**IN 101 - TD 09**
18 novembre 2011

# Corrigé

ENSTA ParisTech

**Solution 1 :** Pile et calculatrice

**1.a-d ]** Voici le code pour les 4 premières questions :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef struct cell_st {
5    int val;
6    struct cell_st* next;
7  } cell;
8
9  cell* P = NULL;
10
11 void push(int a) {
12   cell* new = (cell*) malloc(sizeof(cell));
13   new->val = a;
14   new->next = P;
15   P = new;
16 }
17
18 int pop() {
19   int res;
20   cell* tmp = P;
21   if (P == NULL) {
22     printf("Pile vide!!!\n");
23     return -1;
24   }
25   res = P->val;
26   P = P->next;
27   free(tmp);
28   return res;
29 }
30
31 int main(int argc, char* argv[]) {
32   int i,a;
33   for (i=1; i<argc; i++) {
34     if (argv[i][0] == '+') {
35       push(pop() + pop());
36     } else if (argv[i][0] == '-') {
37       a = pop();        // attention à l'ordre des arguments
38       push(pop() - a);
39     } else if (argv[i][0] == 'x') {
40       push(pop() * pop());
41     } else if (argv[i][0] == '/') {
42       a = pop();
```

```
43        push(pop() / a);
44      } else {
45        push(atoi(argv[i]));
46      }
47    }
48    printf("%d\n", pop());
49    return 0;
50  }
```

**1.e ]** Voici la fonction d'affichage et le `main` pour la version interactive de la calculatrice.

```
1  // print_stack retourne aussi le nombre d'éléments dans la pile
2  int print_stack() {
3    cell* cur = P;
4    int c = 0;
5    if (cur == NULL) {
6      printf("Pile vide.\n");
7      return 0;
8    }
9    while (cur != NULL) {            // parcours de liste chaînée
10     printf("%d  ", cur->val);
11     cur = cur->next;
12     c++;
13   }
14   printf("\n");
15   return c;
16 }
17
18 int main(int argc, char* argv[]) {
19   int c,a;
20   char input[256];
21   while (1) {  // on continue indéfiniment, le . sert à quitter
22     printf("\n");
23     c = print_stack();
24     printf("> ");
25     scanf("%s", input);
26
27     if (input[0] == '.') {          // pour quitter la calculatrice
28       break;                        // break sort de la boucle while
29     } else if (input[0] == 'c') {   // pour retirer un élément
30       if (c < 1) {
31         printf("Pas assez d'éléments dans la pile.\n");
32       } else {
33         pop();
34       }
35     } else if (input[0] == '+') {
36       if (c < 2) {         // on vérifie que la pile est assez pleine
37         printf("Pas assez d'éléments dans la pile.\n");
38       } else {
39         push(pop() + pop());
40       }
41     } else if (input[0] == '-') {
42       if (c < 2) {
43         printf("Pas assez d'éléments dans la pile.\n");
```

```
44        } else {
45          a = pop();
46          push(pop() - a);
47        }
48      } else if (input[0] == '*') { // on peut utiliser * ici
49        if (c < 2) {
50          printf("Pas assez d'éléments dans la pile.\n");
51        } else {
52          push(pop() * pop());
53        }
54      } else if (input[0] == '/') {
55        if (c < 2) {
56          printf("Pas assez d'éléments dans la pile.\n");
57        } else {
58          a = pop();
59          push(pop() / a);
60        }
61      } else {
62        push(atoi(input));
63      }
64    }
65    return 0;
66  }
```

**Solution 2 :** Conversion en notation polonaise inverse

**2.a ]** Pour cette question, on doit stocker des *char* dans la pile : les symboles des opérateurs. Un *char* est un entier sur 8 bits en C, donc on peut encore utiliser la même structure de pile qu'avant avec des *int*. Il suffit donc de modifier le `main` pour lire une expression infixée.

```
1  int main(int argc, char* argv[]) {
2    int i;
3    for (i=1; i<argc; i++) {
4      if (argv[i][0] == ']') {
5        printf("%c ", pop());    // parenthèse fermante : on dépile un opérateur
6      } else if (argv[i][0] == '[') {
7        // on ne fait rien des parenthèses ouvrantes
8      } else if ((argv[i][0] == '-') ||    // le || est un ou logique
9                 (argv[i][0] == '+') ||
10                (argv[i][0] == '/') ||
11                (argv[i][0] == 'x')) {
12       push(argv[i][0]);        // tous les symboles vont dans la pile
13     } else {
14       printf("%s ", argv[i]);  // on affiche les chiffre directement
15     }
16   }
17   if (P != NULL) {
18     printf("%c ", pop());
19   }
20   printf("\n");
21   return 0;
22 }
```

**2.b** ] Il faut modifier un peu la structure de pile pour que la liste chaînée contienne des pointeurs vers des chaînes de caractères au lieu d'entiers. Ensuite, la fonction `concat` permet de concaténer des chaînes de caractères en mettant des parenthèses uniquement si nécessaire (si l'un des deux arguments contient déjà un opérateur). Notez que la fonction `strcat` de la bibliothèque standard `string.h` pourrait aussi servir ici, mais on veut une concaténation un peu spécifique.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  typedef struct cell_st {
6    char* val;
7    struct cell_st* next;
8  } cell;
9
10 char* copy(char* input) {
11   char* res;
12   int i;
13   int l = strlen(input);
14   res = (char*) malloc((l+1)*sizeof(char));
15   for (i=0; i<l+1; i++) {
16     res[i] = input[i];
17   }
18   return res;
19 }
20
21 char* concat(char* a, char* b, char symb) {
22   char* res;
23   int i, j, par_a, par_b;
24   int l = 0;          // la longueur du resultat
25   // on teste si a et b doivent être entourés de parenthèses
26   i = 0;
27   par_a = 0;
28   while (a[i] != 0) {
29     if (a[i] < '0') { // les opérateurs sont avant les chiffres en ASCII
30       par_a = 1;
31     }
32     i++;  l++;        // on incrémente la longueure totale
33   }
34   if (par_a != 0) {
35     l += 2;           // la place pour les parenthèses
36   }
37   i = 0;
38   par_b = 0;
39   while (b[i] != 0) {
40     if (b[i] < '0') { par_b = 1; }
41     i++;  l++;
42   }
43   if (par_b != 0) { l+=2; }
44   // on reserve la mémoire pour le resultat
45   res = (char*) malloc((l+2)*sizeof(char));
46
47   // maintenant, on colle tous les morceaux bout à bout
48   j=0;   // j est le nombre de char écrits dans res
49   if (par_a) { res[j]='(';  j++; }
```

```
50   i = 0;
51   while (a[i] != 0) {
52     res[j] = a[i];
53     j++;  i++;
54   }
55   if (par_a) { res[j]=')';  j++; }
56   res[j]=symb;  j++;    // le symbole
57   if (par_b) { res[j]='(';  j++; }
58   i = 0;
59   while (b[i] != 0) {
60     res[j] = b[i];
61     j++;  i++;
62   }
63   if (par_b) { res[j]=')';  j++; }
64   res[j] = 0;   // caractère de fin de chaîne
65   return res;
66 }
67
68 int main(int argc, char* argv[]) {
69   int l;
70   char *a, *b, *c;
71   char input[256];
72   while (1) {
73     printf("\n");
74     l = print_stack();
75     printf("> ");
76     scanf("%s", input);
77
78     if (input[0] == '.') {  // pour quitter la calculatrice
79       break;
80     } else if (input[0] == 'c') { // pour retirer un élément
81       if (l < 1) {
82         printf("Pas assez d'éléments dans la pile.\n");
83       } else {
84         a = pop();
85         free(a);            // il faut penser au free après un pop
86       }
87     } else if ((input[0] == '+') || (input[0] == '-') ||
88                (input[0] == '*') || (input[0] == '/')) {
89       if (l < 2) {
90         printf("Pas assez d'éléments dans la pile.\n");
91       } else {
92         a = pop();  b = pop();
93         c = concat(a,b,input[0]);
94         free(a);  free(b);   // on libère a et b qui ne servent plus
95         push(c);
96       }
97     } else {
98       c = copy(input);
99       push(c);
100    }
101  }
102  return 0;
103 }
```

## **Solution 3 :** Polynômes creux

**3.a** ] On utilise encore une structure de liste chaînée, mais chaque cellule contient deux *int*. L'affichage peut se faire de manière récursive, mais pour ne pas afficher de + de trop, une boucle est plus simple. Notez qu'on peut utiliser une boucle `for` aussi facilement qu'une boucle `while` pour cela.

```
1  typedef struct cell_st {
2    int coef;
3    int deg;
4    struct cell_st* next;
5  } cell;
6
7  void print_poly(cell* P) {
8    cell* cur;
9    int first = 1;
10   for (cur=P; cur!=NULL; cur=cur->next) {
11     if (first) {
12       first = 0;
13     } else {
14       printf(" + ");
15     }
16     // on n'affiche la puissance/le coefficient que si nécessaire
17     if (cur->deg > 1) {
18       if (cur->coef == 1) {
19         printf("x^%d", cur->deg);
20       } else {
21         printf("%dx^%d", cur->coef, cur->deg);
22       }
23     } else if (cur->deg == 1) {
24       if (cur->coef == 1) {
25         printf("x");
26       } else {
27         printf("%dx", cur->coef);
28       }
29     } else {
30       printf("%d", cur->coef);
31     }
32   }
33   printf("\n");
34 }
```

**3.b** ] Pour modifier le polynôme passé en argument, on le passe par adresse. Le type de l'argument est donc *cell***, et il faut utiliser `(*P)`.

```
1  void add_monom(cell** P, int c, int d) {
2    cell* new = (cell*) malloc(sizeof(cell));
3    new->coef = c;
4    new->deg = d;
5    new->next = (*P);
6    *P = new;
7  }
```

**3.c** ] En utilisant la fonction `add_monom` précédente, cette fonction est très simple.

```
1 void add_poly(cell** P1, cell* P2) {
2   cell* cur = P2;
3   while (cur != NULL) {
4     add_monom(P1, cur->coef, cur->deg);
5     cur = cur->next;
6   }
7 }
```

**3.d** ] Voici les fonctions nécessaires.

```
1 void free_poly(cell* P) {
2   if (P != NULL) {
3     free_poly(P->next);
4     free(P);
5   }
6 }
7 cell* clone_poly(cell* P) {
8   cell* res = NULL;
9   add_poly(&res, P);
10  return res;
11 }
12 void mul_monom(cell* P, int c, int d) {
13  cell* cur;
14  for (cur=P; cur!=NULL; cur=cur->next) {
15    cur->coef *= c;
16    cur->deg += d;
17  }
18 }
19 cell* mul_poly(cell* P1, cell* P2) {
20  cell *tmp, *res = NULL;
21  cell* cur = P1;
22  while (cur != NULL) {
23    tmp = clone_poly(P2);   // on multiplie P2 par chaque monôme de P1
24    mul_monom(tmp, cur->coef, cur->deg);
25    add_poly(&res, tmp);    // on ajoute le tout dans res
26    free_poly(tmp);         // libére le polynôme tmp qui ne sert plus
27    cur = cur->next;
28  }
29  return res;
30 }
```

**3.e** ] La fonction de réduction utilise ici un tri à bulle (qui est plus compliqué sur une liste chaînée que sur un tableau). Si le polynôme est déjà réduite le coût de la réduction est linéaire.

```
1 void reduce_poly(cell** P) {
2   int sorted = 0;
3   cell **cur, *tmp;
4   if (((*P) == NULL) || ((*P)->next == NULL)) {
5     return;
6   }
```

```
  // tri à bulles sur une liste chaînée
  while (!sorted) {  // continue tant que tout n'est pas trié
    cur = P;
    sorted = 1;
    while (((*cur) != NULL) && ((*cur)->next != NULL)) {
      if ((*cur)->deg < (*cur)->next->deg) {
        tmp = (*cur)->next;
        (*cur)->next = (*cur)->next->next;
        tmp->next = *cur;
        *cur = tmp;
        sorted = 0;
      } else if ((*cur)->deg == (*cur)->next->deg) {
        (*cur)->coef += (*cur)->next->coef;
        tmp = (*cur)->next;
        (*cur)->next = (*cur)->next->next;
        free(tmp);
      }
      cur = &((*cur)->next);
    }
  }
  cur = P;              // on va retirer les coefficients nuls
  while ((*cur) != NULL) {
    if ((*cur)->coef == 0) {
      tmp = *cur;
      *cur = (*cur)->next;
      free(tmp);
    } else {  // on n'avance que si la case n'a pas été supprimée
      cur = &((*cur)->next);
    }
  }
}
```

**3.f ]** La fonction de calcul de pgcd de deux polynômes : à chaque étape on annule le monôme de tête du polynôme de plus haut degré. Attention, par rapport aux entiers il ne faut pas systématiquement inverser les deux polynômes.

```
cell* gcd_poly(cell* P1, cell* P2) {
  cell *r1, *r2, *tmp;
  r1 = clone_poly(P1);  reduce_poly(&r1);
  r2 = clone_poly(P2);  reduce_poly(&r2);
  while (r2 != NULL) {
    tmp = clone_poly(r2);
    mul_monom(tmp, -r1->coef, r1->deg - r2->deg);
    mul_monom(r1, r2->coef, 0);
    add_poly(&r1, tmp);
    free_poly(tmp);
    reduce_poly(&r1);
    if ((r1 == NULL) || (r1->deg < r2->deg)) {
      tmp = r1;  r1 = r2;  r2 = tmp;
    }
  }
  return r1;
}
```