

Examen de Info C

Durée : 1h30

Aucun document autorisé

Mobiles interdits

Notez que les affirmations (antécédents, conséquents, rôles, et invariants) dans vos codes C entreront pour partie dans la note finale.

- 1. Expliquez de façon claire comment sont organisées les données d'un programme au cours de son exécution dans la mémoire centrale.

cf. cours

- 2. Expliquez de façon claire comment un programme C peut accéder aux *paramètres programmes* fournis au moment du lancement de son exécution.

cf. cours

- 3. Écrivez ce qu'affichent les 10 appels de la procédure `trace` dans la programme suivant :

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

void trace(const int temp, const int block[]) {
    printf("temp = %d", temp);
    printf(" - bloc = [%d, %d, %d]\n", block[0], block[1], block[2]);
}

int main(void)
{
    int block[3] = {0, 3, 5};
    int *ptr = &block[1];
    int temp;

    temp = block[1];    trace(temp, block);
    temp = *(block + 1); trace(temp, block);
    temp = *(ptr - 1);   trace(temp, block);
    temp = *ptr;         trace(temp, block);

    ptr = block+1;      trace(temp, block);
    temp = *ptr;         trace(temp, block);
    temp = *(ptr+1);     trace(temp, block);

    ptr = block;
```

```
temp = *ptr++;    trace(temp, block);
temp = ++*ptr;    trace(temp, block);
temp = *ptr--;    trace(temp, block);

return EXIT_SUCCESS;
}
```

```
temp = 3 - bloc = [0, 3, 5]
temp = 3 - bloc = [0, 3, 5]
temp = 0 - bloc = [0, 3, 5]
temp = 3 - bloc = [0, 3, 5]
temp = 3 - bloc = [0, 3, 5]
temp = 3 - bloc = [0, 3, 5]
temp = 5 - bloc = [0, 3, 5]
temp = 0 - bloc = [0, 3, 5]
temp = 4 - bloc = [0, 4, 5]
temp = 4 - bloc = [0, 4, 5]
```

- 4. Écrivez en C la fonction booléenne `estPrefixe` qui teste si une chaîne de caractères `s1` est préfixe d'une chaîne de caractères `s2`. Les deux chaînes sont paramètres de la fonction. Par exemple, "abri" est préfixe de "abricot". Deux chaînes identiques ne sont pas préfixes l'une de l'autre. Vous utiliserez **exclusivement** la notation de pointeur, et **aucune** fonction de `string.h`.

```
/*
 * Rôle : teste si la chaîne s1 et préfixe de la chaîne s2
 */
int estPrefixe(const char *s1, const char *s2) {
    if (*s1 == '\0') return 0;
    while (*s1==*s2) {
        if (*s1 == '\0')
            // s1 et s2 sont égales
            return 0;
        // avancer dans les 2 chaînes
        s1++;
        s2++;
    }
    // si fin de chaîne de s1, s1 est préfixe
    return *s1=='\0';
}
```

- 5. Soit la déclaration suivante du type `Liste` vu en TD.

```
typedef int T;

typedef struct noeud {
    T elt;
    struct noeud *suivant;
```

```
} *Liste;
```

Écrivez à l'aide de la procédure du TD `supprimer`, la procédure `supprimerEnQueue` qui supprime le dernier élément d'une liste. Son en-tête est le suivant :

```
/*
 * Rôle : supprime le dernier élément de la liste
 */
void supprimerEnQueue(Liste *l)

.....

/*
 * Rôle : supprime le dernier élément de la liste
 */
void supprimerEnQueue(Liste *l) {
    supprimer(l, longueur(*l)); // idem que supprimer(0(*l), longueur(*l))
}
```

- 6. Sans utiliser la procédure du TD `supprimer`, récrivez la procédure `supprimerEnQueue` précédente.

```
/*
 * Rôle : supprime le dernier élément de la liste
 */
void supprimerEnQueue(Liste *l) {
    const int lg = longueur(*l);
    assert(lg>0);
    // se placer sur l'avant dernier noeud
    struct noeud *p;
    if (lg==1) {
        // cas particulier, suppression du 1er et seul élément
        p = *l;
        *l = NULL;
    } else {
        // cas général : se placer sur l'avant dernier noeud
        struct noeud *q = *l;
        for (int i=2; i<lg; i++)
            q = q->suivant;
        // q désigne l'avant dernier noeud
        p = q->suivant; // l'élément à supprimer
        q->suivant = NULL;
    }
    // libérer la place occupée par l'élément à supprimer
    free(p);
}
```

- 7. À l'aide de la fonction `ieme` (vue en TD), écrivez la procédure `ecrireListe` qui prend en paramètre une liste de type `Liste` et qui écrit tous ses éléments sur la sortie standard. On considérera que la liste contient des entiers (type `int`).

```
void ecrireListe(const Liste l) {
    const int lg = longueur(l);
    for (int i=1; i<=lg; i++)
        printf("%d ", ieme(l, i));
    printf("\n");
}
```

- 8. Que pensez-vous de l'efficacité de cette procédure ? Expliquez.

Au premier abord, cette procédure semble parfaite, mais en regardant de plus près, elle nécessite **deux** parcours de la liste. Le second étant fait à chaque itération par la fonction `ieme`. Pour une liste de longueur n , cette fonction a une complexité $\mathcal{O}(n^2)$. Pire encore, comme la fonction `longueur` fait également un parcours, la complexité est en fait, $\mathcal{O}(n^3)$, car `longueur` est appelée une fois dans `ecrireListe`, mais aussi à chaque appel de `assert` dans `ieme`. Cette procédure n'est donc pas du tout efficace.

Pour une liste linéaire, son parcours doit aussi être linéaire, et donc de complexité $\mathcal{O}(n)$. Nous verrons l'an prochain, la notion d'*itérateur* qui règle ce problème.