Structures de données : les bases

Un premier programme C sous LINUX

Dans un terminal LINUX, créez un répertoire de travail (`3i_in9`, par exemple) et placez-y vous.

```
acme1:~> mkdir 3i_in9 ↩
acme1:~> cd 3i_in9 ↩
acme1:~/3i_in9>
```

Avec un éditeur de texte (par exemple, nedit),

```
acme1:~/3i_in9> nedit & ↩
```

saisissez le programme source C suivant :

```
1  /*
2  pgm01.c Un premier exemple de programme C
3  */
4  #include<stdio.h>
5  int main()  {
6    int n;
7    printf("Entrez un nombre entier : ");
8    scanf("%d", &n);
9    printf("%d x %d = %d\n", n, n, n*n);
10   return 0;
11 }
```

Sauvegardez-le sous le nom `pgm01.c`, puis compilez-le :

```
acme1:~/3i_in9> gcc pgm01.c ↩
```

Si la compilation se passe bien, un fichier exécutable de nom `a.out` est créé et vous pouvez l'exécuter en lançant la commande :

```
acme1:~/3i_in9> ./a.out ↩
```

Un message apparaît, correspondant à la ligne 7 du listing.

```
Entrez un nombre entier :
```

Le programme s'interrompt alors, attendant l'intervention de l'utilisateur pour la saisie du nombre (ligne 8 du listing).

Si on entre 12, par exemple, le programme affiche le résultat (ligne 9 du listing), puis se termine :

```
Entrez un nombre entier : 12 ↩
12 x 12 = 144
```

Exemple d'écriture d'une fonction

Écrivez deux fonctions calculant la factorielle de n, la première à l'aide d'une boucle `for` et la deuxième en utilisant la récursivité.

Exemple de solution :

```
/* int factorielle_iterative(int n)
 * calcule la factorielle de n
 * Principe : accumulation des produits des n premiers nombres
 * arguments : n entier positif
 * retour : factorielle de n si n>=0
                    1 si n est negatif
 * JCG ecrit le 22/03/2001 modif le 14/04/2013
 */
int factorielle_iterative(int n) {
  int i, p = 1;
  for (i = 1; i <= n; ++i) p *= i;
  return p;
}
/*****************************************************************/
/* int factorielle_recursive(int n)
 * calcule la factorielle de n
 * Principe : F(n)=1 si n=0, F(n)=n*F(n-1) si n>0
 * arguments : n entier positif
 * retour : factorielle de n si n>=0
                    1 si n est negatif
 * JCG ecrit le 22/03/2001 modif le 14/04/2013
*/
int factorielle_recursive(int n) {
  if (n <= 0)
    return 1;
  else
    return n * factorielle_recursive(n-1);
}
/*****************************************************************/
#include <stdio.h>
int main(void) {
  int n;
  do {
    printf("Entrez un nombre entier (0 pour terminer) : ");
    scanf("%d", &n);
    printf("factorielle_iterative(%2d) = %15d\n", n, factorielle_iterative(n));
    printf("factorielle_recursive(%2d) = %15d\n", n, factorielle_recursive(n));
  } while (n != 0);
  return 0;
}
```

Compilez et exécutez ce programme. Entrez les valeurs 1, 2, 5, 10, 13. Comment peut-on voir que le résultat pour 13 est faux ? Pourquoi est-il faux ?

Comment modifier les fonctions pour que le nombre retourné soit un nombre impossible (-1 par exemple) lorsque le résultat ne peut pas être calculé ?

1 Les types du langage C

1.1 Types de base

Exercice 1. Détermination de type

Choisissez parmi les types de base (`char`, `int`, `double`, `char[]`) le type vous semblant le plus adapté pour représenter les données suivantes :

- une année de naissance ;
- un numéro de téléphone ;
- un numéro de département français ;
- un taux de TVA ;
- une taille de fichier ;
- un numéro de chambre d'hôtel.

Exercice 2. Proximité de deux `double`

Avec les définitions suivantes,

```
#define ZERO 1e-100
#define EPSILON 1e-10
```

écrivez la fonction

```
int proche(double a, double b);
```

qui renvoie un entier non nul si `a` et `b` sont proches l'un de l'autre (tous deux inférieurs à `ZERO` en valeur absolue, ou égaux à `EPSILON` près), et 0 sinon.

Exercice 3. Visualisation de `double`

Nous voulons "visualiser" le codage hexadecimal d'un `double` (voir le site : http://babbage.cs.qc.cuny.edu/IEEE-754/index.xhtml).
Sachant que un `double` est stocker en mémoire sous la forme de 8 octets, proposez une méthode pour convertir une variable de type `double` en un tableau de `char[8]`.
Astuces : pensez à la conversion (cast) de pointeur ou à l'utilisation du type `union` (voir : https://gcc.gnu.org/onlinedocs/gcc/Unnamed-Fields.html)
Écrivez la fonction

```
void print_hexa(double x);
```

qui affiche la valeur (avec la précision maximum) et le codage hexadécimal du double `x`.
Testez votre fonction, affichiez la valeur d'un double avant et après lui avoir ajouter une très petite valeur (par exemple :10^{-100}). Visualisez le codage utilisé pour les valeurs `inf` et `nan`.

1.2 Les pointeurs

Un pointeur est une variable destinée à contenir l'adresse d'une zone mémoire typée. Le rôle principal d'un pointeur est de permettre d'accéder à une zone mémoire de façon indirecte, pour pouvoir la modifier.

Exercice 4. Modification d'une variable par pointeur

Écrivez la fonction suivante

```
void eleve_au_carre(int* p) {
/* modifie en l'elevant au carre l'entier dont l'adresse est dans p */
// a completer
}
```

Testez-la avec :

```
int main() {
    int n = 5;
    eleve_au_carre(&n);
    printf("%d\n", n); /* => 25 */
    return 0;
}
```

1.3 Les tableaux

En C, l'accès à un élément de tableau n'est pas sécurisé. Pour éviter de *sortir* d'un tableau, on utilise principalement deux solutions :

- on fournit à la fonction travaillant sur le tableau la longueur exploitable du tableau : tout accès peut alors être testé ;
- on marque la *fin* du tableau par une sentinelle (un élément impossible) : utilisable principalement pour une exploitation séquentielle.

Exercice 5. Parcours de tableau

Complétez les fonctions manquantes du programme suivant. Vérifiez que vos fonctions passent les tests.

```
#include <stdio.h>
/* la reponse a la question precedente */
#define ZERO 1e-100
#define EPSILON 1e-10
/* inclure <math.h> et compiler avec gcc -lm */
int proche(double a, double b) {
    // Votre code de l'exercice 2
}

double moyenne(double t[], int n) {
/* calcule la moyenne des n premiers elements du tableau t */
/* --------------
 * a faire
 * --------------
*/
  return 0.0;
}

double moyenne_positifs(double t[]) {
/* calcule la moyenne des elements du tableau t jusqu'a rencontrer un
element negatif et -1.0 si le premier element est deja negatif */
/* --------------
 * a faire
 * --------------
*/
  return 0.0;
}

double test_moyenne() {
  double v[] = {1.0, 2.0, 3.0, 4.0, 5.0, 6.0, -1.0};
  double d, attendu;
  attendu = 1;
/* test moyenne */
  if (! proche((d = moyenne(v, 1)), attendu)) {
    printf("Pb moyenne. Attendu : %f Obtenu :%f\n", attendu, d);
  }
  attendu = 2.0;
  if (! proche((d = moyenne(v, 3)), attendu)) {
    printf("Pb moyenne. Attendu : %f Obtenu :%f\n", attendu, d);
  }
  attendu = 3.5;
  if (! proche((d = moyenne(v, 6)), attendu)) {
    printf("Pb moyenne. Attendu : %f Obtenu :%f\n", attendu, d);
```

```
  }
/* test moyenne positifs */
  attendu = 3.5;
  if (! proche((d = moyenne_positifs(v)), attendu)) {
    printf("Pb moyenne_positifs. Attendu : %f Obtenu :%f\n", attendu, d);
  }
  attendu = 5.0;
  if (! proche((d = moyenne_positifs(v + 3)), attendu)) {
    printf("Pb moyenne_positifs. Attendu : %f Obtenu :%f\n", attendu, d);
  }
  attendu = -1.0;
  if (! proche(d = moyenne_positifs(v + 6), -1.0)) {
    printf("Pb moyenne_positifs. Attendu : %f Obtenu :%f\n", attendu, d);
  }
}
int main() {
  test_moyenne();
  return 0;
}
```

Exercice 6. Parcours récursifs de tableau

Écrivez la fonction :

```
int chaine_longueur_rec(char* s);
```

qui calcule **récursivement** la longueur d'une chaîne de caractères terminée par `'\0'`.

Écrivez la fonction **récursive** :

```
int chaine_debute_par(char* s1, char* s2);
```

qui retourne un entier non nul si `s1` commence par `s2`, et 0 sinon.

Exercice 7. Recherche de motif

Écrivez la fonction ;

```
int chaine_index(char* s1, char* s2);
```

qui retourne la position de `s2` dans `s1` si `s1` contient `s2`, -1 sinon.

2 Les piles

Une pile est une structure de donnée abstraite permettant de stocker et de récupérer des données selon un schéma LIFO (last in, first out) : seul l'accès au dernier élément de la pile est permis. L'ajout d'un nouvel élément (empiler) se fait au sommet de la pile, la consultation ou la suppression d'un élément n'est possible que sur le dernier élément ajouté.

Les opérations sur pile sont les suivantes :

- **empile** : ajoute dans la pile un élément
- **depile** : supprime de la pile le dernier élément empilé et retourne sa valeur
- **lit_sommet** retourne la valeur du dernier élément empilé (la pile est inchangée)
- **est_vide** : retourne Vrai si la pile est vide, Faux sinon
- **init** : vide la pile

Exercice 8. Implémentation de pile par tableau

Une implémentation de pile simple (pile d'entiers par exemple) peut se faire en utilisant un tableau : la pile est constituée d'un tableau d'entiers, et d'un indice spécial indiquant la position dans le tableau du dernier élément empilé.

Par exemple :

```
#define TMAX 10
typedef struct{
        int donnees[TMAX];
        int sommet;
} t_pile;
```

La **création** d'une pile se fera simplement par la déclaration d'une variable de ce type :

```
t_pile p;
```

L'**initialisation** se fera par la mise à 0 du champ sommet de la structure :

```
p.sommet = 0;
```

ou mieux en écrivant et en utilisant la fonction qui le fera :

```
void init_pile(t_pile* pp) {
        pp->sommet = 0;
}

...

init_pile(&p);
```

L'**utilisation** de la pile créée et initialisée se fera également par l'intermédiaire de fonctions plutôt que par des instructions en ligne :

```
int est_vide_pile(t_pile* pp) {
        return pp->sommet == 0;
}

void empile(t_pile* pp, int x) {
        pp->contenu[pp->sommet] = x;
        ++pp->sommet;
        /* pour les voltigeurs :
        pp->contenu[pp->sommet++] = x;
        */
}

int depile(t_pile* pp) {
        --pp->sommet;
        return pp->contenu[pp->sommet];
        /* pour les voltigeurs :
        return pp->contenu[--pp->sommet];;
        */
}


int lit_sommet_pile(t_pile* pp) {
        return pp->contenu[pp->sommet - 1];
}
```

Écrivez une fonction `main` pour tester la pile ainsi créée.

Que se passe-t-il lorsqu'on empile sur une pile pleine ?

Écrivez une fonction `empile_securit` qui empile une valeur sur la pile si elle n'est pas pleine, mais ne fait rien sur une pile pleine (l'écriture d'une fonction annexe `est_pleine_pile` sera peut-être utile).

Peut-on imaginer une valeur de retour pour cette fonction indiquant si l'empilement a bien pu se faire ?

De même, écrivez une fonction `depile_securit` qui dépile sur une pile non vide, mais ne fait rien sur sur une pile vide.

Peut-on imaginer une valeur de retour pour cette fonction indiquant si le dépilement a bien pu se faire ?