Prog. Or. Système - Correction série 07 : Pointeurs (3) : char* ; pointeurs sur fonctions

Exercice 1 : Générateur automatique de lettres
Première version du code :

(fichier src/lettre1.c)

```
#include <stdio.h>

void genereLettre(void)
{
  printf(
    "Bonjour chère Mireille,\n"
    "Je vous écris à propos de votre cours.\n"
    "Il faudrait que nous nous voyons le 18/12 pour en discuter.\n"
    "Donnez-moi vite de vos nouvelles !\n"
    "Amicalement, John.\n"
  );
}

int main(void)
{
  genereLettre();
  return 0;
}
```

Seconde version du code :

(fichier src/lettre2.c)

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

 typedef enum { MASCULIN, FEMININ } Genre;

void genereLettre(Genre genre, const char* destinataire, const char* sujet,
                  unsigned int jour, unsigned int mois, const char* politesse,
                  const char* auteur)
{
  printf("Bonjour ");
  if (genre == MASCULIN)
    printf("cher");
  else
```

```
    printf("chère");
  printf(" %s,", destinataire);

  printf(
    "Je vous écris à propos de %s.\n"
    "Il faudrait que nous nous voyons le %d/%d pour en discuter.\n"
    "Donnez-moi vite de vos nouvelles !\n"
    "%s, %s.\n"
    , sujet, jour, mois, politesse, auteur);
}

int main(void)
{
  genereLettre(FEMININ, "Mireille", "votre cours" , 18, 12, "Amicalement",
               "John");
  putchar('\n');
  genereLettre(MASCULIN, "John", "votre demande de rendez-vous", 16, 12,
               "Sincèrement", "Mireille");
  return 0;
}
```

Note :

1. L'utilisation du type `char*` dans le prototype de la fonction permet de passer des chaînes de caractères en paramètre, comme illustré dans le `main`.
2. On utilise ici des `const` pour ces arguments, car cette fonction ne modifie pas leur valeur. Cela évite aussi les risques de mauvaises manipulations de constantes littérales.

Exercice 2 : Segmentation en mots
(fichier src/token.c)

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

/* La fonction suivante teste si le caractère est un séparateur
 *
 * Écrire une fonction présente l'avantage de pouvoir redéfinir facilement
 * la notion de séparateur (et éventuellement d'en définit plusieurs)
 */
int issep (char c) {
  return (c == ' '); /* retourne 1 si la condition est vérifiée, 0 sinon */
}

/* Il y a *plein* d'autres façons d'écrire cette fonction.
 *
 * Je trouve celle-ci élégante.
```

```
 */
int nextToken(char const * str, int* from, int* len)
{
  const int taille = strlen(str); /* taille totale de la ligne entrée */
  int i;

  /* D'abord, on saute tous les séparateurs avant le premier
   * mot à partir de from.
   * Notez que *from représente la valeur pointée par from,
   * càd l'index qu'on a donné en paramètre.
   */
  while ( (*from < taille) && issep(str[*from]) ) {
    ++(*from); /* on veut incrémenter la valeur pointée par from, pas son adresse! */
  }

  /* Maintenant, from pointe sur l'index de la première lettre
   * du premier mot qui nous intéresse.
   * On avance jusqu'au prochain séparateur ou la fin de str.
   */
  *len = 0;
  for (i = *from; ((i < taille) && !issep(str[i])); ++(*len), ++i);

  return (*len != 0);
}

/* ---------------------------------------------------------------------- */

/* On définit une TAILLE_MAX pour la phrase qui sera entrée par
    l'utilisateur. */
#define TAILLE_MAX 1024

int main(void)
{
  char phrase[TAILLE_MAX+1];
  char mot[TAILLE_MAX+1];
  int debut = 0;
  int longueur = 0;
  int taille_lue;

  do {
    printf("Entrez une chaîne : ");
    fgets(phrase, TAILLE_MAX, stdin);
    taille_lue = strlen(phrase) - 1;
    /* On supprime le "\n" lu, càd le dernier caractère
     * (en le remplaçant par "\0")
     */
    if ((taille_lue >= 0) && (phrase[taille_lue] == '\n'))
```

```
    phrase[taille_lue] = '\0';
  } while ((taille_lue < 1) && !feof(stdin));

  printf("Les mots de \"%s\" sont :\n", phrase);
  while (nextToken(phrase, &debut, &longueur)) {

    /* On copie dans la variable mot les caractères
     * de la phrase depuis l'index debut,
     * de la taille longueur.
     */
    strncpy(mot, &(phrase[debut]), longueur);

    /* On rajoute un '\0' pour indiquer la fin du mot
     * (sinon on ne peut pas employer %s).
     */
    mot[longueur] = '\0'; /* fin de mot */
    printf("'%s'\n", mot);
    debut += longueur;
  }
  return 0;
}
```

Note: Dans le `main`, l'appel de la fonction `nextToken` utilise comme paramètres `&debut` et `&longueur`. On doit placer un `&` devant ces variables, car elles ne sont pas des pointeurs, et la fonction veut des pointeurs en paramètres. En ajoutant ce `&` devant les variables, on passe leur adresse en paramètre, et non pas leur valeur.

Exercice 3 : Intégrales revisitées
(fichier src/integrale2.c)

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

double f1(double x) { return x*x;             }
double f2(double x) { return sqrt(exp(x));    }
double f3(double x) { return log(1.0+sin(x)); }

typedef double (* Fonction)(double);

double demander_nombre(void)
{
  double res;
  printf("Entrez un nombre réel : ");
  scanf("%lf", &res);
  return res;
}
```

```
Fonction demander_fonction(void)
{
  int rep;
  Fonction choisie;
  do {
    printf("De quelle fonction voulez vous calculer l'intégrale [1-5] ? ");
    scanf("%d", &rep);
  } while ((rep < 1) || (rep > 5));

    switch (rep) {
    case 1: choisie = f1 ; break ;
    case 2: choisie = f2 ; break ;
    case 3: choisie = f3 ; break ;
    case 4: choisie = sin ; break ;
    case 5: choisie = exp ; break ;
    }

  return choisie;
}

double integre(Fonction f, double a, double b)
{
  double res;
  res =  41.0 * ( f(a) + f(b) )
      + 216.0 * ( f((5*a+b)/6.0) + f((5*b+a)/6.0) )
      +  27.0 * ( f((2*a+b)/3.0) + f((2*b+a)/3.0) )
      + 272.0 * f((a+b)/2.0) ;
  res *= (b-a)/840.0;
  return res;
}


int main(void)
{
  double a = demander_nombre();
  double b = demander_nombre();

  Fonction choix = demander_fonction();

  printf("Integrale entre %f et %f : %f\n", a, b,
         integre(choix, a, b));
  return 0;
}
```

Exercice 4 : mini VM

(fichier src/interp_cmd.c)

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

typedef void (*Cmd)(void* data);

// Notre machine simpliste
void print(void* data);
void add(void* data);
void push(void* data);
void pop(void* data);
void quit(void* data);

Cmd interprete(const char* nom_commande);

// ----------------------------------------------------------------------
int main(void)
{
  double registres[] = { 0.0, 0.0 };

  Cmd cmd = quit;
  do {
    char lu[] = "nom de la plus longue commande";
    printf("Entrez une commande (print, add, push, pop, quit) : ");
    scanf("%s", lu);
    (cmd = interprete(lu))(registres);
  } while (cmd != quit);

  return 0;
}

// ----------------------------------------------------------------------
Cmd interprete(const char* nom)
{
  if (nom == NULL) return quit;

  if ( ! strcmp(nom, "print") ) {
    return print;
  } else
  if ( ! strcmp(nom, "add") ) {
    return add;
  } else
  if ( ! strcmp(nom, "push") ) {
    return push;
  } else
  if ( ! strcmp(nom, "pop") ) {
```

```
    return pop;
  }
  return quit;
}

// ----------------------------------------------------------------------
void print(void* data)
{
  const double * const px = data;
  printf("-> %g\n", *px);
}

// ----------------------------------------------------------------------
void add(void* data)
{
  double * const regs = data;
  regs[0] += regs[1];
}

// ----------------------------------------------------------------------
void push(void* data)
{
  double * const regs = data;
  regs[1] = regs[0];
  printf("Valeur ? ");
  scanf("%lf", regs);
}

// ----------------------------------------------------------------------
void pop(void* data)
{
  double * const regs = data;
  regs[0] = regs[1];
}

// ----------------------------------------------------------------------
void quit(void* useless)
{
  puts("Bye!");
}
```

Dernière mise à jour : Dernière mise à jour le 15 mars 2016

Last modified: Tue Mar 15, 2016