TD 7 : Révisions

Programmation en C (LC4)

Semaine du 12 mars 2007

1 Divers

▶ **Exercice 1**

```
20097261
```

▶ **Exercice 2**

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

void majuscule(char *mot) {
    while (*mot != '\0') {
        *mot += 'A' - 'a';
        mot++;
    }
}

void majuscule2(char *mot) {
    while (*mot != '\0') {
        *mot = toupper(*mot);
        mot++;
    }
}

int main(int argc, char *argv[]) {
    majuscule(argv[1]);
    printf("%s\n", argv[1]);
    return 0;
}
```

▶ **Exercice 3**

En C, la conversion d'une valeur de type **double** (ou **float**) vers une valeur de type **int** se fait en arrondissant vers 0 (en tronquant les chiffres après la virgule), la fonction affiche donc : `6, 5`.

▶ **Exercice 4**

```
#include <stdio.h>

int pythagore(int a, int b, int c) {
    return (a * a + b * b == c * c);
}

int main(int argc, char *argv[]) {
    int a, b, c, n;
    int cpt;
    printf("nombre de triplets : ");
    scanf("%d", &n);
```

```
    cpt = 0;
    for (c = 3; cpt < n; c++)
        for (b = 2; b < c && cpt < n; b++)
            for (a = 1; a < b && cpt < n; a++)
                if (pythagore(a, b, c)) {
                    cpt++; /* nouveau triplet */
                    printf("%d: %d^2 + %d^2 = %d^2\n", cpt, a, b, c);
                }
    return 0;
}
```

2 Tableaux

► **Exercice 5**

```
/* double maximum(int taille, double tableau[]) */
double maximum(int taille, double *tableau) {
    int i;
    double m = tableau[0];
    for (i = 0; i < taille; i++)
        if (tableau[i] > m)
            m = tableau[i];
    return m;
}

/* void echange(int i, int j, double tableau[]) */
void echange(int i, int j, double *tableau) {
    double t = tableau[i];
    tableau[i] = tableau[j];
    tableau[j] = t;
}

/* int verifie(int taille, double tableau[]) */
int verifie(int taille, double *tableau) {
    int i;
    for (i = 0; i < taille - 1; i++)
        if (tableau[i] > tableau[i + 1])
            return 0;
    return 1;
}

/* void renverse(int taille, double tableau[]) */
void renverse(int taille, double *tableau) {
    int i;
    for (i = 0; i < taille / 2; i++)
        echange(i, taille - 1 - i, tableau);
}

/* double *concat(int taille1, double tableau1[],
                  int taille2, double tableau2[]) */
double *concat(int taille1, double *tableau1, int taille2, double *tableau2) {
    double *resultat = malloc((taille1 + taille2) * sizeof(double));
    int i;
    for (i = 0; i < taille1; i++)
        resultat[i] = tableau1[i];
    for (i = 0; i < taille2; i++)
        resultat[i + taille1]= tableau2[i];
```

```
    return resultat;
}
```

3 Pointeurs

► **Exercice 6**

`i == 17` et `j == 22`.

► **Exercice 7**

- &tab[0] == &s1
- ∗tab == s1 ("Un")
- tab[0] == s1 ("Un")
- tab[1] == s2 ("Deux")
- ∗(tab + 1) == s2 ("Deux")
- ∗∗tab == 'U'
- ∗tab[0] == 'U'
- ∗∗(tab + 1) == 'D'
- ∗tab[1] == 'D'

► **Exercice 8**

```
#include <stdio.h>

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

int main_permute(void) {
    int i = 5, j = 2;
    printf("%d %d\n", i, j);
    permute(&i, &j);
    printf("%d %d\n", i, j);
    return 0;
}
```

4 Chaînes de caractères

► **Exercice 9**

```
char *lit_chaine(void) {
    int taille = 0, capacite = 8;
    int c;
    char *s = malloc(capacite);

    while ((c = getchar()) != '\n') {
        if (taille == capacite - 1) {
            capacite *= 2;
            s = realloc(s, capacite);
        }
        s[taille] = c;
        taille++;
    }
    s[taille] = '\0';
    return s;
```

```
}

char *inverse_chaine(char *s) {
    char *t;
    int i, n = strlen(s);
    t = malloc(n + 1);
    for (i = 0; i < n; i++)
        t[i] = s[n - 1 - i];
    t[n]= '\0';
    return t;
}

int est_palindrome(char *s) {
    char *t, *u = inverse_chaine(s);
    t = u;
    while (*s++ == *u++) {
    }
    free(t);
    return (*s == '\0');
}

int est_palindrome2(char *s) {
    int cmp;
    char *t = inverse_chaine(s);
    cmp = strcmp(s, t);
    free(t);
    return !cmp;
}
```

5 Fonctions récursives

► **Exercice 10**

```
int coefficient_binomial(int n, int p)
{
    if (p == 0 || p == n)
        return 1;
    else
        return coefficient_binomial(n - 1, p)
            + coefficient_binomial(n - 1, p - 1);
}
```

6 Listes chaînées

► **Exercice 11**

Avec l'exemple donné et en utilisant un simple tableau de coefficients, le polynome aurait besoin d'un espace mémoire très conséquent de 1000000 **double** alors qu'en utilisant uniquement ses monômes utiles, le polynome n'aurait besoin que d'une liste à 5 éléments.

```
typedef struct monome {
    int degre;
    double coefficient;
    struct monome *suivant;
} *monome;

typedef monome polynome;
```

```
void affiche_polynome(polynome p) {
    while (p != NULL) {
        if (p->degre > 1) {
            printf("%gX^%d", p->coefficient, p->degre);
        } else {
            if (p->degre == 1)
                printf("%gX", p->coefficient);
            else
                printf("%g", p->coefficient);
        }
        p = p->suivant;
        if (p != NULL)
            printf("+");
    }
    printf("\n");
}

monome cree_monome(int degre, double coefficient) {
    monome m = malloc(sizeof(struct monome));
    m->degre = degre;
    m->coefficient = coefficient;
    m->suivant = NULL;
    return m;
}

void detruit_polynome(polynome p) {
    monome m;
    while (p != NULL) {
        m = p->suivant;
        free(p);
        p = m;
    }
}

polynome convertit_polynome(int degre, double *coefficients) {
    int i;
    polynome p = NULL, *q = &p;
    monome m;
    for (i = degre; i >= 0; i--)
        if (coefficients[i] != 0.0) {
            m = cree_monome(i, coefficients[i]);
            *q = m;
            q = &m->suivant;
        }
    return p;
}

polynome derive_polynome(polynome p)
{
    polynome q = NULL, *r = &q;
    monome m;
    while (p != NULL && p->degre != 0) {
        m = cree_monome(p->degre - 1, p->degre * p->coefficient);
        *r = m;
        r = &m->suivant;
        p = p->suivant;
```

```
    }
    return q;
}

polynome ajoute_polynome(polynome p, polynome q)
{
    double coefficient;
    polynome r = NULL, *s = &r, t;
    monome m;
    while (p != NULL && q != NULL) {
        if (p->degre < q->degre) {
            t = p; p = q; q = t;
        }
        coefficient = p->coefficient;
        if (p->degre == q->degre) {
            coefficient += q->coefficient;
            q = q->suivant;
        }
        m = cree_monome(p->degre, coefficient);
        *s = m;
        s = &m->suivant;
        p = p->suivant;
    }
    if (q != NULL) /* seul un des pointeurs p ou q peut être non NULL */
        p = q;
    while (p != NULL) {
        m = cree_monome(p->degre, p->coefficient);
        *s = m;
        s = &m->suivant;
        p = p->suivant;
    }
    return r;
}
```