Révision: chapitres 1 à 4

Corrigé

Question 1

Le répertoire courant contient uniquement le fichier `calc.c`. On entre les commandes suivantes:

```
$ gcc -c calc.c
$ gcc calc.o
```

Quels fichiers contiendra alors le répertoire?

`calc.c` et `a.out`

`calc.c`, `calc.o` et `calc`

`calc.c`, `calc.o` et `a.out`

`calc.c` et `calc.o`

EXPLICATION

- `gcc -c calc.c` **compile** seulement: elle produit le fichier objet `calc.o`;
- `gcc calc.o` fait l'**édition des liens**. Comme l'option `-o` n'est pas utilisée, l'exécutable porte le nom par défaut, `a.out`.

Le fichier `calc.o` n'est pas supprimé par la deuxième commande.

Question 2

Un répertoire contient le fichier `prog.c` et le `Makefile` suivant:

```
prog: prog.o
	gcc -o prog prog.o

prog.o: prog.c
	gcc -c prog.c
```

On entre `make`, puis on supprime le fichier `prog` avec la commande `rm prog`. Quelles commandes seront exécutées si on relance `make`?

Aucune, puisque `prog.c` n'a pas été modifié

Seulement `gcc -c prog.c`

Seulement `gcc -o prog prog.o`

`gcc -c prog.c`, puis `gcc -o prog prog.o`

EXPLICATION

`make` reconstruit une cible lorsqu'elle **n'existe pas** ou lorsqu'une de ses **dépendances** est plus récente qu'elle:

- `prog.o` existe toujours et est plus récent que `prog.c`: il n'est pas recompilé;
- `prog` n'existe plus: il est reconstruit à partir de `prog.o`.

Question 3

Dans votre copie locale d'un dépôt Git, vous modifiez le fichier `main.c` et vous créez un nouveau fichier `notes.txt`. Vous entrez ensuite:

```
$ git add main.c
$ git commit -m "Corrige main"
$ git push
```

Que récupérera votre coéquipier s'il entre `git pull`?

- Les modifications de `main.c` et le fichier `notes.txt`
- **Seulement les modifications de `main.c`**
- Seulement le fichier `notes.txt`
- Rien, car `notes.txt` n'a pas été ajouté

EXPLICATION

Seuls les changements ajoutés avec `git add` font partie du *commit*. Le fichier `notes.txt` n'est pas versionné: il reste dans votre copie locale, sans empêcher le *commit* ni le `git push` des modifications de `main.c`.

Question 4

Le programme `analyse.c` suivant distingue les options (arguments qui commencent par `-`) des noms de fichiers reçus sur la ligne de commande:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
  int nb_options = 0;
  for (int i = 1; i < argc; ++i)
    if (argv[i][0] == '-')
      ++nb_options;
  printf("%d option(s), %d fichier(s)\n",
         nb_options, argc - 1 - nb_options);
  return 0;
}
```

Une fois le programme compilé, qu'affichera la commande suivante?

```
$ ./analyse -v notes.txt -n "mon rapport.txt"
```

- `2 option(s), 2 fichier(s)`
- `2 option(s), 3 fichier(s)`
- `2 option(s), 4 fichier(s)`
- `0 option(s), 4 fichier(s)`

EXPLICATION

- `argc` vaut 5: `argv[0]` est le nom du programme (`./analyse`), suivi des 4 arguments `-v`, `notes.txt`, `-n` et `mon rapport.txt`. Les guillemets regroupent `mon rapport.txt` en **un seul** argument, malgré l'espace;
- La boucle commence à `i = 1` pour ignorer le nom du programme;
- `argv[i][0]` est le premier caractère du `i`-ème argument: il vaut `'-'` pour `-v` et `-n`, d'où 2 options;
- Il reste `argc - 1 - nb_options = 5 - 1 - 2 = 2` fichiers.

Question 5

Qu'affiche le fragment suivant?

```c
int n = 1234, s = 0;
while (n > 0) {
  s += n % 10;
  n /= 10;
}
printf("%d\n", s);
```

10

4

1234

4321

EXPLICATION

À chaque tour, `n % 10` donne le **dernier chiffre** de `n`, et `n /= 10` le retire (division **entière**). On additionne donc les chiffres: `4 + 3 + 2 + 1 = 10`. La boucle s'arrête quand `n` vaut 0.

Question 6

Qu'affiche le fragment suivant?

```c
for (int i = 0; i < 10; ++i) {
  if (i % 2 == 0)
    continue;
  if (i > 6)
    break;
  printf("%d ", i);
}
```

```
1 3 5 7 9
```

```
0 2 4 6
```

```
1 3 5
```

```
1 3 5 7
```

EXPLICATION

- `continue` passe **directement au tour suivant**: les valeurs paires ne sont jamais affichées;
- `break` **termine** la boucle: dès que `i` vaut 7, on sort de la boucle sans afficher 7.

Question 7

Qu'affiche le fragment suivant?

```c
int i = 0, j = 0;
if (i != 0 && ++j > 0)
  printf("A ");
if (i == 0 || ++j > 0)
  printf("B ");
printf("%d\n", j);
```

```
A B 2
```

```
B 1
```

```
B 0
```

```
B 2
```

EXPLICATION

Les opérateurs `&&` et `||` sont évalués de gauche à droite et s'arrêtent dès que le résultat est connu (évaluation **paresseuse**):

- `i != 0` est faux, donc `i != 0 && ...` est faux: `++j` n'est **pas** évalué;
- `i == 0` est vrai, donc `i == 0 || ...` est vrai: `++j` n'est **pas** évalué non plus.

`j` vaut donc toujours 0.

Question 8

Qu'affiche le fragment suivant?

```c
char s[] = "Bonjour";
for (int i = 0; s[i] != '\0'; i += 2)
  s[i] = '*';
printf("%s\n", s);
```

*o*j*u*

B*n*o*r

*o*j*u

****

EXPLICATION

On remplace les caractères aux indices **pairs** (0, 2, 4 et 6), en commençant par le `B`. La chaîne `"Bonjour"` contient 7 caractères: le dernier, `r` (indice 6), est donc aussi remplacé.

Question 9

Qu'affiche le fragment suivant?

```c
int j = 0, m = 0, a = 0;
int n = sscanf("2026-10-19", "%d/%d/%d", &a, &m, &j);
printf("%d %d %d %d\n", n, a, m, j);
```

```
3 2026 10 19
```

```
0 0 0 0
```

```
1 2026 0 0
```

```
3 2026 0 0
```

EXPLICATION

`sscanf` lit selon le format **tant que possible** et retourne le **nombre de variables affectées**: `2026` est lu dans `a`, mais le caractère `-` ne correspond pas au `/` attendu. La lecture s'arrête là: `n` vaut 1, et `m` et `j` conservent leur valeur initiale.

D'où l'importance de **vérifier** la valeur de retour de `sscanf`.

Question 10

Considérez la fonction suivante:

```c
void f(int t[], int n) {
  t[0] = n;
  n = 0;
}
```

Qu'affiche le fragment suivant?

```c
int t[] = {1, 2, 3}, n = 5;
f(t, n);
printf("%d %d\n", t[0], n);
```

```
1 5
```

```
5 0
```

```
5 5
```

```
1 0
```

EXPLICATION

- Un tableau passé en argument est converti en **pointeur** vers son premier élément: `f` modifie donc directement `t[0]` dans le bloc appelant;
- Un `int` est passé **par valeur**: `f` modifie une **copie** de `n`, sans effet sur le bloc appelant.

Question 11

Qu'affiche le fragment suivant?

```c
struct Point { int x, y; };

void deplacer_x(struct Point p) { p.x += 10; }
void deplacer_y(struct Point *p) { p->y += 10; }

// ...
struct Point a = {1, 2};
deplacer_x(a);
deplacer_y(&a);
printf("%d %d\n", a.x, a.y);
```

```
11 12
```

```
1 12
```

```
11 2
```

```
1 2
```

EXPLICATION

- `deplacer_x` reçoit une **copie** de la structure: la modification est perdue au retour de la fonction;
- `deplacer_y` reçoit l'**adresse** de `a`: la modification est visible dans le bloc appelant.

Question 12

On considère le fichier `prog.c` suivant:

```c
#include <stdio.h>

int carre(int x);

int main(void) {
  printf("%d\n", carre(4));
  return 0;
}
```

Qu'est-ce qui se passera si on entre les deux commandes suivantes?

```
$ gcc -c prog.c
$ gcc -o prog prog.o
```

La première commande échoue, car `carre` n'est pas implémentée

La première commande réussit, mais la deuxième échoue

Les deux commandes réussissent, et `./prog` affiche `16`

Les deux commandes réussissent, mais `./prog` produit une erreur de segmentation

EXPLICATION

- **Compilation** (`gcc -c`): la **déclaration** de `carre` suffit au compilateur pour vérifier l'appel `carre(4)`. Le fichier `prog.o` est produit sans erreur;
- **Édition des liens** (`gcc -o prog prog.o`): il faut maintenant trouver le **code** de `carre`, qui n'existe nulle part:

```
prog.c:(.text+0xe): undefined reference to `carre'
collect2: error: ld returned 1 exit status
```

Question 13

Considérez la fonction suivante:

```
bool f(const char *s) {
  for (int i = 0; s[i] != '\0'; ++i)
    if (!isdigit(s[i]))
      return false;
  return true;
}
```

Quel nom devrait-on donner à `f` pour améliorer la lisibilité du code?

`contient_chiffre`

`est_numerique`

`est_vide`

`commence_par_chiffre`

EXPLICATION

La fonction retourne `false` dès qu'un caractère **n'est pas** un chiffre, et `true` si elle atteint la fin de la chaîne: elle vérifie donc que **tous** les caractères sont des chiffres. Par exemple, `f("123")` retourne `true`, mais `f("12a")` retourne `false`.

Le nom `contient_chiffre` ne convient pas: `"12a"` contient des chiffres, mais `f` retourne `false`.

Question 14

On veut tester une fonction `int categorie(int age)` qui doit retourner 0 si `age < 18`, 1 si `18 <= age < 65` et 2 si `age >= 65`. Son implémentation contient une erreur:

```
int categorie(int age) {
  if (age <= 18) return 0;
  if (age < 65) return 1;
  return 2;
}
```

Lequel des cadres de tests suivants **détecte** l'erreur?

```
10, 30, 70
```

```
0, 17, 64, 65
```

```
17, 19, 64, 66
```

```
18, 65
```

EXPLICATION

Seule la valeur 18 révèle l'erreur: on obtient 0 au lieu de 1. Les autres cadres de tests couvrent pourtant toutes les catégories, mais aucun ne teste la valeur exacte de cette **frontière**.

Un bon cadre de tests contient un représentant de chaque classe d'équivalence **et** les valeurs aux **frontières** entre les classes.

Question 15

On considère un programme `somme` qui affiche la somme de deux entiers passés en arguments. S'il ne reçoit pas exactement deux arguments, il écrit `usage: somme A B` sur `stderr` et retourne 1.

```
@test "Argument manquant" {
  run ./somme 2
  # TODO
}
```

Par quoi faut-il remplacer `# TODO` pour que le test **réussisse**?

```
assert_success
```

```
assert_output ''
```

```
refute_output --partial 'usage'
```

```
assert_output --partial 'usage'
```

EXPLICATION

La commande `run` remplit `$output` avec `stdout` **et** `stderr` combinés: `$output` contient donc `usage: somme A B`.

- `assert_output --partial 'usage'` **réussit**;
- `assert_output ''` et `refute_output --partial 'usage'` échouent, puisque la sortie contient le message;
- `assert_success` échoue, car le code de retour vaut 1. Il aurait fallu utiliser `assert_failure`.

Question 16

Qu'affiche le fragment suivant?

```c
const char *s = "pointeur";
const char *p = s + 3;
printf("%c %c %s %d\n", *p, p[2], p + 4, (int)(p - s));
```

```
n e r 3
```

```
i t eur 3
```

```
n e teur 3
```

```
n t r 4
```

EXPLICATION

- `p` pointe vers `s[3]`, soit `'n'`;
- `p[2]` équivaut à `*(p + 2)`, soit `s[5]`, qui vaut `'e'`;
- `p + 4` pointe vers `s[7]`: avec `%s`, on affiche la chaîne à partir de cette adresse, soit `"r"`;
- `p - s` donne le nombre d'éléments qui séparent les deux pointeurs, soit 3.

Question 17

Considérez la fonction suivante:

```c
int max_vu(int x) {
  static int m = 0;
  if (x > m)
    m = x;
  return m;
}
```

Qu'affiche le fragment suivant?

```c
printf("%d ", max_vu(3));
printf("%d ", max_vu(1));
printf("%d\n", max_vu(5));
```

```
3 1 5
```

```
3 3 5
```

```
0 3 5
```

```
3 4 9
```

EXPLICATION

La variable `m` est **statique**: elle est initialisée à 0 une seule fois, et elle conserve sa valeur d'un appel à l'autre. Elle mémorise donc la plus grande valeur reçue jusqu'à présent: 3, puis toujours 3 (car 1 < 3), puis 5.

Sans `static`, `m` serait réinitialisée à chaque appel, et on obtiendrait `3 1 5`.

Question 18

Qu'affiche le programme suivant?

```c
#include <stdio.h>

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a * b; }

int main(void) {
  int (*ops[])(int, int) = {add, sub, mul};
  printf("%d\n", ops[1](ops[2](2, 3), ops[0](1, 1)));
  return 0;
}
```

-1

4

8

Rien, le programme ne compile pas

EXPLICATION

`ops` est un **tableau de pointeurs de fonctions**: `ops[0]` est `add`, `ops[1]` est `sub` et `ops[2]` est `mul`. L'expression devient donc `sub(mul(2, 3), add(1, 1))`, soit `sub(6, 2) = 4`.

Question 19

Qu'affiche le fragment suivant?

```
struct Etudiant { char nom[10]; int note; };

int cmp(const void *a, const void *b) {
  const struct Etudiant *x = a, *y = b;
  return y->note - x->note;
}

// ...
struct Etudiant t[] = {{"Ana", 72}, {"Bo", 85}, {"Cy", 64}};
qsort(t, 3, sizeof(struct Etudiant), cmp);
printf("%s %s %s\n", t[0].nom, t[1].nom, t[2].nom);
```

```
Ana Bo Cy
```

```
Cy Ana Bo
```

```
Bo Ana Cy
```

```
Cy Bo Ana
```

EXPLICATION

`qsort` place `a` avant `b` lorsque `cmp(a, b)` est **négatif**. Ici, on retourne `y->note - x->note`, qui est négatif lorsque `x->note > y->note`: les étudiants sont triés par note **décroissante** (85, 72, puis 64).

Pour trier par note croissante, il suffirait de retourner `x->note - y->note`.

Question 20

On considère le programme `calcul.c` suivant:

```c
#include <stdio.h>

int main(void) {
  printf("Résultat: 42\n");
  fprintf(stderr, "Avertissement: valeur arrondie\n");
  return 0;
}
```

On lance la commande `./calcul > sortie.txt`. Que verra-t-on à l'écran, et que contiendra le fichier `sortie.txt`?

- Rien à l'écran; le fichier contient les deux lignes
- `Avertissement: valeur arrondie` à l'écran; le fichier contient `Résultat: 42`
- `Résultat: 42` à l'écran; le fichier contient `Avertissement: valeur arrondie`
- Les deux lignes à l'écran; le fichier est vide

EXPLICATION

L'opérateur `>` redirige seulement la **sortie standard** (`stdout`) vers le fichier. La **sortie d'erreur** (`stderr`) n'est pas redirigée: elle s'affiche toujours à l'écran.

C'est justement l'intérêt d'avoir deux canaux: les messages d'erreur ou d'avertissement restent visibles, sans « polluer » le résultat du programme.