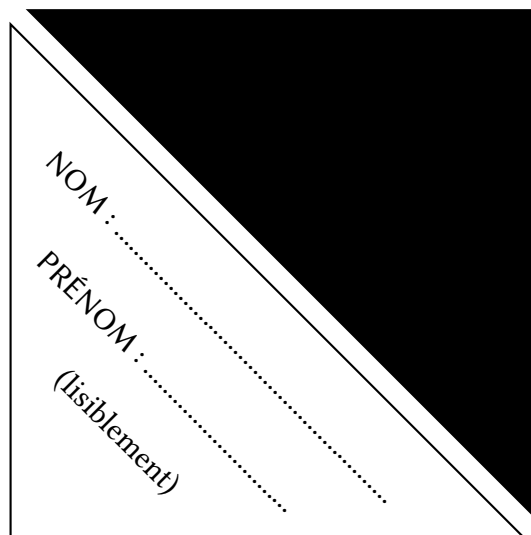


MIDO – L2 MI2E – 2022-2023

Programmation C

Examen du 11 janvier 2023
(durée 2h)



NOM : ...

PRÉNOM : ...

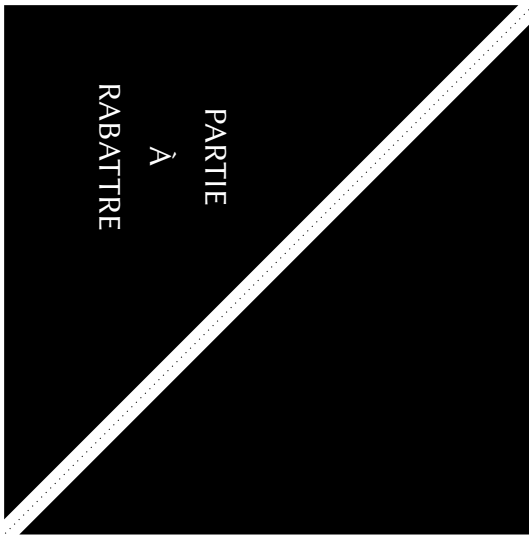
(lisiblement)

Répondez directement sur le sujet.

Documents autorisés : une feuille A4 manuscrite recto-verso

Calculatrice autorisée

Si de la place manque, vous pouvez utiliser les pages supplémentaires situées à la fin.



Exercice 1

Cocher la ou les bonne(s) réponse(s)

1. Dans la définition d'une fonction :

```
int fct(char c, float f) {...}
```

c et f :

- ☐ sont des indications de type;
- ☒ **sont des variables locales;**
- ☐ sont optionnels;
- ☒ **sont des arguments formels.**

2. Un fichier d'entête d'extension .h :

- ☐ contient tout le code source d'un programme;
- ☐ n'est jamais obligatoire même pour un programme fait de plusieurs fichiers sources;
- ☒ **contient les définitions de nouveaux types ainsi que des prototypes de fonctions;**
- ☒ **sert au compilateur pour, entre autres, vérifier les appels aux fonctions externes au fichier.**

3. L'instruction :

```
char *pc = &c;
```

- ☐ définit un caractère pc;
- ☐ définit un pointeur pc toujours initialisé à NULL;
- ☐ définit un pointeur pc non initialisé;
- ☒ **définit un pointeur pc pointant sur une variable c existante.**

4. L'expression `i+=1`

- ☒ **est équivalente à l'expression `++i`**
- ☐ est équivalente à l'expression `i++`
- ☐ est équivalente à l'expression `i=+1`

☐ n'est pas une expression légale en C.

5. Que vaut i après la boucle suivante :

```
int i = 2;
while (1) {
    if (i%2 == i%3)
        break;
    if (++i <= 4)
        continue;
    i += 3;
}
```

☐ 6

☐ 7

☒ 12

☐ c'est une boucle infinie;

☐ il y a en fait une erreur de syntaxe;

Exercice 2

Soit le programme suivant :

```
#include <stdio.h>

int a = -1;

int func(int *X, int *Y, int Z) {
    a++;
    Z = *X;
    *Y = ++(*X) + Z++;
    *X = (*(Y++))++;
    return --*Y + 2*Z;
}

int main() {
    int b = 0;
    int c[3] = {4, 4, 4};
    int d = func(&b, c, b);
    printf("%d %d %d %d %d %d\n", a, b, c[0], c[1], c[2], d);
    return 0;
}
```

L'exécution du programme affichera :

Solution :

0 1 2 3 4 5

Exercice 3

Soit le programme suivant :

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

int main() {
    char str[] = "jcuirs";
    char *p1 = str;
    char *p2 = str + strlen(str) - 1 ;
    do {
        * (--p2) = ++(* (p1++));
    } while (*(p1+1));
    printf("%s\n", str);
    return 0;
}
```

L'exécution du programme affichera :

Solution :

levels

Exercice 4

Une image en noir et blanc est représentée par un ensemble de lignes, chaque ligne étant une chaîne constituée de caractères '0' et '1' représentant les pixels de l'image. Par exemple, "000100110101110" ou "101100000011001" sont des lignes possibles.

On souhaite compresser les lignes en les transformant chacune en tableau de int :

1. le premier élément est le code ascii du premier caractère de la ligne (pour savoir comment commencer);
2. ensuite, chaque élément du tableau donne la longueur d'un bloc de pixels identiques consécutifs, y compris le premier bloc avec le premier caractère;
3. le tableau se termine par la valeur 0.

On a par exemple les compressions suivantes (à droite chaque élément du tableau de int) :

"000100110101110" -> '0' 3 1 2 2 1 1 1 3 1 0

On a '0' comme premier caractère et il fait partie d'un bloc de trois '0' ; ensuite on a un caractère '1' puis deux caractères '0', puis deux caractères '1'...

"101100000011001" -> '1' 1 1 2 6 2 2 1 0

'1' est le premier caractère et il apparait une fois, puis une fois '0', puis deux fois '1', puis six fois '0'...

Écrire une fonction

```
int *comprime(char *str);
```

qui applique l'algorithme ci-dessus à la chaîne passée en argument et renvoie le nouveau tableau d'entiers créé. On ne cherchera pas à optimiser la taille du tableau d'entiers (il peut donc être plus grand que le nombre nécessaire d'éléments).

Solution :

Listing 1. compression de lignes de pixels

```
int *comprime(char *str) {
    int i, j, compteur;
    char c;
    int *new = malloc(sizeof(int)*(strlen(str)+2));
        // '+2' car premier caractère et 0 final
    new[0] = c = str[0];
    i = compteur = 0;
    j = 1;
    while (i <= strlen(str)) { // avec le dernier bloc
        if (str[i] == c)
            compteur++;
        else {
            new[j++] = compteur;
            compteur = 1;
            c = str[i];
        }
        i++;
    }
    new[j] = 0;
    return new;
}
```

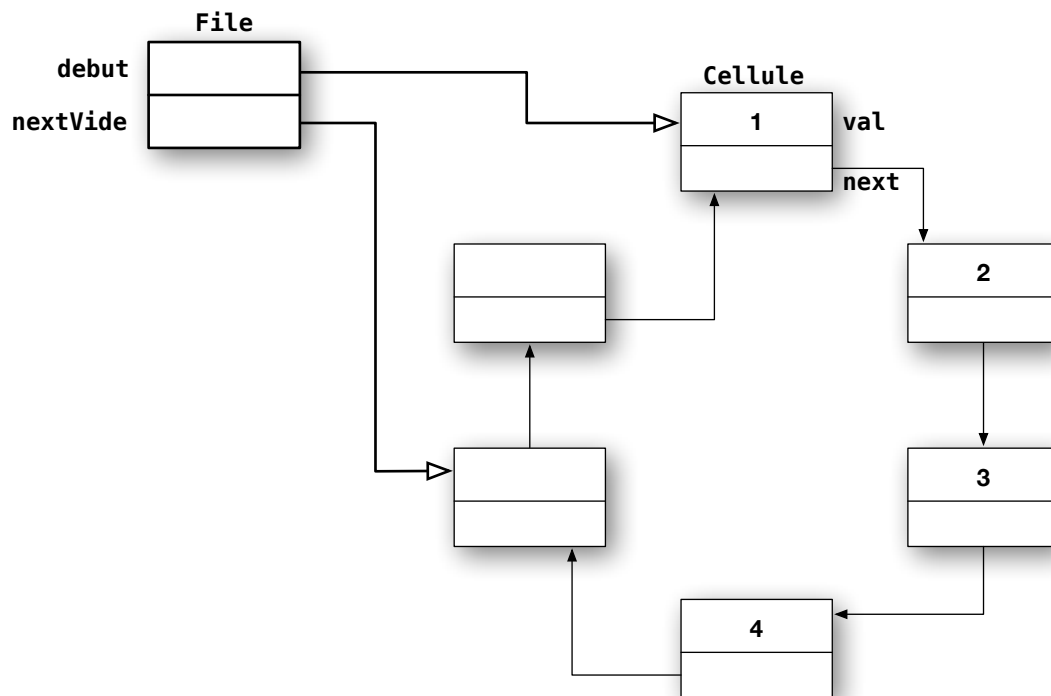
Autre solution :

Listing 2. compression de lignes de pixels

```
int *comprime(char *str) {
    int i, j;
    int *new = malloc(sizeof(int)*(strlen(str)+2));
        // car 1er char et 0 final
    new[0] = str[0];
    j = 1;
    new[1] = 1;
    for(i = 1; i < strlen(str); i++)
        if (str[i] == str[i-1])
            new[j]++;
        else
            new[++j] = 1;
    new[++j] = 0;
    return new;
}
```

Exercice 5

On souhaite gérer une file d'attente FIFO de valeurs entières (pensez à des tickets d'attente portant un numéro dans une boutique). Avec cette file, on doit pouvoir récupérer le premier numéro de la file, et insérer un numéro en queue de file. Il est possible de gérer cette file d'attente avec une structure de liste chaînée mais cela oblige à supprimer une cellule lors du retrait d'un ticket et à en créer une lors de l'insertion, opérations qui peuvent être coûteuses en temps. On décide plutôt d'utiliser une liste circulaire où les cellules sont créées dès le départ et où deux pointeurs indiquent respectivement la première cellule occupée (le début de la file) et la première cellule vide (fin de la file). Ces deux pointeurs sont gérés par une autre structure qui sera le point d'entrée de la file.



Récupérer le premier élément de la file revient à prendre sa valeur puis à déplacer le pointeur `debut` ; insérer un élément revient à initialiser la première cellule vide disponible puis à déplacer le pointeur `nextVide`. L'avantage de cette structure est qu'elle ne nécessite aucune création de cellule mais simplement des déplacements de pointeurs.

Si `debut` et `nextVide` pointent vers la même cellule, cela signifie soit que la file est vide (la première cellule est aussi la première cellule vide), soit qu'elle est pleine (la dernière cellule est occupée et `nextVide` a « fait le tour »). Pour éviter cette ambiguïté, on imposera que la file ne puisse pas être pleine : si on remplit la dernière cellule, il faudra étendre la file en ajoutant une cellule vide.

1. Définir une structure de cellule de file avec un champ `val` pour l'entier et un champ `next` pour le pointeur sur la cellule suivante ainsi qu'un `typedef Cellule` équivalent.
Définir une structure de file avec deux champs `debut` et `nextVide` pointant vers les cellules correspondantes ainsi qu'un `typedef File` équivalent.

Solution :

Listing 3. Cellule et File

```
typedef struct cell {  
    int val;  
    struct cell *next;  
} Cellule;  
  
typedef struct file {  
    Cellule *debut;  
    Cellule *nextVide;  
} File;
```

2. Écrivez une fonction

`File *init(int taille);`

qui crée une structure de file avec autant de cellules qu'indiqué, initialise les bons pointeurs et renvoie un pointeur sur cette file. On pourra supposer que la taille est strictement positive.

Solution :

Listing 4. Initialisation file

```
File *init(int taille) {
    Cellule *p;
    int i;
    File *F = malloc(sizeof(File));
    F->debut = F->nextVide = NULL;
    if (taille > 0) {
        /* on crée une première cellule */
        p = F->nextVide = F->debut = malloc(sizeof(Cellule));
        /* puis toutes les autres */
        for (i = 1; i < taille; i++) {
            p->next = malloc(sizeof(Cellule));
            p = p->next;
        }
        /* et on termine en bouclant sur la première cellule */
        p->next = F->debut;
    }
    return F;
}
```

3. Écrivez une fonction

`int estVide(File *F);`

qui teste si la file passée en paramètre est vide et renvoie 1 dans ce cas et 0 sinon.

Solution :

Listing 5. Test file vide

```
int estVide(File *F) {
    return F->nextVide == F->debut;
}
```


4. Écrivez une fonction

```
void put(File *F, int x);
```

qui insère la valeur x en fin de file. Si la file est alors pleine, il faudra créer une cellule vide supplémentaire en fin de file.

Solution :

Listing 6. Ajouter une valeur

```
void put(File *F, int x) {
    F->nextVide->val = x;
    /* est-elle pleine maintenant ? */
    if (F->nextVide->next == F->debut) {
        F->nextVide->next = malloc(sizeof(Cellule));
        F->nextVide->next->next = F->debut;
    }
    F->nextVide = F->nextVide->next;
}
```

5. Écrivez une fonction

```
int getNext(File *F, int *val);
```

qui récupère la cellule en tête de file et met sa valeur dans la variable passée en paramètre par référence; la fonction renvoie 1 si la valeur a correctement été récupérée et 0 sinon.

Solution :

Listing 7. Récupérer une valeur

```
int getNext(File *F, int *val) {
    if (estVide(F))
        return 0;
    *val = F->debut->val;
    /* on décale le pointeur */
    F->debut = F->debut->next;
    return 1;
}
```

6. Écrivez une fonction

```
void destroy(File *F);
```

qui détruit une file passée en paramètre en libérant proprement la mémoire.

Solution :

Listing 8. Supprimer la file

```
void destroy(File *F) {
    Cellule *p, *q;
    p = q = F->debut;
    p = p->next;
    free(q);
    while (p != F->debut) {
        q = p;
        p = p->next;
        free(q);
    }
    free(F);
}
```

Ou

Listing 9. Supprimer la file

```
void destroy(File *F) {
    Cellule *p, *q;
    p = F->debut;
    do {
        q = p;
        p = p->next;
        free(q);
    } while (p != F->debut);
    free(F);
}
```
