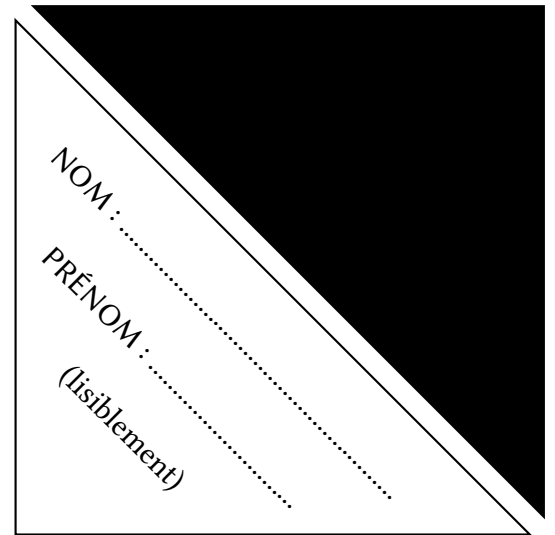


MIDO – L2 MI2E – 2019-2020

Programmation C

Examen du 13 janvier 2020
(durée 2h)

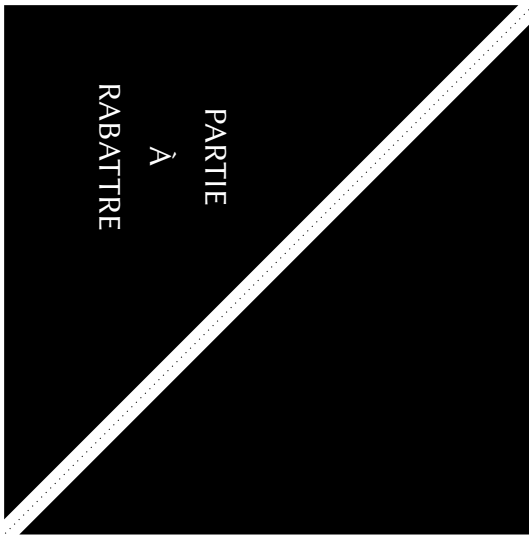


NOM : ...

PRÉNOM : ...

(lisiblement)

Répondez directement sur le sujet.
Si de la place manque, vous pouvez utiliser la page supplémentaire située à la fin.



Exercice 1

Cocher la ou les bonne(s) réponse(s)

1. Avec l'instruction :

```
long *p = malloc(sizeof(long) * 8);
```

- ☐ la déclaration est illégale en C ;
- ☐ je ne peux utiliser p que pour stocker un entier ;
- ☒ **je peux utiliser p comme un identificateur de tableau de 8 entiers longs ;**
- ☐ je peux utiliser p pour stocker autant d'entiers longs que je veux.

2. L'instruction :

```
int i, j = 0;
```

- ☐ déclare deux variables entières, toutes les deux initialisées à zéro ;
- ☐ déclare une variable entière et une autre nulle et sans type ;
- ☒ **déclare deux variables entières, une non-initialisée, l'autre initialisée à zéro.**

3. L'expression :

```
a ? 0 : 1
```

- ☒ **renvoie 0 si a est non-nul et 1 sinon ;**
- ☐ renvoie 0 si a est nul et 1 sinon ;
- ☐ renvoie 1 si a vaut 0 ou 1 ;
- ☐ renvoie 1 si a vaut 1 et 0 sinon ;
- ☐ n'a pas de sens en C.

4. En C, les arguments d'une fonction sont

- ☐ toujours passés par reference ;
- ☒ **toujours passés par valeur ;**
- ☐ parfois passés par référence, parfois par valeur (au choix du programmeur) ;
- ☐ parfois passés par référence, parfois par valeur suivant le type de l'argument (simple ou complexe).

5. Le test :

```
if (!a) {...} /* a est de type int */
```

- ☐ est une erreur de syntaxe;
- ☐ est équivalent à `if (a==1);`
- ☐ est équivalent à `if (a=0) ;`
- ☒ est équivalent à `if (a==0);`
- ☐ est équivalent à `if (a!=0);`
- ☐ est équivalent à `if (a==False).`

Exercice 2

Voici une fonction.

Son rôle est, à partir d'une chaîne de caractère passée en argument, de renvoyer une nouvelle chaîne composée uniquement des chiffres présents dans la chaîne initiale. Ainsi, si la fonction est appelée avec l'argument "10 inventent 1, 2, 3, soleil !", la fonction est supposée renvoyer "10123".

Malheureusement, bien que cette fonction compile sans erreurs ni avertissements, le programmeur y a laissé plusieurs bugs. Saurez-vous les trouver tous ? (*Indiquez les lignes erronées ainsi que leur correction dans l'espace de réponse situé après le listing.*)

Listing 1. Une fonction erronée

```
1 char *extract(char *str) {
2     char *pdigits = malloc(strlen(str));
3     int i = 0;
4
5     do {
6         if (str[i]>'0' && str[i]<='9')
7             *pdigits++ = str[i];
8         if ((str[++i] = '\0'))
9             break;
10    } while (0);
11    return pdigits;
12 }
```

Solution :

- ligne 2, il faut prévoir un octet de plus pour stocker la chaîne en raison du '\0' final;
- ligne 6, le test sera faux pour '0', il faut un >=;
- ligne 8, il faut incrémenter i après récupération (et donc écrire i++); sinon, on dépasse la fin de la chaîne si la chaîne initiale est vide (et que donc str[0] vaut '\0');
- ligne 8, le test correct est un test d'égalité (==) et pas une affectation;
- ligne 10, pour pouvoir faire une boucle infinie, le test du while doit être 1 et pas 0;
- ligne 11, il manque le '\0' final dans la nouvelle chaîne;
- ligne 11, pdigits a été modifié et ne pointe donc plus sur le début de la chaîne; il faut un pointeur de sauvegarde égal à l'adresse allouée.

Une écriture correcte serait donc :

Listing 2. Correction

```
char *extract(char *str) {
    char *pdigits = malloc(strlen(str)+1);
    char *p = pdigits;
    int i = 0;

    do {
        if (str[i]>='0' && str[i]<='9')
            *pdigits++ = str[i];
        if (str[i++] == '\0')
            break;
    } while (1);
    *pdigits = '\0';
    return p;
}
```

Exercice 3

Soit le programme suivant :

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

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

L'exécution du programme affichera :

Solution :

ababab

Exercice 4

Soit le programme suivant :

```
#include <stdio.h>

int a = 1;

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

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

L'exécution du programme affichera :

Solution :

2 1 2 0 1

Exercice 5

On souhaite afficher tous les nombres premiers strictement inférieurs à une limite indiquée par l'utilisateur. Pour ce faire, on utilise le crible d'Ératosthène :

- un tableau `char T[]` de taille égale à la limite est créé puis tous ses éléments sont initialisés à 0, sauf les deux premiers qui sont mis à 1 (car ni 0, ni 1, ne sont considérés comme premiers). Ce tableau va progressivement être rempli pour indiquer les nombres premiers (`T[i]==0`) ou multiples d'un autre (`T[i]==1`);
- pour tous les nombres `i` variant de 2 à la racine carrée de la limite : si `T[i]` est nul (`i` est premier), on « élimine » du tableau tous les multiples de `i` en effectuant une seconde boucle démarrant à `j=2*i` et en mettant l'élément correspondant du tableau à 1;
- une fois la boucle générale sur `i` terminée, les indices pour lesquels `T[i]` est toujours nul correspondent aux nombres premiers.

1. Écrire une fonction

```
void eliminer(char *T, int nombre, int limite);
```

qui met à 1 les éléments du tableau `T` correspondant aux multiples de `nombre` (à partir de `2 * nombre`) jusqu'à la limite.

Solution :

Listing 3. Éliminer dans le tableau

```
void eliminer(char *T, int nombre, int limite) {  
    int j;  
    for (j = 2*nombre; j < limite; j += nombre) {  
        T[j] = 1;  
    }  
}
```

2. Écrire une fonction

```
void crible(char *T, int limite);
```

qui parcourt le tableau en commençant à l'indice 2 et jusqu'à la racine carrée de limite, et élimine (en appelant la fonction précédente) à chaque fois les multiples de i si i est premier.

Solution :

Listing 4. Crible

```
void crible(char *T, int limite) {  
    int i;  
    float rac = sqrt(limite);  
    for (i = 2; i < rac; i++)  
        if (!T[i])  
            eliminer(T, i, limite);  
}
```

3. Écrire la fonction principale

- qui demande la limite à l'utilisateur;
- crée dynamiquement le tableau et l'initialise;
- élimine tous les multiples du tableau à l'aide des fonctions précédentes;
- et affiche tous les nombres premiers inférieurs à la limite ainsi que leur nombre total.

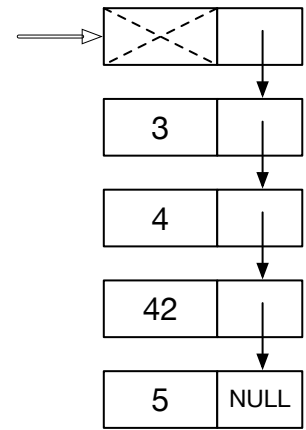
Solution :

Listing 5. Fonction principale

```
int main() {  
    int limite, total, n;  
  
    printf("Limite ?");  
    scanf("%d", &limite);  
    char *T = malloc(sizeof(char)*limite);  
    /* On suppose limite > 1 pour simplifier l'écriture */  
    T[0] = T[1] = 1;  
    for (n = 2; n < limite; n++)  
        T[n] = 0;  
    crible(T, limite);  
    total = 0;  
    for (n = 2; n < limite; n++) {  
        if (!T[n]) {  
            printf("%d ", n);  
            total++;  
        }  
    }  
    printf("\n%d nombres premiers inférieurs a %d\n", total,  
        limite);  
    return 0;  
}
```

Exercice 6

On souhaite gérer une liste d'entiers (int) à l'aide d'une structure de pile. Cette pile sera implémentée sous la forme d'une liste simplement chaînée. On accédera à la pile à l'aide d'un pointeur sur la première cellule vide à son sommet (voir figure ci-contre).



1. Définir la structure d'un élément de la pile `struct cell` puis un `typedef Cell` correspondant.

Solution :

Listing 6. Structure

```
struct cell {
    int val;
    struct cell *next;
};
typedef struct cell Cell;
```

2. Écrire une fonction

`Cell *creerPile(void);`

qui crée une pile vide et renvoie un pointeur sur la première cellule vide au sommet.

Solution :

Listing 7. creerPile

```
Cell *creerPile(void) {
    Cell *p = malloc(sizeof(Cell));
    p->val = 0; /* Mais ce n'est pas nécessaire */
    p->next = NULL;
    return p;
}
```


3. Écrire une fonction

```
int estVide(Cell *sommet);
```

qui teste si la pile dont on indique le sommet est vide (signifiant qu'il n'y a pas d'élément empilé); la fonction renvoie 1 dans ce cas et 0 sinon.

Solution :

Listing 8. estVide

```
int estVide(Cell *sommet) {  
    if ((sommet == NULL) || (sommet->next == NULL))  
        return 1;  
    else  
        return 0;  
}
```

4. Écrire une fonction

```
void push(Cell **p, int nbr);
```

qui empile l'entier indiqué en modifiant le pointeur désignant le sommet de la pile (premier paramètre).

Solution :

Listing 9. push

```
void push(Cell **p, int nbr) {  
    if ((p == NULL) || (*p == NULL))  
        return;  
    (*p)->val = nbr;  
    Cell *nouv = malloc(sizeof(Cell));  
    nouv->val = 0;  
    nouv->next = *p;  
    *p = nouv;  
}
```

On peut aussi simplement insérer derrière la cellule vide.

Listing 10. push

```
void push(Cell **p, int nbr) {  
    if ((p == NULL) || (*p == NULL))  
        return;  
    Cell *nouv = malloc(sizeof(Cell));  
    nouv->val = nbr;  
    nouv->next = (*p)->next;  
    (*p)->next = nouv;  
}
```

5. Écrire une fonction

```
int pop(Cell **p);
```

qui dépile l'entier se trouvant au sommet et le renvoie, et modifie le pointeur indiquant le sommet de la pile (paramètre). Si la pile est vide, la fonction renvoie -1 .

Solution :

Listing 11. pop

```
int pop(Cell **p) {  
    Cell* last;  
    int val;  
    if ((p == NULL) || (*p == NULL))  
        return -1;  
    last = (*p)->next;  
    if (last == NULL)  
        return -1;  
    val = last->val ;  
    free(*p);  
    *p = last;  
    return val;  
}
```