

Identification

/5

Prénom:

Nom:

Code permanent:

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

Q1. Quel projet, annoncé par Richard Stallman en 1984, a notamment mené au développement du compilateur `gcc`?

- ☐ Unix
 ☒ GNU
 ☐ Linux
 ☐ Git

Q2. Considérez un répertoire courant contenant uniquement le fichier `calc.c` et le `Makefile` suivant:

```
calc: calc.o
gcc -o calc calc.o
```

```
calc.o: calc.c
gcc -c calc.c
```

Quelles commandes seront affichées, dans l'ordre, lorsqu'on entre `make`?

- ☐ `gcc -o calc calc.o`, puis `gcc -c calc.c`
☒ `gcc -c calc.c`, puis `gcc -o calc calc.o`
☐ `gcc -c calc.c` seulement
☐ `gcc -o calc calc.c` seulement

Q3. Vous avez cloné un dépôt Git, puis modifié le fichier `README.md`. Quelle séquence de commandes permet de rendre cette modification disponible sur le dépôt distant?

- ☐ `git commit`, puis `git add README.md`, puis `git push`
☐ `git add README.md`, puis `git push`
☒ `git add README.md`, puis `git commit`, puis `git push`
☐ `git add README.md`, puis `git commit`, puis `git pull`

Q4. Qu'est-ce qui sera affiché sur la sortie standard par le programme suivant?

```
#include <stdio.h>

enum Taille { PETIT, MOYEN, GRAND };

int main(void) {
    int x = 0x0A;
    enum Taille t = GRAND;
    switch (t) {
        case PETIT: printf("p ");
        case MOYEN: printf("m ");
        case GRAND: printf("g ");
        default: printf("d ");
    }
    printf("%d\n", x + t);
    return 0;
}
```

- ☐ g 12
 ☒ g d 12
 ☐ g d 13
 ☐ p m g d 12

Q5. Écrivez un programme complet en C qui lit des caractères sur l'entrée standard, un à un, jusqu'à la fin du fichier (EOF), et qui les réécrit sur la sortie standard, en remplaçant toutefois chaque chiffre (0 à 9) par le caractère #. Par exemple, le texte `Local PK-1234` devient `Local PK-####`. *Rappel:* la fonction `int getchar(void)` lit un caractère sur l'entrée standard et retourne sa valeur, convertie en `int`, ou la constante `EOF` lorsque la fin du fichier est atteinte; la fonction `int putchar(int c)` écrit le caractère `c` sur la sortie standard. *Note:* utilisez le verso au besoin.

[Voir explications page suivante](#)

Q1. Richard Stallman a annoncé le projet GNU (*GNU is not Unix*) en 1984, dans le but de développer des logiciels Unix libres, dont le compilateur C `gcc`. Unix a plutôt été développé à partir de 1969 par Ken Thompson et Dennis Ritchie, chez Bell. Linux est un noyau libre annoncé par Linus Torvalds en 1991, alors que Git est un logiciel de contrôle de versions créé par ce même Linus Torvalds en 2005.

Q2. Lorsqu'on invoque `make` sans argument, c'est la première cible du `Makefile` qui est construite, soit `calc`. Or, celle-ci dépend du fichier `calc.o`, qui n'existe pas encore: `make` doit donc d'abord construire cette dépendance, à l'aide de la règle `calc.o`, qui exécute `gcc -c calc.c`. Une fois le fichier objet produit, la commande de la cible `calc`, soit `gcc -o calc calc.o`, peut être exécutée pour l'édition des liens. Par défaut, `make` affiche chacune des commandes exécutées, dans l'ordre. L'ordre inverse est impossible, puisque l'édition des liens nécessite que `calc.o` existe déjà. La commande `gcc -o calc calc.c`, qui combine compilation et édition des liens, n'apparaît nulle part dans le `Makefile`.

Q3. Il faut d'abord ajouter la modification à l'aire de *staging* avec `git add`, puis la sauvegarder dans l'historique *local* du dépôt avec `git commit`, et finalement la téléverser vers le dépôt distant avec `git push`. Si on inverse `git commit` et `git add`, le *commit* est effectué alors que l'aire de *staging* est vide: aucun *commit* n'est créé, et la modification ajoutée ensuite n'est jamais téléversée. Si on omet `git commit`, la modification reste dans l'aire de *staging*: `git push` ne téléverse que des *commits*, donc rien de nouveau n'est envoyé. Finalement, `git pull` sert à *récupérer* les changements du dépôt distant, et non à y téléverser les siens.

Q4. Dans une énumération, la première constante vaut 0 (et non 1), la seconde vaut 1, etc. Ainsi, `PETIT` vaut 0, `MOYEN` vaut 1 et `GRAND` vaut 2. Dans le `switch`, les `case` sont parcourus séquentiellement jusqu'à ce qu'il y ait une correspondance, soit `case GRAND`: les `case` précédents sont ignorés, de sorte que ni `p` ni `m` ne sont affichés. Puisqu'il n'y a aucun `break`, l'exécution se poursuit ensuite avec toutes les instructions suivantes, y compris celle du `default`: on affiche donc `g` puis `d`. Finalement, le littéral hexadécimal `0x0A` vaut 10 en décimal et, puisqu'une variable de type `enum` est traitée comme un `int`, l'expression `x + t` vaut `10 + 2 = 12`. On obtient donc `g d 12`.

Q5. La variable qui reçoit la valeur retournée par `getchar` doit être de type `int` (et non `char`), car `EOF` est une valeur entière négative qui ne peut pas nécessairement être représentée dans un `char`. On relit un caractère à la fois à l'aide de `getchar` jusqu'à ce que la valeur `EOF` soit obtenue, tout comme dans l'exemple `maj.c` vu en classe. Pour chaque caractère lu, on vérifie s'il s'agit d'un chiffre: si c'est le cas, on écrit `#` à l'aide de `putchar`; sinon, on réécrit le caractère tel quel. Notez qu'il ne faut *pas* oublier la branche `else`: tous les caractères qui ne sont pas des chiffres (y compris les espaces et les sauts de ligne) doivent aussi être réécrits. Deux solutions équivalentes sont présentées ci-dessous: la première compare le caractère aux bornes de l'intervalle `'0'..'9'`, alors que la seconde utilise la fonction `isdigit` de la bibliothèque `ctype.h`. Une variante plus compacte, utilisant l'opérateur ternaire, ainsi qu'une quatrième solution, moins bonne, utilisant une instruction `switch`, sont présentées ensuite.

Version avec intervalle

```
#include <stdio.h>

int main(void) {
    int c;
    while ((c = getchar()) != EOF) {
        if (c >= '0' && c <= '9') {
            putchar('#');
        } else {
            putchar(c);
        }
    }
    return 0;
}
```

Version avec isdigit

```
#include <stdio.h>
#include <ctype.h>

int main(void) {
    int c;
    while ((c = getchar()) != EOF) {
        if (isdigit(c)) {
            putchar('#');
        } else {
            putchar(c);
        }
    }
    return 0;
}
```

Version avec intervalle. Cette version exploite le fait qu'un caractère est un nombre (son code ASCII) et que les codes des chiffres '0' à '9' sont consécutifs (de 48 à 57). Il faut utiliser l'opérateur `&&` (et non `||`): la condition `c >= '0' || c <= '9'` serait *toujours* vraie, puisque tout caractère est soit supérieur ou égal à '0', soit inférieur ou égal à '9'. Il faut aussi prendre garde à comparer `c` aux *caractères* '0' et '9', et non aux *entiers* 0 et 9, qui correspondent à des caractères de contrôle non imprimables.

Version avec isdigit. La fonction `isdigit`, déclarée dans `ctype.h` (la même bibliothèque que `toupper`, utilisée dans `maj.c`), retourne une valeur non nulle (vraie) si son argument est un chiffre décimal, et 0 (faux) sinon. Le code est ainsi plus lisible, puisque l'intention est exprimée directement par le nom de la fonction. Il ne faut pas oublier d'inclure `ctype.h`.

Version avec l'opérateur ternaire. Puisque les deux branches du `if` appellent `putchar` et ne diffèrent que par le caractère écrit, on peut les fusionner à l'aide de l'opérateur ternaire `? : .` L'expression `condition ? a : b` vaut `a` si la condition est vraie, et `b` sinon: on écrit donc `'#'` si `c` est un chiffre, et `c` lui-même autrement. Cette version est équivalente aux précédentes et tout aussi correcte. Comme la boucle ne contient qu'une seule instruction, on peut même omettre les accolades. On obtient ainsi une version à la fois compacte et lisible, qui est sans doute la meilleure des solutions présentées. La condition `isdigit(c)` pourrait d'ailleurs être remplacée par `c >= '0' && c <= '9'`.

```
#include <stdio.h>
#include <ctype.h>

int main(void) {
    int c;
    while ((c = getchar()) != EOF)
        putchar(isdigit(c) ? '#' : c);
    return 0;
}
```

Version avec switch. On pourrait aussi utiliser une instruction `switch`, avec un `case` par chiffre, comme ci-dessous. Cette solution est correcte, mais elle est moins bonne que les deux précédentes, car elle est beaucoup moins concise: il faut énumérer les dix chiffres un à un, alors qu'un intervalle de valeurs consécutives s'exprime naturellement avec une seule condition. Remarquez aussi le `break` après `putchar('#')`: s'il était omis, l'exécution se poursuivrait dans le `default`, et chaque chiffre serait remplacé par `#` *suivi* du chiffre lui-même (voir la question 4).

```
#include <stdio.h>
```

```
int main(void) {
    int c;
    while ((c = getchar()) != EOF) {
        switch (c) {
            case '0': case '1': case '2': case '3': case '4':
            case '5': case '6': case '7': case '8': case '9':
                putchar('#');
                break;
            default:
                putchar(c);
        }
    }
    return 0;
}
```

Comparaison. Les quatre versions sont correctes et produisent exactement le même résultat, mais la version avec `switch` est à éviter, pour les raisons mentionnées plus haut. La version avec `isdigit` est généralement la plus recommandée: elle est concise, lisible, et ne dépend pas de la façon dont les caractères sont encodés.