence de valeurs numériques
- **-gt**, **-ge**, **-lt** et **-le**[11] : pour respectivement : **>**, **≥**, **<** et **≤** de valeurs numériques

Ne confondez pas les opérateurs, un **-gt** sur une variable non numérique par exemple !

Consultez le manuel de la commande **test** pour une liste exhaustive des opérateurs.

Test vs affectation avec =

C'est ici qu'il est important de bien se rappeler des syntaxes :

- Affectation de variable : **pas d'espace** devant ni derrière le **=**
- Test de valeur : **un espace devant et un espace derrière** le **=**

Ceci fait **TOUTE LA DIFFÉRENCE** !

De l'importance des guillemets

Dans les deux exemples précédents, vous aurez noté que la variable **$nom** était encadrée de **"** (guillemets). On pourrait penser que ce n'est pas vraiment nécessaire, mais c'est **ABSOLUMENT ESSENTIEL** de les utiliser.

Relancez le **script5**, et cette fois-ci appuyez simplement sur la touche **ENTREE** sans saisir de nom. Le script fonctionne correctement et détecte l'absence de saisie du nom.

Maintenant, modifiez votre **script5** en supprimant les **"** de la façon suivante :

[8] Humour de barbu...
[9] **eq** signifie **eq**ual
[10] **ne** signifie **n**ot **e**qual
[11] **g**reater **t**han, **g**reater or **e**qual, **l**ess **t**han, **l**ess or **e**qual

```
...
if test $nom = ""
...
```

et exécutez-le en ne saisissant pas de **nom** encore une fois. Vous obtenez certainement une erreur bizarre.

Q.3 Que se passe-t-il ? Essayez de remplacer, sur le papier, le contenu de la variable **$nom** par ce qui a été saisi au clavier, c'est à dire rien (une chaîne vide), que devient cette ligne de code ? Si vous ne comprenez pas, demandez à votre enseignant.e.

Boucles

Il existe plusieurs types de boucles mais nous allons nous limiter ici à celle qui est la plus commune et sans doute la plus utile : la boucle **for ... in** qui permet de boucler sur une série de valeurs explicitées ou une liste d'objets de l'arborescence.

Saisissez ceci dans un **script7** :

```
#!/bin/bash
for fic in mickey donald dingo
do
   touch $fic
done
ls -l
```

Exécutez-le et observez.

Que faut-il retenir ?

La syntaxe de **for ... in** est toujours :

```
for variable in liste_de_valeurs ...
do
   ...
done
```

- Le **do** est obligatoire, c'est une sorte de **{** du C.
- Le **done** termine le **do**, c'est une sorte de **}** du C.

- A chaque tour de boucle, la **variable** prend une des valeurs de la **liste_de_valeurs**

La liste_de_valeurs peut aussi correspondre à un développement de jokers du shell.

Exemple, saisissez ceci dans un **script8** :

```
#!/bin/bash
for fic in *
do
   echo "J'ai trouve un fichier $fic"
done
```

Exécutez-le et observez.

Le joker ***** qui signifie *n'importe quel fichier* est développé par le shell quand le script exécute cette ligne et est remplacé par la liste exhaustive des fichiers présents dans le répertoire courant. On pourrait tout aussi bien écrire ceci :

```
for fic in ../*
```

ou encore :

```
for fic in ~/Documents/algo/TP3/*.c
```

Redirections

On connaît déjà les redirections des commandes, utilisées notamment avec les filtres.

Résultat de commande

Le bash est capable d'une autre redirection intéressante : rediriger le résultat d'une commande vers une variable.

Saisissez ceci dans un **script9** :

```
#!/bin/bash
nb_lignes=$(wc -l < $0)
echo "Ce script s'appelle $0 et contient $nb_lignes lignes"
```

Q.4 Exécutez-le et observez.

Essayez de le comprendre. Si vous n'y parvenez pas, demandez de l'aide à votre enseignant.e.

Ce qu'on ne verra pas...

...mais qu'il faut savoir que ça existe[12] !

Le langage bash ayant parfois une syntaxe peu conventionnelle et compliquée à mémoriser/utiliser, nous avons fait le choix de ne vous présenter que les bases de bash qui vous permettront déjà de réaliser des scripts simples et utiles, des choses du quotidien.

Pour la création de scripts plus complexes, nous avons choisi de nous attarder sur un autre langage de scripting que l'on va étudier très prochainement.

Voici donc quelques éléments du langage bash qui existent et qu'on ne va pas aborder :

- Les boucles traditionnelles, comme en C : **`for (...)`**, **`while`**, **`until`**
- Les structures conditionnelles : **`switch... case`**
- Les fonctions
- Les tableaux/listes
- Les calculs arithmétiques
- Les manipulations/extractions de sous-chaînes depuis des variables
- La gestion de processus
- Les Regex dans les tests

Mise en pratique

Ecrivez et testez des scripts réalisant les tâches suivantes.

Chaque script doit être écrit dans un fichier séparé nommé du nom de la question, exemple **`q5`** puis **`q6`** etc.

Q.5 Demandez la saisie au clavier d'un nom de fichier (choisissez un fichier inexistant pour vos tests !), et créez un fichier vide portant ce nom.

Q.6 Sur la base de la Q.5, le fichier créé portera le nom saisi par l'utilisateur, auquel vous coderez l'ajout d'un **`.txt`**

[12] Et donc chercher dans la doc si besoin.

Q.7 Sur la base de la Q.5, le fichier créé portera le nom saisi par l'utilisateur, auquel vous coderez l'ajout d'un **`_old`** à son nom (attention, piège)

Q.8 A l'aide d'une boucle, créez les fichiers suivants :

```
test123
abc
_def
toto.old
```

Q.9 Sur la base de la Q.8, créez les dossiers suivants :

```
dir1
doss.old
docs
docs/backup
```

Q.10 Sur la base de la Q.9, créez les dossiers suivants :

```
folderA
folderB
folderC
```

vous devez optimiser votre boucle afin de ne boucler que sur la partie variable (**`A`**, **`B`** et **`C`**), la partie **`folder`** étant la même pour tous les noms.

Ensuite, chaque dossier créé devra contenir les fichiers vides suivants :

```
TODO.txt
README.md
main.c
```

Q.11 Créez un fichier **`prog.c`** avec le code C suivant :

```
#include <stdio.h>
int main() {
```

```
    return 123;
}
```

Compilez-le et faites ceci dans un script :

- Appelez l'exécutable binaire compilé (donc **prog** en principe)
- Juste après, affichez la valeur de la variable **$?**

Qu'obtenez-vous ? Comprenez-vous d'où vient cette valeur ? Modifiez le code source C pour changer la valeur retournée.

$? est donc une variable spéciale qui contient la valeur de retour de la dernière commande exécutée. Cette valeur est toujours positive ou nulle.

C'est pour cette raison qu'on vous a appris à toujours retourner **EXIT_SUCCESS** (**0**) ou **EXIT_FAILURE** (**1**) dans vos programmes en C. Mais vous pouvez retourner la valeur que vous voulez. Il est conventionnel de retourner **0** quand l'exécution est OK et autre chose sinon. Documentez toujours ce que votre programme retourne pour que l'utilisateur sache interpréter la valeur de retour.

NB : la valeur de retour (**exit**) n'a rien à voir avec l'affichage qu'une commande fait.

Q.12 Récupérez le fichier **depts** du **TP 5** et placez-le dans votre dossier de travail.

Écrivez un script qui demande au clavier un numéro de département et qui affiche à l'écran la ligne de **depts** correspondant à ce numéro de département.

Astuce : pensez à la façon dont vous feriez cela manuellement et directement dans le Terminal. Un script est capable de faire la même chose, sauf qu'on souhaite le faire avec la valeur contenue dans une variable.

Q.13 Récupérez le fichier **prod** du **TP 5** et placez-le dans votre dossier de travail.

Écrivez un script qui demande au clavier un **texte** et un n° de champ **num**, et qui affiche uniquement le champ n° **num** des lignes de **prod** qui contiennent le **texte** quelque part sur la ligne. On suppose que l'utilisateur saisit des informations correctes (texte sans espace et une valeur numérique valide)

Q.14 Écrivez un script qui demande au clavier un texte à rechercher dans **prod** et affiche le nombre de lignes dans lesquelles figure ce texte.

Q.15 Sur la base de la Q.14, ajoutez l'affichage spécial du texte : **Aucun** lorsqu'aucune ligne n'est trouvée (comptage = 0).

Q.16 Sur la base de la Q.15, ajoutez le test que la valeur saisie n'est pas vide. Pour vous aider, voici un extrait de code qui permet d'afficher un message d'erreur et de sortir immédiatement du script, en interrompant son exécution (bien sûr il vous reste le reste à écrire, notamment le test) :

```
echo Erreur de saisie
exit 1
```

Q.17 Reprenez la Q.13 en utilisant deux paramètres à la commande au lieu de les lire au clavier. Vous devrez vous assurer que l'utilisateur a bien passé 2 paramètres.

Astuce : il existe une variable spéciale qui contient le nombre de paramètres passés au script, il s'agit de **$#**

Q.98 Reprenez la base de la Q.5, le fichier créé portera le nom saisi par l'utilisateur, auquel vous coderez l'ajout d'un **.c**.

Vous placerez, par code dans votre script, dans le fichier créé, les lignes suivantes :

```
#include <stdio.h>
int main() {
   printf("Hello World!\n");
}
```

Conseil : placez ces lignes (par code), une par une.

Attention, il y a quelques pièges.

Q.99 Sur la base de la question précédente, ajoutez ensuite la compilation du fichier **.c** créé, puis l'exécution du binaire compilé, tout ceci dans le script.

Bonus :

Puis, testez dans votre script que la compilation s'est bien déroulée (introduisez une erreur de syntaxe dans le code C pour tester, par exemple remplacez le **printf** par un **print**).