Identification /5

Prénom: .....

Nom: .....

Code permanent: .....

Veuillez choisir **une seule réponse** par question, avec une croix (×) ou un crochet (✓).

Q1. Considérez une machine fonctionnant dans un environnement Linux, dont le répertoire courant contient le fichier `q1.c` suivant:

```
#include <stdio.h>

int main(void) {
  putchar(getchar());
  return 0;
}
```

Qu'est-ce qui se produira si on entre la commande suivante?

```
$ gcc -c q1.c
```

- ☐ Le programme `q1` s'exécutera anormalement
- ☐ Le programme `q1` s'exécutera normalement
- ■ Un fichier `q1.o` sera produit
- ☐ Un fichier `q1` sera produit

Q2. Considérez le fichier `q2.c` suivant:

```
#include <stdio.h>

int main(void) {
  int c = 'A';
  return c == 'B' ? 0 : 1;
}
```

Supposons qu'on entre les commandes suivantes:

```
$ gcc q2.c -o q2
$ ./q2
$ echo $?
```

Qu'est-ce qui sera affiché sur la sortie standard par la dernière commande?

- ☐ `$?`
- ■ `1`
- ☐ `0`
- ☐ Rien

Q3. Qu'est-ce qui sera affiché sur la sortie standard par le programme suivant?

```
#include <stdio.h>

int main(void) {
  unsigned char c = -2;
  printf("%d\n", (char)c);
  return 0;
}
```

- ☐ `126`
- ■ `-2`
- ☐ `254`
- ☐ `%d`

Q4. Parmi les commandes Git suivantes, laquelle peut entraîner un conflit?

- ☐ `git clone`
- ☐ `git push`
- ☐ `git fetch`
- ■ `git rebase`

Q5. Supposez que le répertoire courant contienne deux fichiers: le premier nommé `b` et le second nommé `Makefile`, dont le contenu est comme suit:

```
a: b
  cp b a

c: b
  mv b d

d: a
  rm b
```

Que se passera-t-il si on entre la commande suivante (indiquez l'énoncé qui est vrai)?

```
$ make d
```

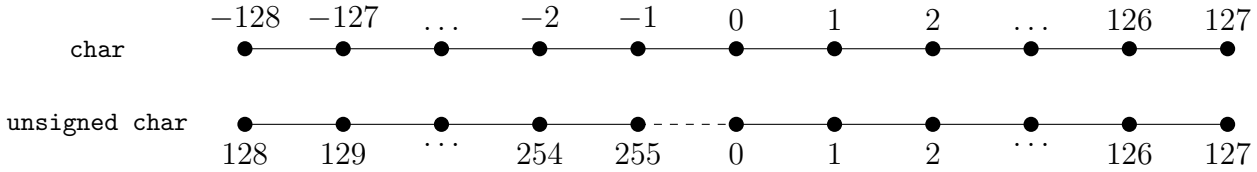
- ■ Le répertoire courant contiendra le fichier `a`
- ☐ Le répertoire courant contiendra le fichier `b`
- ☐ Le répertoire courant contiendra le fichier `c`
- ☐ Le répertoire courant contiendra le fichier `d`

Voir explications page suivante

Q1. La commande `gcc -c q1.c` sert à compiler le fichier source `q1.c` en un fichier objet. Par défaut, le nom du fichier objet produit est `q1.o`. En particulier, l'option `-c` permet de compiler un programme, mais ne fait pas l'édition des liens. Il est important de ne pas confondre une commande qui compile un programme avec celle qui exécute un programme compilé.

Q2. La commande `gcc q2.c -o q2` permet de compiler le programme en un exécutable nommé `q2`. La commande `./q2` lance l'exécutable `q2`. À la fin de l'exécution, un code de retour est retourné, qui est égal à la valeur retournée par la fonction `main`. Dans le cas présent, cette valeur est `1`, puisque la condition `c == 'B'` est fausse. Finalement, la commande `echo $?` permet d'afficher sur la sortie standard le code retourné par la dernière commande, de sorte que c'est la valeur `1` qui sera affichée.

Q3. Le type `char` permet de représenter une valeur entière signée entre -128 et 127, alors que le type `unsigned char` permet de représenter une valeur entière entre 0 et 255. Ainsi, lorsqu'on stocke la valeur `-2` dans la variable `c`, celle-ci est convertie en valeur `254`, puisque le type de `c` est `unsigned int`. Lors de l'appel de `printf`, on convertit la variable `c` en caractère signé avec l'expression `(char)c`, qui nous ramène à `-2`. L'image ci-bas illustre visuellement la correspondance lors de conversions entre les types `char` et `unsigned char`.



Q4. La notion de *conflit* dans Git est très spécifique: il y a un conflit de versions lorsqu'on tente de fusionner deux branches (ou deux versions) d'un historique et qu'il y a des zones de texte qui ont été modifiées de façon commune aux deux branches. Un tel conflit peut survenir lorsqu'on effectue un rebasement, avec la commande `git rebase`, qui réapplique une série de *commits* successivement en vérifiant, à chaque fois, qu'il n'y a pas de conflit. La sous-commande `git clone` n'entraîne jamais de conflit, car elle sert à dupliquer un dépôt Git déjà existant. Quant à la commande `git fetch`, elle télécharge les branches d'un dépôt distant vers un dépôt local. Elle ne peut pas entraîner de conflit, car elle ne fusionne pas les branches distantes et locales entre elles: elle ne fait que les télécharger. Finalement, la commande `git push` peut échouer si on tente de partager sur un dépôt distant des branches locales qui ont un historique incompatible, mais il ne s'agit pas d'un conflit.

Q5. La commande `make d` demande à l'outil Make de produire la cible `d` du fichier `Makefile`, c'est-à-dire la troisième cible. Pour la produire, il faut d'abord produire la cible `a`, qui est une dépendance de cette cible. La cible `a` dépend de la cible `b`, qui est un fichier qui existe dans le répertoire courant. La recette `cp b a` est donc appliquée, ce qui permet de produire un nouveau fichier `a` en copiant le fichier `b`. Lorsque la cible `a` est réalisée, on peut produire la cible `d` en exécutant la recette `rm b`, qui supprime le fichier `b`. On obtient donc, à la toute fin, un répertoire ne contenant que le fichier `a`, obtenu en copiant `b`. À noter que la cible `c` n'est jamais appelée, car `c` n'est pas une dépendance de `a` ou de `d`. C'est donc le premier énoncé qui est vrai.