Total /100

Identification

Prénom: .....

Nom: .....

Code permanent: .....

Instructions

- **Aucune documentation** n'est permise. Les rappels nécessaires sont donnés dans les questions.
- Avant de quitter, vous devez présenter votre carte étudiante et signer la feuille de présence.
- Pour chacune des questions à choix multiples, vous devez sélectionner **exactement une** réponse.
- Les questions 1 à 24 valent 80 points; les questions 25 et 26 valent 20 points.
- Pour chacune des questions, vous pouvez supposer que:
  - les programmes sont compilés avec `gcc` (norme C23) sous Linux 64 bits;
  - les en-têtes nécessaires (`<stdio.h>`, `<string.h>`, etc.) sont inclus;
  - `sizeof(char)` retourne 1, `sizeof(int)` retourne 4, `sizeof(double)` retourne 8;
  - les fichiers de tests Bats chargent la bibliothèque `bats-assert`.

Q1. /4 Votre copie locale d'un dépôt Git est à jour. Vous modifiez le fichier `main.c`, puis vous entrez la commande suivante:

```
$ git add main.c
```

Vous ajoutez ensuite une ligne à la fin de `main.c`, vous créez un nouveau fichier `notes.txt`, puis vous entrez les commandes suivantes:

```
$ git commit -m "Corrige l'affichage"
$ git push
```

Pour chacune des affirmations suivantes, indiquez si elle est vraie (V) ou fausse (F) en encerclant la lettre appropriée.

(a) Le *commit* poussé contient la ligne ajoutée à la fin de `main.c` — V / (F)

(b) La commande `git status` mentionne encore `main.c` — (V) / F

(c) La commande `git status` mentionne `notes.txt` — (V) / F

(d) Un coéquipier qui entre `git pull` récupère le fichier `notes.txt` — V / (F)

Q2. /2 Considérez le fichier `prog.c` suivant et le message produit par sa compilation:

```
double moyenne(double a, double b);
int main(void) {
  printf("%.1f\n", moyenne(2.0, 5.0));
  return 0;
}
double moyene(double a, double b) {
  return (a + b) / 2;
}
```

```
$ gcc -Wall -std=c23 -o prog prog.c
/usr/bin/ld: /tmp/ccu0rIvb.o: in function ‘main’:
prog.c:(.text+0x21): undefined reference to ‘
    moyenne’
collect2: error: ld returned 1 exit status
```

À quelle étape l'erreur survient-elle, et pourquoi?

- ☐ À la compilation: `moyenne` est appelée avant d'être définie
- ☐ À la compilation: `moyene` n'est jamais appelée
- ■ À l'édition des liens: `moyenne` est déclarée, mais n'est définie nulle part
- ☐ À l'exécution: l'appel `moyenne(2.0, 5.0)` échoue

Q3. /4 Un répertoire contient uniquement les fichiers `prog.c`, `tests.bats` et le `Makefile` suivant, où chaque commande est identifiée par une lettre en commentaire:

```
.PHONY: test clean

prog: prog.o
  gcc -o prog prog.o          # (A)

prog.o: prog.c
  gcc -std=c23 -c prog.c      # (B)

test: prog
  bats tests.bats             # (C)

clean:
  rm -f prog prog.o           # (D)
```

On entre successivement les commandes du tableau ci-dessous. Pour chacune des lignes, indiquez les lettres des commandes exécutées par `make`, dans l'ordre (ou « aucune »).
Rappel: Make n'exécute les commandes d'une règle que si la cible n'existe pas ou si l'une de ses dépendances est plus récente qu'elle; `touch f` met à jour la date de modification du fichier `f`.

Commande(s)	**Lettres**
`make`	B, A
`make test`	C
`touch prog.c; make test`	B, A, C
`rm prog.o; make`	B, A

Q4. /4 Donnez la valeur, en base 10, de chacun des littéraux entiers suivants:

`0x1F`	`017`	`0b1010`	`0x100`
31	15	10	256

Q5. /4 Qu'est-ce qui sera affiché sur la sortie standard par le fragment de code suivant?

```
int a = 7, b = 2;
double x = a / b;
double y = (double) a / b;
int k = y * 3;
printf("%.1f\n", x);
printf("%.1f\n", y);
printf("%d\n", k);
printf("%d\n", k / b % 3);
```

Rappel: les opérateurs `*`, `/` et `%` ont la même priorité et sont évalués de gauche à droite; la conversion explicite `(double)` est prioritaire sur ces opérateurs.

(a) Ligne 1: ..... 3.0 .....
(b) Ligne 2: ..... 3.5 .....
(c) Ligne 3: ..... 10 .....
(d) Ligne 4: ..... 2 .....

Q6. /3 Qu'est-ce qui sera affiché sur la sortie standard par le fragment de code suivant? *Suggestion:* faites une trace à la main.

```
int n = 0, i = 10;
do {
  i -= 3;
  if (i % 2 == 0)
    continue;
  n += i;
} while (i > 0);
printf("%d %d\n", n, i);
```

Rappel: dans une boucle `do-while`, l'instruction `continue` passe directement à l'évaluation de la condition.
Réponse: ..... 8 -2 .....

Q7. /4 On considère la variable suivante:

```
uint8_t x = 0xB4;    // 1011 0100
```

Donnez la valeur, en hexadécimal sur 8 bits, de chacune des expressions suivantes. Les expressions sont *indépendantes*: aucune ne modifie `x`.
Rappel: `uint8_t`, déclaré dans `<stdint.h>`, est un type entier non signé sur exactement 8 bits (valeurs de 0 à 255); `|` (ou bit à bit), `&` (et bit à bit), `^` (ou exclusif bit à bit), `>>` (décalage vers la droite).

(a) `x & 0x0F`: ..... 0x04 .....
(b) `x | 0x03`: ..... 0xB7 .....
(c) `x ^ 0xFF`: ..... 0x4B .....
(d) `x >> 4`: ..... 0x0B .....

Q8. /3 Considérez le programme suivant:

```
int main(void) {
  int c, n = 0;
  while ((c = getchar()) != EOF
         && c != '\n')
    if (isdigit(c))
      n = 10 * n + (c - '0');
  printf("%d\n", n);
  return 0;
}
```

Rappel: `isdigit(c)` retourne une valeur non nulle si et seulement si `c` est un chiffre (de `'0'` à `'9'`); dans le terminal, `^D` (*Ctrl+D*) est un raccourci pour envoyer la fin de fichier (`EOF`).
Qu'est-ce qui sera affiché sur la sortie standard si on entre le texte suivant?

```
a1b2 3
c4
^D
```

Réponse: ..... 123 .....

Q9. /4 Considérez le fragment de code suivant:

```
char s[12] = "bonjour";
char t[] = "bon";
s[3] = '\0';
```

Une fois ce fragment exécuté, que vaut chacune des expressions suivantes?
Rappel: `strlen(s)` retourne le nombre de caractères de `s` qui précèdent le premier `'\0'`; `strcmp(s, t)` retourne `0` si et seulement si les chaînes `s` et `t` sont égales.

Expression	**Valeur**
`strlen(s)`	3
`sizeof(s)`	12
`sizeof(t)`	4
`strcmp(s, t) == 0`	1

Q10. /4 Considérez le fragment de code suivant, où `texte` est une chaîne de caractères:

```
int h = -1, m = -1;
int n = sscanf(texte, "%d:%d",
               &h, &m);
```

Rappel: `sscanf` lit les valeurs dans l'ordre, ignore les espaces qui précèdent un entier, s'arrête dès que le texte ne correspond plus au format attendu, et retourne le nombre de variables affectées.
Complétez le tableau suivant:

`texte`	`n`	`h`	`m`
`"14:30"`	2	14	30
`"9h15"`	1	9	-1
`"h12"`	0	-1	-1
`" 7:45 fin"`	2	7	45

Q11. /2 Qu'est-ce qui sera affiché sur la sortie standard par le fragment de code suivant?

```
char s[] = "lun,,mer;jeu,";
int n = 0;
char *t = strtok(s, ",;");
while (t != NULL) {
  printf("[%s]", t);
  ++n;
  t = strtok(NULL, ",;");
}
printf(" %d\n", n);
```

Rappel: `strtok(s, delims)` retourne le premier segment de `s` délimité par un des caractères de `delims`; les appels suivants, avec `NULL` comme premier argument, retournent les segments suivants de la même chaîne, puis `NULL` lorsqu'il n'y en a plus.

Réponse: ..... `[lun][mer][jeu] 3` .....

Q12. /2 On souhaite écrire une fonction `ordonner` telle qu'après l'appel `ordonner(&x, &y)`, les variables `x` et `y` contiennent les deux mêmes valeurs qu'avant l'appel, mais avec `x <= y`. Par exemple, si `x` vaut `5` et `y` vaut `2`, alors `x` doit valoir `2` et `y`, `5`. Considérez les quatre versions suivantes:

Version A

```
void ordonner(int *a, int *b) {
  if (*a > *b) { int *t = a; a = b; b = t; }
}
```

Version B

```
void ordonner(int *a, int *b) {
  if (*a > *b) { *a = *b; *b = *a; }
}
```

Version C

```
void ordonner(int *a, int *b) {
  if (*a > *b) { int t = *a; *a = *b; *b = t; }
}
```

Version D

```
void ordonner(int *a, int *b) {
  if (a > b) { int t = *a; *a = *b; *b = t; }
}
```

Quelle version est correcte?

☐ A ☐ B ■ C ☐ D

Q13. /4 Qu'est-ce qui sera affiché sur la sortie standard par le fragment de code suivant?

```
struct Article {
  char nom[10]; int qte;
};
void ajouter(struct Article *a, int n) {
  a->qte += n;
}
void vider(struct Article a) {
  a.qte = 0;
}
// ...
struct Article stock[] = {
  {"pomme", 5}, {"poire", 2}
};
struct Article *p = &stock[1];
ajouter(p, 3);
vider(stock[0]);
ajouter(stock, p->qte);
printf("%d %d\n", stock[0].qte,
       stock[1].qte);
struct Article copie = stock[0];
copie.qte = 1;
printf("%d %c\n", stock[0].qte,
       p->nom[1]);
```

(a) Ligne 1: ..... `10 5` .....

(b) Ligne 2: ..... `10 o` .....

Q14. /4 Considérez le programme `longueurs.c` suivant:

```
int main(int argc, char *argv[]) {
  int total = 0;
  for (int i = 1; i < argc; ++i)
    total += strlen(argv[i]);
  printf("%d %d\n", argc, total);
  return 0;
}
```

Une fois le programme compilé, qu'est-ce qui sera affiché sur la sortie standard par chacune des commandes suivantes?

Commande	**Affichage**
`./longueurs`	1 0
`./longueurs ab cde`	3 5
`./longueurs "ab cde"`	2 6
`./longueurs '' x`	3 1

Q15. /2 Le dépôt GitLab du projet de la question 3 contient le fichier `.gitlab-ci.yml` suivant:

```
stages: [build, test]
build:
  stage: build
  script:
    - make
test:
  stage: test
  script:
    - make test
```

Vous poussez un *commit* dans lequel `prog.c` contient une erreur de compilation. Que se passe-t-il?

- ☐ Le `git push` est refusé
- ■ L'étape `build` échoue, et l'étape `test` n'est pas exécutée
- ☐ L'étape `build` échoue, mais l'étape `test` réussit avec l'ancienne version de `prog`
- ☐ Les deux étapes réussissent

Q16. /2 Le programme `inverser` affiche ses arguments dans l'ordre inverse, un par ligne. Par exemple, `./inverser a b` affiche `b`, puis `a` sur la ligne suivante. Considérez le test Bats suivant:

```
@test "Trois arguments" {
  run ./inverser un deux trois
  assert_success
  # TODO
}
```

Rappel: `assert_line TEXTE` vérifie qu'*au moins une* ligne de la sortie est exactement `TEXTE` (avec `--index i`: la ligne d'indice `i`, à partir de `0`); `assert_output --partial TEXTE` vérifie que la sortie *contient* `TEXTE`.
Par quelles **deux** lignes doit-on remplacer `# TODO` pour que le test **réussisse**?

- ☐ `assert_line --index 0 "trois"`
  `assert_output --partial "deux un"`
- ☐ `assert_line --index 1 "deux"`
  `assert_line --index 3 "un"`
- ■ `assert_line "trois"`
  `assert_line --index 2 "un"`
- ☐ `assert_line --index 2 "un"`
  `assert_output "trois deux un"`

Q17. /4 On teste le programme `longueurs` de la question 14 avec le fichier `tests.bats` suivant:

```
@test "Sans argument" {
  run ./longueurs
  assert_success
  assert_output "1 0"
}
@test "Deux arguments" {
  run ./longueurs ab cde
  assert_output --partial "5"
}
@test "Argument vide" {
  skip
  run ./longueurs ''
  assert_output "2 1"
}
@test "Guillemets" {
  run ./longueurs "a b"
  assert_output "3 2"
}
```

Rappel: `assert_output --partial TEXTE` vérifie que la sortie *contient* `TEXTE`; `skip` désactive le test.
Complétez le rapport affiché par la commande `bats --tap tests.bats`, en indiquant pour chaque test `ok`, `not ok` ou `ok # skip`:

Test	Résultat
1 `Sans argument`	ok
2 `Deux arguments`	ok
3 `Argument vide`	ok # skip
4 `Guillemets`	not ok

Q18. /2 Selon les conventions de nomenclature vues en classe, quel nom convient le mieux à la fonction `f` suivante?

```
unsigned int f(const char *s, char c) {
  unsigned int k = 0;
  for (; *s != '\0'; ++s)
    if (*s == c)
      ++k;
  return k;
}
```

- ☐ `contains_char`
- ■ `num_occurrences`
- ☐ `print_occurrences`
- ☐ `get_k`

Q19. /4 Qu'est-ce qui sera affiché sur la sortie standard par le fragment de code suivant?

```
int t[] = {3, 1, 4, 1, 5, 9};
int *p = t, *q = &t[5];
p += 2;
printf("%d\n", *p);
printf("%d\n", *(q - 1) + 1);
printf("%d\n", (int) (q - p));
q = p + 1;
*q = 7;
printf("%d %d\n", t[3], p[1] + p[-1]);
```

(a) Ligne 1: ..... 4 .....
(b) Ligne 2: ..... 6 .....
(c) Ligne 3: ..... 3 .....
(d) Ligne 4: ..... 7 8 .....

Q20. /4 On considère une fonction dont la signature est la suivante:

```
void analyser(const char *s);
```

Pour chacune des instructions suivantes, placée dans le corps de `analyser`, indiquez si elle est acceptée (A) par le compilateur ou si elle provoque une erreur (E) de compilation en encerclant la lettre appropriée.

(a) `s++;` (A) / E
(b) `s[0] = 'x';` A / (E)
(c) `char c = *s;` (A) / E
(d) `*(s + 1) = 'y';` A / (E)

Q21. /4 Qu'est-ce qui sera affiché sur la sortie standard si on exécute le programme suivant?

```
int compteur = 0;

int prochain_id(void) {
  static int id = 100;
  compteur++;
  return id++;
}

int main(void) {
  int a = prochain_id();
  int b = prochain_id();
  int id = 5;
  int c = prochain_id() + id;
  printf("%d %d %d\n", a, b, c);
  printf("%d %d\n", compteur, id);
  return 0;
}
```

(a) Ligne 1: ..... 100 101 107 .....
(b) Ligne 2: ..... 3 5 .....

Q22. /3 Considérez les fonctions suivantes:

```
int triple(int x) { return 3 * x; }
int carre(int x)  { return x * x; }

int iteration(int (*f)(int),
              int x, int n) {
  for (int i = 0; i < n; ++i)
    x = f(x);
  return x;
}
```

Complétez le tableau suivant:

Expression	Valeur
`iteration(triple, 3, 2)`	27
`iteration(carre, 2, 3)`	256
`iteration(carre, 5, 0)`	5

Q23. /3 Qu'est-ce qui sera affiché sur la sortie standard par le fragment de code suivant?

```
int ordre_parite(const void *a,
                 const void *b) {
  int x = *(const int *)a;
  int y = *(const int *)b;
  if (x % 2 != y % 2)
    return x % 2 - y % 2;
  return x - y;
}
// ...
int t[] = {5, 2, 8, 1, 4, 7};
qsort(t, 6, sizeof(int), ordre_parite);
for (int i = 0; i < 6; ++i)
  printf("%d ", t[i]);
```

Rappel: `qsort` trie le tableau de telle sorte que `a` précède `b` lorsque `ordre_parite(&a, &b)` est négatif.

Réponse: ..... 2 4 8 1 5 7 .....

Q24. /4 Considérez le programme `double.c` suivant:

```
void verifier(int x) {
  if (x < 0) {
    fprintf(stderr, "neg: %d\n", x);
    exit(3);
  }
}
int main(int argc, char *argv[]) {
  for (int i = 1; i < argc; ++i) {
    int x = atoi(argv[i]);
    verifier(x);
    printf("%d\n", 2 * x);
  }
  return 0;
}
```

Rappel: `atoi(s)` convertit `s` en entier; `exit(n)` termine immédiatement le programme avec le code de retour `n`. On entre ensuite les commandes suivantes:

```
$ ./double 4 -1 5 2> err.txt
$ echo $?
$ ./double 4 5 > out.txt
$ echo $?
```

(a) Affichage à l'écran (1re commande): .. 8 ..
(b) Contenu de `err.txt`: ..... neg: -1 .....
(c) Affichage de la 2e commande: ..... 3 .....
(d) Affichage de la 4e commande: ..... 0 .....

Q25. /10 On souhaite retirer les valeurs strictement négatives d'un tableau d'entiers, *sans utiliser de tableau supplémentaire*. Les valeurs conservées sont regroupées au début du tableau, dans leur ordre d'origine, et la fonction retourne leur nombre.

(a) **(6 points)** Écrivez la définition de la fonction suivante, documentée par sa *docstring*:

```
/**
 * Retire les valeurs strictement negatives d'un tableau.
 *
 * Les valeurs conservees sont placees au debut du tableau,
 * dans leur ordre d'origine. Le contenu des cases suivantes
 * n'a pas d'importance.
 *
 * @param t  Le tableau a modifier
 * @param n  Le nombre d'elements de t
 * @return   Le nombre de valeurs conservees
 */
unsigned int retirer_negatifs(int *t, unsigned int n);
```

Par exemple, le fragment de code suivant affiche les valeurs indiquées en commentaire:

```
int t[] = {4, -2, 0, -7, 9};
unsigned int k = retirer_negatifs(t, 5);
printf("%u: %d %d %d\n", k, t[0], t[1], t[2]);  // 3: 4 0 9
```

(b) **(4 points)** Proposez quatre cas de tests pour la fonction `retirer_negatifs`, chacun formé d'une entrée (`t` et `n`) et du résultat attendu (valeur retournée et début du tableau). Au moins deux d'entre eux doivent être des *cas limites*. Justifiez brièvement le choix de chacun.

Solution (a)

```
unsigned int retirer_negatifs(int *t, unsigned int n) {
  unsigned int k = 0;
  for (unsigned int i = 0; i < n; ++i)
    if (t[i] >= 0)
      t[k++] = t[i];
  return k;
}
```

L'indice `k` désigne la prochaine case libre au début du tableau: comme `k <= i` en tout temps, on n'écrase jamais une valeur qui n'a pas encore été examinée. *Barème suggéré:* parcours correct des `n` éléments (1 point); condition `t[i] >= 0` (et non `> 0`) (1 point); recopie au début du tableau en conservant l'ordre (2 points); valeur retournée correcte (1 point); lisibilité (1 point). Une solution qui utilise un tableau supplémentaire perd les points de la recopie.

Solution (b) Par exemple:

- `n = 0` → `0`: cas limite, tableau vide (la boucle ne s'exécute pas);
- `{-3, -1}` → `0`: cas limite, toutes les valeurs sont retirées;
- `{2, 0, 5}` → `3`, `{2, 0, 5}`: cas limite, aucune valeur n'est retirée (et `0` est conservé);
- `{-1, 6, 8, -2}` → `2`, `{6, 8}`: valeurs négatives aux extrémités;
- `{4, -2, 0, -7, 9}` → `3`, `{4, 0, 9}`: cas normal.

Barème suggéré: 1 point par cas pertinent, avec résultat attendu correct et justification; au plus 2 points si moins de deux cas limites.

Q26. /10 On souhaite écrire un programme `moyenne` qui reçoit des notes entières sur la ligne de commande et qui affiche leur moyenne, avec une décimale. Une note est *valide* si c'est un entier compris entre 0 et 100 (inclusivement).
Rappel: `sscanf(s, "%d", &n)` retourne `1` si un entier a été lu au début de la chaîne `s` et stocké dans `n`, et `0` sinon.

(a) **(5 points)** Écrivez la définition de la fonction suivante, documentée par sa *docstring*:

```
/**
 * Lit une note dans une chaine de caracteres.
 *
 * @param s     La chaine a lire
 * @param note  L'adresse ou stocker la note lue
 * @return      true si une note valide a ete lue, false sinon
 */
bool lire_note(const char *s, int *note);
```

(b) **(5 points)** Complétez la fonction `main` suivante, en utilisant `lire_note`:

```
int main(int argc, char *argv[]) {
  if (argc < 2) {
    fprintf(stderr, "usage: moyenne NOTE...\n");
    return 1;
  }
  // TODO
}
```

Si toutes les notes sont valides, le programme affiche leur moyenne sur la sortie standard et retourne `0`. Sinon, il écrit `note invalide:` suivi de la première note invalide sur la sortie d'erreur et retourne `2`, sans rien afficher sur la sortie standard. Par exemple:

```
$ ./moyenne 70 75
72.5
$ ./moyenne 80 abc 90
note invalide: abc
$ echo $?
2
```

Solution (a)

```
bool lire_note(const char *s, int *note) {
  return sscanf(s, "%d", note) == 1
      && *note >= 0 && *note <= 100;
}
```

Grâce à l'évaluation paresseuse de `&&`, la note n'est comparée aux bornes que si elle a effectivement été lue.
Barème suggéré: appel de `sscanf` avec `note` (et non `&note`, qui est déjà une adresse) (2 points); vérification de la valeur de retour (1,5 point); vérification des bornes avec `*note` (1,5 point).

Solution (b)

```
  int somme = 0, note;
  for (int i = 1; i < argc; ++i) {
    if (!lire_note(argv[i], &note)) {
      fprintf(stderr, "note invalide: %s\n", argv[i]);
```

```
      return 2;
    }
    somme += note;
  }
  printf("%.1f\n", (double) somme / (argc - 1));
  return 0;
```

Barème suggéré: boucle sur les arguments à partir de l'indice `1` et appel de `lire_note` avec `&note` (2 points); message sur `stderr` et `return 2` dès la première note invalide (1,5 point); moyenne calculée en `double` (et non par division entière), divisée par `argc - 1` (1 point); affichage avec `%.1f` et `return 0` (0,5 point). Une solution qui affiche quelque chose sur la sortie standard avant de détecter une note invalide perd des points.