INF203 - Partiel
Année 2025-2026

Durée : 2h.
Une feuille A4 manuscrite recto/verso autorisée, dictionnaire papier (non annoté) autorisé pour les étudiants étrangers uniquement.
Tout autre document, calculatrices et appareils électroniques interdits.
Pour chaque question, une partie des points peut tenir compte de la présentation.
Le barème est indicatif.

Toute réponse même incomplète sera valorisée à partir du moment où elle réalise au moins une partie de ce qui est demandé. Les questions sont relativement indépendantes, et dans tous les cas vous pouvez utiliser les scripts et les fonctions demandées dans les questions précédentes même si vous n'avez pas réussi à les écrire.

Par ailleurs, tout code ou algorithme *élégant*, en plus d'être correct, sera bonifié.

ENGLISH TRANSLATION : an English translation of the questions is available on page 10.

1 Script de sauvegarde avec vérification des arguments et gestion des versions (12 points)

1.1 Énoncé

Vous êtes un administrateur système dans une entreprise, votre responsable vous confie une mission : automatiser la sauvegarde des fichiers essentiels de l'entreprise. Vous devez écrire un script Bash nommé `sauvegarde.sh` pour y parvenir.

Nous allons écrire ce script progressivement tout au long de l'énoncé. À la fin, la réunion de tout le code que vous aurez écrit devrait constituer un script complet.

1.2 Vérification des arguments et existence du répertoire

Avant de procéder à une sauvegarde, il faut s'assurer que l'utilisateur a bien spécifié quel répertoire sauvegarder et que ce répertoire existe. Il n'est pas pensable, sur ce genre de données, de commettre une erreur qui pourrait entraîner une tentative de sauvegarde d'un chemin invalide, ce qui compromettrait l'intégrité des données.

Question 1. (3 points) Écrivez un script qui :

- Vérifie que l'utilisateur a fourni un argument (le répertoire à sauvegarder).
- Vérifie que l'argument correspond bien à un répertoire existant.
- Affiche un message d'erreur approprié si l'argument est manquant ou si le répertoire n'existe pas.

Exemple d'exécution incorrecte (aucun argument fourni) :

```
./sauvegarde.sh
```

Sortie attendue :

```
Erreur : veuillez spécifier un répertoire à sauvegarder.
```

Exemple d'exécution incorrecte (répertoire inexistant) :

```
./sauvegarde.sh dossier_inexistant
```

Sortie attendue :

```
Erreur : le répertoire dossier_inexistant n'existe pas.
```

Solution:

```
#!/bin/bash

# Vérifier qu'un argument a été fourni
if [ $# -ne 1 ]
then
    echo "Erreur : veuillez spécifier un répertoire."
    exit 1
fi

# Vérifier que l'argument est un répertoire existant
if [ ! -d "$1" ]
then
    echo "Erreur : le répertoire $1 n'existe pas."
    exit 2
fi

echo "Répertoire $1 trouvé, préparation de la sauvegarde..."
```

Barème

- ½ : utilisation des variables automatiques $# et $1
- 1 : utilisation de if/then et structure correcte du code
- 1 : conditions booléennes : tests appropriées, opérateur !
- ½ : affichages corrects

On ne pénalise pas l'absence de guillemets ou de exit.

1.3 Création du dossier de sauvegarde avec date

Pour permettre une meilleure gestion, il faut que chaque sauvegarde soit stockée dans un répertoire qui lui soit propre. Elles seront donc étiquetées avec la date du jour en respectant la convention suivante : `<repertoire_à_sauvegarder>_backup_YYYYMMDD`

(où YYYYMMDD correspond à l'année, au mois et au jour).

Afin d'optimiser l'espace disque et d'accélérer le processus, on va comparer la dernière sauvegarde effectuée avec l'état actuel des fichiers. Cela permettra de conserver un historique pertinent des modifications.

Question 2. (2 points) Écrivez un script qui :

- Détermine s'il existe déjà un ou plusieurs répertoires de sauvegarde de la forme `<repertoire_à_sauvegarder>_backup_YYYYMMDD` (pas forcément du jour d'aujourd'hui !)
- Si oui, mémorise le nom du plus récent de ces répertoires dans une variable `LAST_BACKUP`.
  Indice : grâce à la convention de nommage, c'est le dernier par ordre alphabétique.
- Sinon, indique qu'il n'y a pas d'ancienne sauvegarde.

Exemple d'exécution correcte :

```
./sauvegarde.sh mon_dossier
```

Sortie attendue :

```
Dernière sauvegarde : mon_dossier_backup_20241224
```

Solution:

```
LAST_BACKUP=$(ls -d $1_backup_* | tail -n 1)

if [ -z $LAST_BACKUP ]
then
    echo Pas d\'ancienne sauvegarde
else
    echo Dernière sauvegarde : $LAST_BACKUP
fi
```

On peut imaginer beaucoup d'autres stratégies :

- tester le code de retour de ls pour savoir si un tel fichier existe
- compter le nombre de répertoires de cette forme et le comparer à 0
- comparer `LAST_BACKUP` à la chaîne `""`

Barème

- ½ : utilisation de ls avec un pattern * qui capture les bons répertoires
- ½ : utilisation de tail avec la bonne option
- ½ : utilisation correcte du pipe
- ½ : proposition d'un test pertinent pour déterminer si une sauvegarde existe déjà

L'utilisation de l'option –d n'est pas exigée.

Le test demandé est un peu inhabituel, on cherche à voir si l'étudiant saurait le mettre au point sur une machine. On ne pénalise pas la structure du if/then/else.

Dans le cas où une sauvegarde a déjà été réalisée dans la journée, elle devra être mise à jour, sinon un nouveau dossier sera créé.

Question 3. (2 points) Écrivez un script qui :

- Vérifie si une sauvegarde datée **du jour d'aujourd'hui** existe déjà.
- Si elle existe, signale qu'on va mettre à jour la sauvegarde existante.
- Sinon, crée un nouveau répertoire de sauvegarde en ajoutant la date du jour (`_backup_YYYYMMDD`) au nom du dossier et affiche un message correspondant.

Indice : `date +"%Y%m%d"` *permet d'obtenir la date du jour au format* `YYYYMMDD`.

Exemple d'exécution correcte :

```
./sauvegarde.sh mon_dossier
```

Sortie attendue :

```
Mise à jour de la sauvegarde existante : mon_dossier_backup_20260309
OU
Création de la sauvegarde : mon_dossier_backup_20260309
```

Solution:

```
# Obtenir la date du jour
DATE=$(date +"%Y%m%d")
BACKUP_DIR="$1_backup_$DATE"

# Vérifier si le répertoire de sauvegarde existe déjà
if [ -d "$BACKUP_DIR" ]
then
    echo "Mise à jour de la sauvegarde existante : $BACKUP_DIR"
else
    echo "Création de la sauvegarde : $BACKUP_DIR"
    mkdir "$BACKUP_DIR"
fi
```

Barème

- ½ : utilisation correcte de $()
- ½ : construction du nom du répertoire de sauvegarde
- ½ : utilisation de else avec la bonne structure
- ½ : utilisation de mkdir

Si un else ou un $() ont déjà été utilisés correctement à une question précédente on peut donner les points ici.

1.4 Copie des fichiers modifiés

On s'intéresse maintenant à la copie proprement dite. Par souci de simplicité, **on ne copiera que les fichiers du répertoire spécifié et on ignorera ses sous-répertoires.**

Par ailleurs, seule la copie des fichiers modifiés depuis la dernière sauvegarde doit être effectuée. Pour cela, le test `[ fich1 -nt fich2 ]` peut être utilisé dans un script Bash pour déterminer si `fich1` est plus récent que `fich2`.

Question 4. (4,5 points) Rappel : le répertoire `LAST_BACKUP` a été déterminé à la question 2 ; par souci de simplicité on supposera ici que ce répertoire existe.

Écrivez un script qui :

— Copie dans le nouveau répertoire de sauvegarde uniquement les fichiers qui n'existaient pas dans `LAST_BACKUP` ou qui sont plus récents que le fichier de même nom dans `LAST_BACKUP`.

— Affiche la date de dernière modification de chaque fichier copié.

Indice : `ls -l --time-style=long-iso <mon_fichier>` *est une commande qui permet d'obtenir le type de sortie suivante :*

```
-rw-rw-r-- 1 user usergroup    0 2025-02-11 21:05 <mon_fichier>
```

Exemple d'exécution :

```
./sauvegarde.sh mon_dossier
```

Sortie attendue :

```
Sauvegarde en cours...
Copie de mon_fichier (nouvelle version de 2025-02-11 10:33)
Copie de mon_autre_fichier (nouvelle version de 2025-03-15 22:43)
Sauvegarde terminée.
```

Solution:

```
echo "Sauvegarde en cours..."

for fichier in "$1"/*
do
    if [ -f "$fichier" ]
    then
        NOM_FICHIER=$(basename "$fichier")

        if [ ! -f "$LAST_BACKUP/$NOM_FICHIER" -o
             "$fichier" -nt "$LAST_BACKUP/$NOM_FICHIER" ]
        then
            cp "$fichier" "$BACKUP_DIR/"

            # Extraire la date complète
            DATE_SOURCE=$(ls -l --time-style=long-iso" "$fichier" | tr -s ' ' | cut -d '
            echo "Copie de $NOM_FICHIER (nouvelle version de $DATE_SOURCE)"
        fi
    fi
done

echo "Sauvegarde terminée."
```

Barème

— 1 : présence d'un for qui parcourt la bonne liste et construction correcte de la boucle

— 1 : structure du code : if dans for pour ne manipuler que des fichiers, if dans if pour éviter les copies inutiles

— ½ : utilisation correcte de basename
— ½ : construction de la condition booléenne avec –o et –nt
— ½ : utilisation de cp avec les bons arguments
— 1 : détermination de la date avec tr et cut

Question 5. (1,5 points) Que faut-il modifier dans votre script pour compter et afficher le nombre de fichiers ignorés lors de la sauvegarde (parce qu'ils existaient déjà et n'ont pas été modifiés depuis la dernière sauvegarde) ?

Vous pouvez indiquer les endroits à modifier et les lignes à ajouter à votre script, ou vous pouvez réécrire tout ou partie du script proposé pour la question précédente.

Solution:
— on initialise une variable `IGNORES` à zéro en début de script
— on complète le dernier `if` par :

```
else
    IGNORES=$(expr $IGNORES + 1)
```

— on affiche `IGNORES` en fin de script

Barème
— ½ : initialisation de la variable pour compter
— ½ : utilisation de expr pour incrémenter
— ½ : insertion du code supplémentaire aux bons endroits

2 Programmation C : Afficher un calendrier (9 points)

La commande `cal` affiche le calendrier du mois en cours, par exemple :

```
benjamin@bagheera:~$ cal
     Mars 2026
lu ma me je ve sa di
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

L'objectif de l'exercice est d'écrire un programme en C pour produire ce type d'affichage. On ne tiendra pas compte des années bissextiles (années contenant 29 jours en février).

Pour cela, on définit tout d'abord un tableau contenant les noms des mois, comme suit :

```
char noms_mois[12][10] = {"Janvier", "Février", "Mars", "Avril",
                  "Mai", "Juin", "Juillet", "Août",
                  "Septembre", "Octobre", "Novembre", "Décembre" };
```

Pour récupérer le i-ème mois, on pourra utiliser `noms_mois[i-1]`, qui est de type `char[10]`.

On représente par ailleurs les dates par un tableau de 3 entiers (`int`) : jour, mois et année. Par exemple, la date de ce partiel pourrait être stockée dans un tableau `partiel[3]` avec `partiel[0]` qui vaut 9, `partiel[1]` qui vaut 3 et `partiel[2]` qui vaut 2026.

Dans tout cet exercice, on suppose que le programme dispose d'une variable nommée `aujourdhui[3]` qui contient la date du jour où on l'exécute.

Question 1. (1 point) Donnez le code du tableau `jour_dans_mois` contenant 12 entiers tel que le i-ème élément contient le nombre de jours du mois `i` (sans tenir compte des années bissextiles).

Solution:

```
int jours_dans_mois[12]={31,28,31,30,31,30,31,31,30,31,30,31};
```

Barème

- ½ : déclaration du tableau
- ½ : initialisation

Question 2. (2 points) Écrivez une fonction `longueur` qui calcule la longueur d'une chaine de caractères (la bibliothèque `string.h` n'est pas autorisée).

Exemple : `longueur("abcd")` devrait renvoyer 4.

Solution:

```
int len(char *chaine) {
  int i=0;
  while (chaine[i] != '\0')
    i++;
  return i;
}
```

Barème

- ½ : en-tête de la fonction
- ½ : structure du while
- ½ : parcours correct jusqu'au \0
- ½ : renvoi de la bonne valeur

À partir d'ici les codes écrits constitueront la fonction principale de votre programme C.

Question 3. (2,5 points) À l'aide de la fonction précédente, écrivez un bout de code qui détermine combien d'espaces il faut ajouter au début de la première ligne du calendrier pour centrer le nom du mois en cours et l'année, puis qui produit cette affichage centré. On suppose que l'année s'écrit toujours sur 4 chiffres.

Exemple : pour le mois de mars 2026, il faut laisser 4 espaces en début de ligne.

Solution:

```
char *mois = noms_mois[aujourdhui[1]-1];
// On suppose que l'année a toujours 4 chiffres
int padding = (20 - 4 - len(mois)) / 2;
// ou 5 à la place de 4 pour tenir compte de l'espace
for (int i=0 ; i<padding ; i++)
  printf(" ");
printf("%s %d\n", mois, aujourdhui[2]);
```

Barème

- ½ : récupération correcte du mois voulu (d'indice aujourdhui[1]-1)
- ½ : appel correct à la fonction longueur
- ½ : calcul correct du nombre d'espaces nécessaires
- ½ : construction correcte de la boucle for
- ½ : printf correct pour le mois et l'année

Question 4. (1 point) On donne une fonction `int jour_semaine(int date[3])` , qui renvoie un entier indiquant le jour de la semaine correspondant à la date donnée en argument : 0 pour lundi, 1 pour mardi, . . ., 6 pour dimanche. Par exemple `jour_semaine(partiel)` renvoie 0.

Utilisez-la pour déterminer le premier jour du mois en cours (la date d'aujourd'hui est donnée dans le tableau `aujourdhui[3]` en début de programme).

Solution:

```
int premier_jour[3];
premier_jour[0] = 1;
premier_jour[1] = aujourdhui[1];
premier_jour[2] = aujourdhui[2];
int premier_semaine = jour_semaine(premier_jour);
```

Barème

- ½ : construction du premier jour du mois
- ½ : appel correct à jour_semaine avec mémorisation du résultat

On accepte que la variable aujourdhui soit modifiée pour être le premier jour du mois.

Question 5. (2,5 points) Une fois déterminé le premier jour du mois, affichez les jours du calendrier du mois comme dans l'exemple donné plus haut pour la commande `cal` :

- Écrivez les jours de la semaine sur la première ligne (2 caractères par jour)
- Laissez en blanc les jours de la semaine qui précèdent le premier du mois.
- Revenez à la ligne entre les dimanches et les lundis.
- Arrêtez-vous au bon nombre de jours en fonction du mois courant.

Astuce : Pour écrire le numéro du jour systématiquement sur 2 caractères, on pourra utiliser le format `%2d` *(au lieu de* `%d`*) dans la fonction* `printf`.

Solution:

```
    printf("lu ma me je ve sa di\n");

    for(int i=0 ; i<premier_semaine ; i++)
      printf("   ");

    // Version simple : on garde trace du jour courant
    int jour = premier_semaine;
    for(int j=1 ; j<=jours_dans_mois[aujourdhui[1]-1] ; j++) {
      printf("%2d ", j);
      jour = jour + 1;
      if (jour == 7) {
        jour = 0;
        printf("\n");
      }
    }
```

Barème

- — ½ : affichage des jours de la semaine en en-tête
- — ½ : affichage du bon nombre d'espaces (en utilisant la valeur mémorisée plus haut)
- — ½ : boucle for sur le bon nombre de jours
- — ½ : affichage avec espacement correct du numéro de chaque jour
- — ½ : retour à la ligne après les dimanches

1 Backup script with argument checking and version management (12 points)

1.1 Task

You are a system administrator in a company, and you are tasked with a mission : to automatize the backup of some critical files. You must write a Bash script named `sauvegarde.sh` to that end.

We will write that script all along the exercise. In the end, the concatenation of all the code you have written should constitute a complete script.

1.2 Argument checking and directory existence

Before proceeding to a backup, we must check that the user did specify which directory to save and that this directory exists. It would not be acceptable, on that kind of data, to make a mistake that may attempt at saving an invalid path, thus corrupting the whole data.

Question 1. (3 points) Write a script that :

- Checks the user has provided an argument (the diretory to be saved).
- Checks the argument is indeed an existing directory.
- Displays an appropriate error message is the argument is missing or the directory does not exist.

Example of an incorrect execution (no argument provided) :

```
./sauvegarde.sh
```

Expected output :

```
Erreur : veuillez spécifier un répertoire à sauvegarder.
```

Example of an incorrect execution (nonexistant directory) :

```
./sauvegarde.sh dossier_inexistant
```

Expected output :

```
Erreur : le répertoire dossier_inexistant n'existe pas.
```

Solution:

```
#!/bin/bash

# Vérifier qu'un argument a été fourni
if [ $# -ne 1 ]
then
    echo "Erreur : veuillez spécifier un répertoire."
    exit 1
fi

# Vérifier que l'argument est un répertoire existant
if [ ! -d "$1" ]
then
    echo "Erreur : le répertoire $1 n'existe pas."
    exit 2
fi

echo "Répertoire $1 trouvé, préparation de la sauvegarde..."
```

Barème

- ½ : utilisation des variables automatiques $# et $1
- 1 : utilisation de if/then et structure correcte du code

— 1 : conditions booléennes : tests appropriées, opérateur !
— ½ : affichages corrects

On ne pénalise pas l'absence de guillemets ou de exit.

1.3 Creating the backup directory with a date stamp

To allow for a better management, each backup must be stored in an individual directory. Thus, they will be labeled with the date of the day using the following convention :

`<directory_to_be_saved>_backup_YYYYMMDD`

(where YYYYMMDD corresponds to the year, month and day).

To optimize disk space and make the process faster, we will compare the last performed backup with the current state of the files. This will allow to keep track of the history of modifications.

Question 2. (2 points) Write a script that :

- Checks whether there already exists at least one directory named `<directory_to_be_saved>_backup_YYYYMMDD` (for any date, not specifically today !)
- If so, stores the name of the newest of those directories in a variable `LAST_BACKUP`.
  Hint : because of the naming convention, it will be the last in alphabetical order.
- Otherwise, indicates there is no previous backup.

Example of a correct execution :

```
./sauvegarde.sh mon_dossier
```

Expected output :

```
Dernière sauvegarde : mon_dossier_backup_20241224
```

Solution:

```
LAST_BACKUP=$(ls -d $1_backup_* | tail -n 1)

if [ -z $LAST_BACKUP ]
then
    echo Pas d\'ancienne sauvegarde
else
    echo Dernière sauvegarde : $LAST_BACKUP
fi
```

On peut imaginer beaucoup d'autres stratégies :

- tester le code de retour de ls pour savoir si un tel fichier existe
- compter le nombre de répertoires de cette forme et le comparer à 0
- comparer `LAST_BACKUP` à la chaîne `""`

Barème

- ½ : utilisation de ls avec un pattern * qui capture les bons répertoires
- ½ : utilisation de tail avec la bonne option
- ½ : utilisation correcte du pipe
- ½ : proposition d'un test pertinent pour déterminer si une sauvegarde existe déjà

L'utilisation de l'option –d n'est pas exigée.

Le test demandé est un peu inhabituel, on cherche à voir si l'étudiant saurait le mettre au point sur une machine. On ne pénalise pas la structure du if/then/else.

In the case where a backup has been made the same day as today, it must be updated, otherwise a new directory must be created.

Question 3. (2 points) Write a script that :

- Checks whether a backup **for today** already exists.
- If it exists, warns that we will update the existing backup.
- Otherwise, creates a new backup directory by adding the suffix (`_backup_YYYYMMDD`) for today's date to the directory name and displays a message.

Hint : `date +"%Y%m%d"` *displays the date for today formatted as* `YYYYMMDD`.

Example of a correct execution :

```
./sauvegarde.sh mon_dossier
```

Expected output :

```
Mise à jour de la sauvegarde existante : mon_dossier_backup_20260309
OU
Création de la sauvegarde : mon_dossier_backup_20260309
```

Solution:

```
# Obtenir la date du jour
DATE=$(date +"%Y%m%d")
BACKUP_DIR="$1_backup_$DATE"

# Vérifier si le répertoire de sauvegarde existe déjà
if [ -d "$BACKUP_DIR" ]
then
    echo "Mise à jour de la sauvegarde existante : $BACKUP_DIR"
else
    echo "Création de la sauvegarde : $BACKUP_DIR"
    mkdir "$BACKUP_DIR"
fi
```

Barème

- ½ : utilisation correcte de $()
- ½ : construction du nom du répertoire de sauvegarde
- ½ : utilisation de else avec la bonne structure
- ½ : utilisation de mkdir

Si un else ou un $() ont déjà été utilisés correctement à une question précédente on peut donner les points ici.

1.4 Copy of the updated files

Now we deal with the actual copy. For the sake of simplicity, **we will only copy the files in the specified directory, and we will ignore its child directories.**

Moreover, only the files which were modified since the previous backup must be copied. To that end, one can use the test `[ fich1 -nt fich2 ]` in a Bash script to know whether `file1` is newer than `file2`.

Question 4. (4,5 points) Reminder : the directory `LAST_BACKUP` has been determined in question 2 ; for simplicity, here we assume that this directory exists.

Write a script that :

— Copies in the new backup directory only the files which did not exist in `LAST_BACKUP` or are newer than the file with the same name in `LAST_BACKUP`.

— Diplays the last modification date for each copied file.

Hint : `ls -l --time-style=long-iso <mon_fichier>` *is a command that outputs :*

```
-rw-rw-r-- 1 user usergroup    0 2025-02-11 21:05 <mon_fichier>
```

Example of execution :

```
./sauvegarde.sh mon_dossier
```

Expected output :

```
Sauvegarde en cours...
Copie de mon_fichier (nouvelle version de 2025-02-11 10:33)
Copie de mon_autre_fichier (nouvelle version de 2025-03-15 22:43)
Sauvegarde terminée.
```

Solution:

```
echo "Sauvegarde en cours..."

for fichier in "$1"/*
do
    if [ -f "$fichier" ]
    then
        NOM_FICHIER=$(basename "$fichier")

        if [ ! -f "$LAST_BACKUP/$NOM_FICHIER" -o
             "$fichier" -nt "$LAST_BACKUP/$NOM_FICHIER" ]
        then
            cp "$fichier" "$BACKUP_DIR/"

            # Extraire la date complète
            DATE_SOURCE=$(ls -l --time-style=long-iso" "$fichier" | tr -s ' ' | cut -d '
            echo "Copie de $NOM_FICHIER (nouvelle version de $DATE_SOURCE)"
        fi
    fi
done

echo "Sauvegarde terminée."
```

Barème

— 1 : présence d'un for qui parcourt la bonne liste et construction correcte de la boucle

— 1 : structure du code : if dans for pour ne manipuler que des fichiers, if dans if pour éviter les copies inutiles

— ½ : utilisation correcte de basename

— ½ : construction de la condition booléenne avec –o et –nt
— ½ : utilisation de cp avec les bons arguments
— 1 : détermination de la date avec tr et cut

Question 5. (1,5 points) What modifications must you make in your script to count and display how many files were ignored during backup (because they already existed and were not modified since the previous backup) ?

You may indicate which parts to modify and which lines to add, or you may rewrite part of the script you wrote for the previous question.

Solution:
— on initialise une variable `IGNORES` à zéro en début de script
— on complète le dernier `if` par :

```
else
    IGNORES=$(expr $IGNORES + 1)
```

— on affiche `IGNORES` en fin de script

Barème
— ½ : initialisation de la variable pour compter
— ½ : utilisation de expr pour incrémenter
— ½ : insertion du code supplémentaire aux bons endroits

2 C programming : calendar display (9 points)

The command `cal` displays the calendar of the current month, for instance :

```
benjamin@bagheera:~$ cal
     March 2026
mo tu we th fr sa su
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31
```

The goal of the exercise is to code in C a program that displays this output. We will NOT take care of leap years (years with 29 days in February).

To this end, we provide a table containing the names of the twelve months of the year as follows :

```
char months_names[12][10] = {"January", "February", "March", "April",
                "May", "June", "July", "August",
                "September", "October", "November", "December" };
```

To extract the i-th month, we may call `months_names[i-1]`, which is of type `char[10]`.

We then write the dates under the form of a table of 3 integers (`int`) : day, month, year. For instance, the date of this midterm exam could be stored in a table `midterm[3]` where `midterm[0]` holds the value 9, `midterm[1]` holds 3 and `midterm[2]` holds 2026.

Throughout this exercise, we assume the program has a variable named `today[3]` which holds the date of its execution.

Question 1. (1 point) Provide the code of the table `days_in_month` containing 12 integers such that the i-th element is the number of days in month `i` (again, not accounting for leap years).

Solution:

```
int jours_dans_mois[12]={31,28,31,30,31,30,31,31,30,31,30,31};
```

Barème

- ½ : déclaration du tableau
- ½ : initialisation

Question 2. (2 points) Write a function `length` that returns the length of a sequence of caracters (the libary `string.h` is not allowed).

Example : `length("abcd")` should return 4.

Solution:

```
int len(char *chaine) {
  int i=0;
  while (chaine[i] != '\0')
    i++;
  return i;
}
```

Barème

- ½ : en-tête de la fonction
- ½ : structure du while
- ½ : parcours correct jusqu'au \0
- ½ : renvoi de la bonne valeur

From here on, the codes form the main function of the C program.

Question 3. (2,5 points) Using the previous function, write a code that determines how many spaces are needed at the beginning of the first line of the calendar to center the name of the month and the year, then prints that first centered line. For simplicity we only consider years written on 4 digits.
Example : for March 2026, we should leave 4 white spaces at the beginning of the first line.

Solution:

```
char *mois = noms_mois[aujourdhui[1]-1];
// On suppose que l'année a toujours 4 chiffres
int padding = (20 - 4 - len(mois)) / 2;
// ou 5 à la place de 4 pour tenir compte de l'espace
for (int i=0 ; i<padding ; i++)
  printf(" ");
printf("%s %d\n", mois, aujourdhui[2]);
```

Barème
- ½ : récupération correcte du mois voulu (d'indice aujourdhui[1]-1)
- ½ : appel correct à la fonction longueur
- ½ : calcul correct du nombre d'espaces nécessaires
- ½ : construction correcte de la boucle for
- ½ : printf correct pour le mois et l'année

Question 4. (1 point) We provide a function `int week_day(int date[3])` that returns an integer representing the day of the week corresponding to the date given as an argument : 0 for Monday, 1 for Tuesday, ..., 6 for Sunday. For instance `week_day(midterm)` returns 0.
Use it to determine the first day of the current month (the date of today is given in the table `today[3]` at the beginning of the program).

Solution:

```
int premier_jour[3];
premier_jour[0] = 1;
premier_jour[1] = aujourdhui[1];
premier_jour[2] = aujourdhui[2];
int premier_semaine = jour_semaine(premier_jour);
```

Barème
- ½ : construction du premier jour du mois
- ½ : appel correct à jour_semaine avec mémorisation du résultat

On accepte que la variable aujourdhui soit modifiée pour être le premier jour du mois.

Question 5. (2,5 points) Once the first day of the month is determined, display all days of the month in the calendar form as in the example given above by the command `cal` :
- Write the days of the week on the first row (2 characters per day)
- Leave white spaces for the days preceding the first day of the month.
- Go back to the beginning of the line after Sundays.
- Stop at the correct number of days in the current month.

Tip : To write the number of a day on 2 characters, we might use the format `%2d` *(instead of* `%d`*) in the function* `printf`*.*

Solution:

```
printf("lu ma me je ve sa di\n");
```

```
for(int i=0 ; i<premier_semaine ; i++)
  printf("   ");

// Version simple : on garde trace du jour courant
int jour = premier_semaine;
for(int j=1 ; j<=jours_dans_mois[aujourdhui[1]-1] ; j++) {
  printf("%2d ", j);
  jour = jour + 1;
  if (jour == 7) {
    jour = 0;
    printf("\n");
  }
}
```

Barème

- — ½ : affichage des jours de la semaine en en-tête
- — ½ : affichage du bon nombre d'espaces (en utilisant la valeur mémorisée plus haut)
- — ½ : boucle for sur le bon nombre de jours
- — ½ : affichage avec espacement correct du numéro de chaque jour
- — ½ : retour à la ligne après les dimanches

A Annexe : rappels

On rappelle ici différentes commandes et options qui peuvent être utiles.

Quelques tests utilisables dans un script bash

option	signification
`-z chaine`	`chaine` est vide
`-n chaine`	`chaine` n'est pas vide
`chaine`$_1$ `=` `chaine`$_2$	les 2 chaines sont identiques
`chaine`$_1$ `!=` `chaine`$_2$	les 2 chaines sont différentes

option	signification
n_1 `-eq` n_2	$n_1 = n_2$
n_1 `-ne` n_2	$n_1 \neq n_2$
n_1 `-lt` n_2	$n_1 < n_2$
n_1 `-le` n_2	$n_1 \leq n_2$

option	signification
`expr`$_1$ `-a` `expr`$_2$	et
`expr`$_1$ `-o` `expr`$_2$	ou
`! expr`	non

option	signification
`-e fich`	`fich` existe
`-f fich`	`fich` est un fichier
`-d fich`	`fich` est un répertoire

basename

`basename NOM` affiche le `NOM` de fichier donné en argument, en supprimant tous les répertoires du chemin donné (s'il y en a)

`basename NOM SUFFIXE` réalise la même opération, et de plus supprime le `SUFFIXE` donné s'il est présent à la fin du `NOM`

cut affiche une partie seulement de chaque ligne de ses entrées.

`-c debut-fin` permet de choisir les caractères affichés ;

`-f debut-fin` permet de choisir des colonnes ;

`-d` permet de choisir le délimiteur.

diff détermine si les contenus de 2 fichiers sont identiques :

— aucune réponse : les 2 fichiers sont identiques.

— la réponse est seulement que les 2 fichiers sont différents : l'un au moins n'est pas un fichier texte.

— la liste des différences entre les 2 fichiers : les 2 fichiers sont des fichiers texte.

expr calcule l'expression décrite par ses arguments.

grep n'affiche que les lignes d'un fichier contenant une chaîne de caractères donnée.

`-c` n'affiche que le nombre total de lignes contenant la chaîne recherchée.

`-v` n'affiche que les lignes ne contenant pas la chaîne donnée.

head et tail

`head -n N` affiche les `N` premières lignes de ses entrées.

De même la commande `tail -n N` affiche les `N` dernières lignes de ses entrées.

sed affiche les lignes de ses entrées en les filtrant et/ou modifiant selon les instructions données :

`sed /Candide/d` affiche celles ne contenant pas `Candide`

`sed s/Candide/Toto/` remplace le premier `Candide` par `Toto` sur chaque ligne

`sed s/Candide/Toto/g` remplace tout les `Candide` par `Toto`

sort affiche les lignes de ses entrées triées par ordre alphabétique croissant

`sort -n` affiche les lignes triées selon l'ordre numérique

`sort -R` trie les lignes de son entrée selon un ordre aléatoire

tr copie l'entrée standard sur la sortie standard en substituant

`tr chaîne1 chaîne2 :` chaque caractère présent dans `chaîne1` par le caractère de position correspondante dans `chaîne2`

`tr -s chaîne :` les répétitions des caractères présents dans `chaîne` par une occurrence unique

wc compte les caractères, les mots ou les lignes lus sur son entrée standard :

`-c` compte les caractères

`-w` compte les mots

`-l` compte les lignes

A Appendix : reminders

We recall various commands and options which may be useful.

A few tests you can use in a bash script

option	meaning
`-z string`	`string` is empty
`-n string`	`string` is not empty
$\texttt{string}_1$ `=` $\texttt{string}_2$	the 2 strings are identical
$\texttt{string}_1$ `!=` $\texttt{string}_2$	the 2 strings sont different

option	meaning
n_1 `-eq` n_2	$n_1 = n_2$
n_1 `-ne` n_2	$n_1 \neq n_2$
n_1 `-lt` n_2	$n_1 < n_2$
n_1 `-le` n_2	$n_1 \leq n_2$

option	meaning
$\texttt{expr}_1$ `-a` $\texttt{expr}_2$	and
$\texttt{expr}_1$ `-o` $\texttt{expr}_2$	or
`! expr`	not

option	meaning
`-e file`	`file` exists
`-f file`	`file` is a file
`-d file`	`file` is a directory

basename

`basename FILENAME` prints the `FILENAME` given as an argument, removing all the directories from the given path (if any)

`basename FILENAME SUFFIX` does the same, and also removes the given `SUFFIX` if present at the end of the `FILENAME`

cut prints only a part of each line in its inputs.

`-c start-end` selects the printed characters ;

`-f start-end` selects the printed columns ;

`-d` determines the delimiter for columns.

diff compares 2 files contents :

— no answer : the 2 files are identical.

— the answer is "The given files are different" : at least one of them is not a text file.

— the answer is the list of differences between the files : both files are text files.

expr computes the expression described by its arguments.

grep prints only the lines in a file containing a given string.

`-c` prints only the total number of lines containing the given string.

`-v` prints only the lines that do not contain the given string.

head and tail

`head -n N` prints the first `N` lines of its inputs.

Similarly, the command `tail -n N` prints the last `N` lines of its inputs.

sed prints the lines in its inputs filtering and/or transforming them according to the given instructions :

`sed /Candide/d` prints the lines not containing `Candide`

`sed s/Candide/Toto/` replaces the first `Candide` with `Toto` on each line

`sed s/Candide/Toto/g` replaces every `Candide` with `Toto`

sort prints the lines of its inputs sorted alphabetically

`sort -n` prints the lines sorted according to the numerical order

`sort -R` prints the lines of its inputs sorted in a random order

tr copies the standard input on the standard output substituting

`tr str1 str2 :` each character listed in `str1` with the corresponding character in `str2`

`tr -s str :` the repeated sequences of characters listed in `str` with a single occurrence

wc counts characters, words or lines read on its standard input :

`-c` counts characters

`-w` counts words

`-l` counts lines