TP n° 2 : Vie et mort des variables en mémoire

Exercice 1 : Modèle de pile

Soit le programme suivant (attention, il continue sur la page suivante) :

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

struct point {
  double x;
  double y;
};
typedef struct point sPoint;


double dist(sPoint p1, sPoint p2)
{
  printf("Dans la fonction dist : \n");
  printf("----------------------- \n");
  printf("Adresse de p1       : %u\n", (unsigned int) &p1);
  printf("Adresse de p2       : %u\n\n\n", (unsigned int) &p2);

  return sqrt((p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y));
}


double longueurchemin(const sPoint * chemin, int nb)
/* Preconditions : chemin est un tableau contenant au moins nb points,
                   avec nb avec superieur ou egal a 2
   Resultat : calcule la longueur du chemin de points */
{
  double res;
  double d1, d2;

  printf("Dans la fonction longueurchemin : \n");
  printf("--------------------------------- \n");
  printf("Adresse de nb       : %u\n", (unsigned int) &nb);
  printf("Adresse de chemin   : %u\n", (unsigned int) &chemin);
  printf("Adresse de res      : %u\n", (unsigned int) &res);
  printf("Adresse de d1       : %u\n", (unsigned int) &d1);
  printf("Adresse de d2       : %u\n\n\n", (unsigned int) &d2);

  if (nb==2) res = dist(chemin[1], chemin[0]);
  else
    {
      d1 = dist(chemin[nb-1], chemin[nb-2]);
      d2 = longueurchemin(chemin, nb-1);
      res = d1 + d2;
    }
  return res;
}


int main()
{
  double perimetre;
  sPoint cheminTriangle[4];

  cheminTriangle[0].x = 0.0 ;
  cheminTriangle[0].y = 0.0 ;
```

```
  cheminTriangle[1].x = 3.0 ;
  cheminTriangle[1].y = 0.0 ;
  cheminTriangle[2].x = 3.0 ;
  cheminTriangle[2].y = 1.0 ;
  cheminTriangle[3].x = cheminTriangle[0].x ;
  cheminTriangle[3].y = cheminTriangle[0].y ;

  /* Les adresses sont affichees en base 10 (%u) et non en base 16 (%p)
     pour plus de lisibilite */
  printf("Adresse de perimetre: %u \n\n\n", (unsigned int) &perimetre);

  printf("Adresses des elements du tableau : \n");
  printf("---------------------------------- \n\n");
  printf(" |-cheminTriangle[3].y : %u\n", (unsigned int) &cheminTriangle[3].y);
  printf(" |-cheminTriangle[3].x : %u\n", (unsigned int) &cheminTriangle[3].x);
  printf("cheminTriangle[3] : %u\n\n", (unsigned int) &cheminTriangle[3]);
  printf(" |-cheminTriangle[2].y : %u\n", (unsigned int) &cheminTriangle[2].y);
  printf(" |-cheminTriangle[2].x : %u\n", (unsigned int) &cheminTriangle[2].x);
  printf("cheminTriangle[2] : %u\n\n", (unsigned int) &cheminTriangle[2]);
  printf(" |-cheminTriangle[1].y : %u\n", (unsigned int) &cheminTriangle[1].y);
  printf(" |-cheminTriangle[1].x : %u\n", (unsigned int) &cheminTriangle[1].x);
  printf("cheminTriangle[1] : %u\n\n", (unsigned int) &cheminTriangle[1]);
  printf(" |-cheminTriangle[0].y : %u\n", (unsigned int) &cheminTriangle[0].y);
  printf(" |-cheminTriangle[0].x : %u\n", (unsigned int) &cheminTriangle[0].x);
  printf("cheminTriangle[0] : %u\n\n\n", (unsigned int) &cheminTriangle[0]);

  perimetre = longueurchemin(cheminTriangle, 4);
  printf("Le perimetre du triangle vaut %f\n", perimetre);

  return 0;
}
```

Avant de compiler et d'exécuter ce programme, répondez aux questions suivantes.

a) Combien d'octets une variable de type sPoint occupe-t-elle en mémoire (sur une machine où un int occupe 4 octets et un double 8) ?

b) Combien d'octets occupe le tableau cheminTriangle ?

c) La fonction longueurchemin est-elle récursive ? Même question pour la fonction dist.

d) A quoi sert le mot-clé const dans l'entête de la fonction longueurchemin ?

e) Dessinez sur une feuille séparée l'évolution de la pile lors de l'exécution de ce programme, en utilisant le modèle théorique de pile vu en cours et en TD.

f) Combien de fois la fonction longueurchemin est-elle appelée ? Même question pour la fonction dist.

g) Que pensez-vous de l'implantation récursive de longueurchemin, en termes de ressources mémoire ?

h) Quelle valeur obtenez-vous pour la variable perimetre ? Indication : $\sqrt{10} \approx 3{,}162$.

Exercice 2 : Comparer le modèle de pile avec la réalité

Nous allons à présent comparer l'évolution théorique de la pile avec ce qui se passe en réalité. Récupérez le fichier modele_pile.c sur le site Spiral de l'UE, compilez-le en utilisant la commande `gcc -Wall -ansi -pedantic -lm -o modele_pile.out modele_pile.c` et exécutez-le. Vérifiez que vous aviez bien anticipé la valeur de la variable perimetre. Si ce n'est pas le cas, appelez votre encadrant de TP pour qu'il vous aide à localiser la ou les erreurs dans votre évolution théorique de pile. Vous pouvez ensuite répondre aux questions suivantes.

a) D'après les adresses que vous obtenez à l'écran, dans quel ordre sont empilés les éléments du tableau cheminTriangle ? Pourquoi, à votre avis ?

b) Toujours d'après les adresses que vous obtenez à l'écran, dans quel ordre les paramètres sont-ils empilés lors des appels de fonction ?

c) Quelle est la taille allouée pour le paramètre chemin lors d'un appel à longueurchemin ? Pourquoi ?

d) Quel écart observez-vous entre l'adresse la plus haute et l'adresse la plus basse, parmi les adresses affichées ? Cet écart correspond-il à l'écart théorique (celui de votre trace « papier ») ? Essayez de localiser les sources de cette différence.

e) Ouvrez le fichier source (modele_pile.c) dans gedit et éditez-le pour ajouter, juste avant le return du main, la déclaration et la définition d'une nouvelle variable de type double. Ajoutez également une instruction pour afficher son adresse. Recompilez et réexécutez le programme. L'emplacement mémoire de cette nouvelle variable correspond-il à ce que vous attendiez ? Qu'en concluez-vous ?

Exercice 3 : Découverte de « scanf »

Au cours du TP précédent, vous avez manipulé la fonction printf, qui permet l'affichage à l'écran. Vous allez maintenant découvrir sa sœur : la fonction scanf, qui permet de récupérer les informations de l'entrée standard (il s'agira pour nous des informations saisies au clavier). Pour utiliser scanf, il faut avoir compris la notion d'adresse mémoire.

Exemple :

```
#include <stdio.h>

int main()
{
  double a;
  int i ;
  int valretour;

  printf("Donnez un nombre réel, puis un entier : \n") ;

  valretour = scanf("%lf %d", &a, &i);
  if (valretour == 2) {printf("Vous avez donné les valeurs %f et %d\n", a, i);}
  else {printf("Caractère invalide ou fin de saisie\n");}

  return 0;
}
```

Quelques explications :

- scanf lit une suite de caractères sur l'entrée standard et effectue les opérations de formatage indiquées dans la chaîne de format (1er paramètre). **Les autres paramètres indiquent les emplacements mémoire (adresses)** où scanf doit placer les informations formatées.

Type de la variable	Code de conversion	Remarques
char unsigned char signed char	%c	Contrairement à ce qui se passe pour printf, on ne peut pas lire un caractère par son code numérique
signed short int	%hd	
unsigned short int	%hu	
(signed) int	%d	
unsigned int	%u	
(signed) long	%ld	
unsigned long	%lu	
float	%f, %e ou %g	
double	%lf, %le ou %lg	
void *	%p	
char *	%s ou %*n*s	- Ignore les éventuels espaces initiaux - S'arrête au premier caractère « blanc » rencontré - Attention de bien disposer de la place nécessaire à l'adresse correspondante (contrôle possible : si *n* est précisé, au plus *n* caractères sont lus) - Un '\0' de fin est ajouté
	%*n*c	- Lit exactement *n* caractères - Ne supprime pas les espaces initiaux - Un espace n'interrompt pas l'analyse - Le '\0' n'est pas ajouté, il faut le faire manuellement

- scanf retourne le nombre de valeurs lues convenablement, ou bien la valeur spéciale EOF (généralement -1, en tout cas toujours négative) si l'utilisateur a tapé une combinaison de touches qui est interprétée comme une fin de fichier (CTRL+D sous Unix).
- L'utilisateur peut modifier les informations en cours de frappe (par retour arrière) car scanf attend une « validation » (retour chariot) avant de tenter effectivement de formater les données tapées. Ce mécanisme fait intervenir en interne un emplacement mémoire appelé « tampon » (buffer) dans lequel sont rangées provisoirement les informations tapées avant

qu'elles ne soient effectivement exploitées par scanf. Différents cas peuvent se présenter lorsque l'utilisateur valide sa ligne :

- le tampon contient exactement le nombre voulu de données (ici un réel et un entier) : tout se passe bien, scanf convertit les informations de façon appropriée et les place dans les emplacements désignés et le programme continue son cours.
- le tampon contient plus de données qu'attendu (ex. un réel et trois entiers) : comme précédemment, mais les données supplémentaires restent disponibles pour un (éventuel) prochain appel à scanf.
- le tampon ne contient pas toutes les informations voulues au moment de la validation (ex. seulement un réel) : scanf exploite les données tapées puis attend que l'utilisateur complète sa saisie en tapant une nouvelle ligne. Le programme reste « bloqué » sur le scanf tant que toutes les informations n'ont pas été saisies.

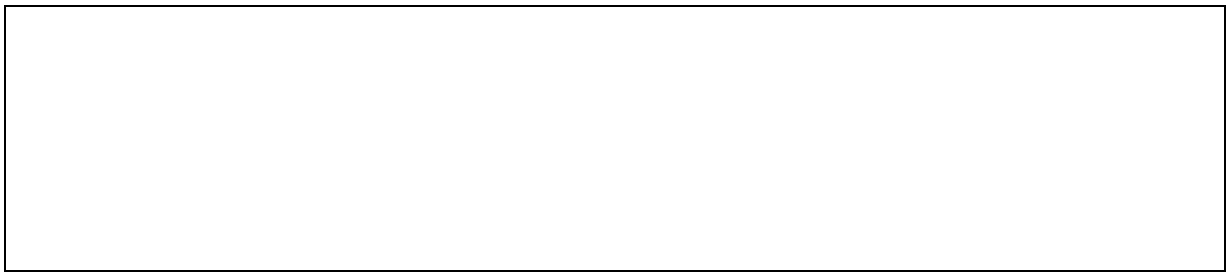
- Pour exploiter les informations du tampon, scanf utilise un indicateur de position appelé « pointeur de tampon » qui pointe sur le premier caractère non encore pris en compte. Tous les codes de format numériques (%d, %u, %f, %lf…) ainsi que le code %s provoquent l'avancement de ce pointeur sur le premier caractère qui n'est ni un espace ni une fin de ligne : les espaces et les retours chariot qui précèdent éventuellement l'information sont ignorés. En revanche, le code %c (caractère) prélève directement le caractère désigné par le pointeur, même s'il s'agit d'un espace ou d'un retour chariot.
- Lorsque scanf lit une information de type caractère (%c) ou chaîne de caractère (%s), tout caractère est acceptable. En revanche, pour les informations numériques, seuls certains caractères sont valides. Par exemple, pour le code %d, une lettre ne conviendra pas. Lorsque le pointeur de tampon pointe sur un caractère invalide pour le code de format spécifié, deux situations peuvent se présenter :
  - on dispose, avant le caractère invalide, d'autres caractères (autres que des espaces) qui permettent de « fabriquer » effectivement une valeur (par exemple, l'utilisateur a tapé « 456ty » pour un %d). Dans ce cas, la valeur 456 est stockée à l'emplacement désigné et le pointeur de tampon reste sur le 't'.
  - on ne dispose d'aucun caractère permettant de fabriquer une valeur (par exemple, frappe de « abc » pour un %d). Dans ce cas, scanf n'affecte aucune valeur à l'emplacement désigné et interrompt son traitement sans tenir compte des éventuels codes de format suivants et donc des autres informations à lire. On parle d'arrêt prématuré de scanf. Pour savoir si ce problème s'est produit, il faut examiner la valeur de retour de scanf. Remarque importante : le tampon n'est pas vidé, et le pointeur de tampon reste positionné sur le caractère invalide.

L'objectif final de cet exercice est d'écrire un sous-programme C de saisie « robuste » d'un nombre à virgule flottante, en utilisant scanf. Procédons par étapes.

a) Voici l'entête du sous-programme à produire en langage algorithmique :

```
Procédure initialisation_robuste_flottant(r : réel)
Précondition : aucune
Postcondition : r contient un nombre à virgule flottante lu sur l'entree
standard
Paramètres en mode donnée : aucun
```

Comment implanter cette entête en C ? Indiquez aussi comment la procédure est appelée.



b) Créez un fichier saisieRobusteFlottant.c et définissez une première version de la procédure `initialisation_robuste_flottant`. En cas de caractère invalide ou de fin de fichier, la procédure affichera un message d'erreur à l'écran et provoquera l'arrêt du programme. Utilisez pour cela l'instruction `exit(EXIT_FAILURE);` (avec `#include <stdlib.h>` en début de fichier). Ecrivez également un programme principal (main) qui appelle cette procédure pour initialiser une variable et qui affiche ensuite la valeur de cette variable avec deux chiffres après la virgule. Compilez et testez votre programme.

c) Modifiez la procédure pour qu'en cas de caractère invalide, elle demande à l'utilisateur de recommencer sa saisie au lieu de provoquer l'arrêt du programme. La saisie devra être recommencée jusqu'à ce que le scanf réussisse. Attention à la gestion du tampon…

d) Dernière étape d'amélioration de la procédure : modifiez-la pour qu'elle provoque un arrêt du programme (exit) lorsque l'utilisateur appuie sur CTRL+D au lieu de saisir une valeur.

/* Fin de la partie obligatoire */

Exercice 4 (pour les plus avancés) : Saisie d'un chemin de points

Le but de cet exercice est de reprendre le programme de l'exercice 1, et de modifier le « main » pour qu'il demande à l'utilisateur le chemin dont on va calculer la longueur totale. Procédons par étapes.

a) Créez une copie de modele_pile.c que vous appelerez saisie_chemin.c (ligne de commande : `cp modele_pile.c saisie_chemin.c`). Ouvrez saisie_chemin.c dans gedit. Supprimez les instructions d'affichage des adresses des variables.

b) Ajoutez dans ce fichier la procédure `initialisation_robuste_flottant` que vous avez codé dans l'exercice précédent.

c) Ajoutez une procédure `initialisation_robuste_entier` en vous inspirant de la procédure précédente.

d) Modifiez à présent le main : remplacez la déclaration et l'initialisation du tableau statique cheminTriangle par un morceau de code qui :
   - demande à l'utilisateur le nombre de points du chemin qu'il va saisir, en utilisant `initialisation_robuste_entier`,
   - vérifie que le nombre saisi est au moins égal à 2, et sinon, redemande une saisie jusqu'à ce que le nombre saisi le soit,
   - prépare un tableau monChemin pour recueillir ces points (**attention : la taille de ce tableau n'est pas connue au moment de la compilation**),
   - demande à l'utilisateur les coordonnées de chaque point, en utilisant `initialisation_robuste_flottant`.

   Le programme principal calculera ensuite la longueur totale du chemin, l'affichera et se terminera **proprement**.

e) Ajoutez une instruction pour afficher l'adresse de monChemin[0], en base 10. Que constatez-vous ?