T.P. numéro IV

Visibilité d'une variable – passsage de paramètres

1 Variables globales / variables locales

1.1 Variables globales

En C, plusieurs fonctions peuvent partager des variables communes. Ces variables s'appellent des **variables globales**. Elles sont déclarées en dehors des fonctions et son visibles de toutes les fonctions déclarées après dans un même code source. L'espace mémoire alloué à cette variable est le même pendant toute l'exécution du programme. Ces variables sont initialisées à zéro par défaut.

Exemple :

```
#include <stdio.h>

int i;              /* globale */

void affichage() {
  printf("passage %d\n",i);
}

main() {
  for (i=1 ; i<=5; i++) {
    affichage();
  }
}
```

Résultat :

```
passage 1
passage 2
passage 3
passage 4
passage 5
```

Dans cet exemple, la variable globale i est accessible dans les deux fonctions main() et affichage().
Remarque : N'importe quelle fonction ayant accès à une variable globale peut éventuellement la modifier. Ce type de variables est donc à utiliser avec prudence et dans des cas limités.

1.2 Variables locales

Les variables déclarées au sein d'une fonction sont dites **variables locales**. Ces variables ne sont « connues » que dans la fonction dans laquelle elles sont déclarées. Elles n'ont aucun lien avec les éventuelles variables globales ou avec les éventuelles variables locales d'une autre fonction qui porteraient le même nom.
Les variables locales ont une « durée de vie » limitée à celle de l'exécution de la fonction dans laquelle elles figurent. À chaque entrée dans la fonction, un nouvel espace mémoire est alloué pour les variables locales. Les variables figurant en argument d'une fonction sont considérées dans la fonction comme des variables locales.
On peut faire en sorte que des variables locales conservent leurs valeurs dans les passages successifs dans la fonction ou elles sont déclarées. Ce sont des **variables locales *statiques*** qui sont déclarées dans le code C avec l'option **static** :

```
static int a;
```

Ces variables statiques sont initialisées à zéro avant leur première utilisation.

Exemple :

```
#include <stdio.h>

int a = 5;      /* globale */

void fonc() {
  int a,b;      /* locales */
  a=1;
  b=2;
  printf("fonc : a = %d\n",a);
  printf("fonc : b = %d\n\n",b);
}

main() {
  int b=100;    /* locale */
  fonc();
  printf("main : a = %d\n",a);
  printf("main : b = %d\n",b);
}
```

Résultat :

```
fonc : a = 1
fonc : b = 2

main : a = 5
main : b = 100
```

Exercice IV.1

Sans compiler ni exécuter le code suivant

```
1  #include <stdio.h>
2
3  int a = 5;      /* globale */
4
5  void fonc() {
6    int b;      /* locale */
7    b=2;
8    printf("fonc : a = %d\n",a);
9    printf("fonc : b = %d\n\n",b);
10 }
11
12 main() {
13   int b=100;    /* locale */
14   fonc();
15   printf("main : a = %d\n",a);
16   printf("main : b = %d\n",b);
17 }
```

1. Y aura-t-il des erreurs lors de la compilation, nottament à la ligne 8 ? Justifier.
2. Si le code peut se compiler sans erreurs, indiquer les résultats affichés lors de l'exécution.

Exercice IV.2 *Comptage*

Télécharger le code comptage.c .

1. Compléter le corps de la fonction comptage() de ce code pour qu'elle affiche le résultat :

```
passage 1
passage 2
passage 3
passage 4
passage 5
```

   (on utilisera une variable locale déclarée avec l'option **static**).

2. Après avoir vérifié que le code fournit le bon résultat, expliquer ce qui se passerait sans l'option **static** .

---

2 Passage de paramètres à une fonction

2.1 « Passage par valeur »

Le « passage par valeur » est le mode **par défaut** de transmission des arguments dans les fonctions. C'est le mode qui a été utilisé lors des 3 séances de T.P. précédentes.
Toute variable d'un programme C est stockée dans une case de la mémoire qu'on appelle une *adresse*.
Le « passage par valeur » signifie que c'est la valeur contenue dans la case mémoire qui est transmise à la fonction et non son adresse. Cela impose des limites assez gênantes ...

Exercice IV.3 *Échange*

Télécharger le code echange.c .

1. Compléter la fonction echange() pour qu'elle réalise l'échange des deux variables a et b.
2. Compiler et exécuter le programme. Est-ce que les valeurs des variables n et p ont été échangées ? Expliquer le résultat obtenu.

Pour contourner les limites du « passage par valeurs » on peut transformer les variables locales en variables globales, mais ce n'est pas très satisfaisant.

2.2 « Passage par adresse »

Dans le « passage par valeur », la variable de la fonction *appelante* est copiée à une nouvelle adresse dès le début de la fonction *appelée*, et les modification éventuelles effectuées durant l'exécution de cette fonction n'affectent que la valeur stockée à la nouvelle adresse.
Pour pouvoir modifier la valeur d'une variable de la fonction *appelante* dans la fonction *appelée*, il faut pouvoir disposer pendant l'execution de cette fonction de l'adresse ou est stockée la valeur de la variable de la première fonction. Au lieu de transmettre à la fonction *appelée* la valeur de la variable, il faut en transmettre l'adresse d'où la formulation « passage par adresse ».

Notion de pointeurs : opérateurs & et *

On dit d'une adresse qu'elle *pointe* sur une variable. Les variables destinées à contenir des adresses sont donc appelés des **pointeurs**.
En C, il est possible de manipuler les pointeurs. Par exemple, pour une variable x quelconque, on accède au pointeur sur x, c'est-à-dire à l'adresse mémoire ou est stockée la valeur de x, par la construction "&x".

L'opérateur & signifie donc « adresse de ... ».

À l'inverse, si on dispose d'une variable p de type pointeur, c'est à dire une adresse, on accède à la valeur stockée à cette adresse par la construction "∗p".

*L'opérateur * signifie donc « valeur stockée à l'adresse ... ».*

En C, les pointeurs sont différents, suivant qu'ils pointent sur un entier ou un flottant, ou ... Quand on déclare une variable p comme étant un pointeur, on doit donc préciser de quel type de pointeur il s'agit. On utilise ici aussi l'opérateur *, comme dans les exemples :

```
int   *p1;
float *p2;
```

qui signifient que *p1 est entier et *p2 est flottant et donc que l'adresse p1 pointe sur un entier et l'adresse p2 pointe sur un flottant.
On peut aussi écrire de façon équivalente :

```
int*  p1;
float* p2;
```

qui signifient que p1 est un « pointeur sur un entier » et p2 un « pointeur sur un flottant ». Ainsi, pour chaque <type> de variable, il existe le type de pointeur associé <type>*.

Exemple :

```
#include <stdio.h>

main() {
  int*  adr;
  int   n;

  n = 20;
  printf("Ancienne : %d\n",n);
  adr = &n;
  *adr = 30;
  printf("Nouvelle : %d\n",n);
}
```

Résultat :

```
Ancienne : 20
Nouvelle : 30
```

Exercice IV.4 *Échange (suite)*

On veut reprendre le problème de l'exercice IV.3 en utilisant le « passage par adresse » dans la fonction echange().

1. Remplacer la ligne de code du programme principal qui appelle cette fonction par
   echange(&n,&p);
   et modifier l'entête et le corps de la fonction en conséquence.
2. Les variables n et p sont elles bien échangées ?

Exercice IV.5 *Conversion*

On veut créer un programme qui convertisse les coordonnées pôlaires en coordonnées carésiennes et réciproquement à l'aide des relations

$$\rho = \sqrt{x^2 + y^2} \qquad x = \rho \cos\varphi$$
$$\varphi = \arctan(y/x) \qquad y = \rho \sin\varphi$$

Télécharger le code conversion.c .

1. Compiler et éxecuter le programme en l'état. Les conversions demandées sont elles effectuées ?
2. Dans le programme principal, modifier les lignes de code qui passent les paramètres aux fonctions car_to_pol () et pol_to_car (). En fonction du résultat souhaité, décider quels paramètres doivent être passés « par adresse » et quels paramètres peuvent être passés « par valeur ».
3. Modifier les entêtes des fonctions car_to_pol () et pol_to_car () pour les rendre compatibles avec les modifications de la question 2. (la compilation doit se faire sans générer d'erreurs).
4. Ajouter le code qui effectue la conversion des coordonnées dans les deux fonctions. Prendre soin à convertir l'angle φ de radian en degré ou de degré en radian, sachant que l'argument des fonctions cos() et sin () est en radians, ainsi que la valeur renvoyée par la fonction atan (). La constante `pi` est déjà définie dans le code.
5. Compiler avec l'option `-lm` et exécuter le code sur plusieurs exemples pour vérifier qu'il fonctionne. Tester en particulier la conversion cartésien → pôlaire pour $x < 0$ ...

Exercice IV.6 *Trajectoire balistique*

On veut résoudre le problème de balistique de l'exercice III.2 du T.P. III de façon exacte. En effet, la position du point matériel dans ce problème est donnée analytiquement par les relations :

$$x(t) = x_0 + v_0 \cos(\alpha)\, t$$
$$y(t) = y_0 + v_0 \sin(\alpha)\, t - \frac{1}{2} g\, t^2$$

À partir de ces relations, on peut déterminer le temps $t_{\max}$ pour lequel l'altitude est maximale, et le temps $t_{\lim}$ pour lequel le point matériel retombe sur l'axe $y = 0$ qui détermine la portée.
On veut déterminer la trajectoire, l'altitude maximale et la protée de façon exacte à l'aide d'une seule fonction position () qui calcule pour un temps t donné les coordonnées x et y du point matériel en fonction également des conditions initiales (v_0, α, x_0, y_0).
Télécharger le code balistique.c

1. Déterminer les paramètres qui doivent être passés « par adresse » et ceux qui peuvent être passés « par valeur » à la fonction position ().
2. Compléter l'entête et le corps de la fonction position () en fonction de la réponse à la question 1.
3. Compléter les paramètres avec lesquels doit être appelée la fonction position () dans la boucle **do** du programme principal (ligne 37).
4. les temps $t_{\max}$ et $t_{\lim}$ sont calculés dans le programme. Compléter les paramètres avec lesquels doit être appelée la fonction position () aux lignes 44 et 45 pour calculer altitude et portee.
5. Compiler avec l'option `-lm` et exécuter le code. Comparer aux résultats obtenus au T.P. III. Conclusions ?

Listing des codes à télécharger

Code comptage.c

```
#include <stdio.h>

void comptage() {
  /* à compléter */
}

main() {
  int j;

  for (j=1; j<=5; j++) {
    comptage();
  }
}
```

Code echange.c

```
#include <stdio.h>

void echange(int a,int b) {
  /* à compléter pour inverser a et b */
  printf("Au début de echange() : %d, %d\n",a,b);
  /* à compléter pour inverser a et b */
  printf("A la fin de echange() : %d, %d\n",a,b);
}

main() {
  int n=1,p2;

  printf("Avant appel de echange() : %d, %d\n\n",n,p);
  echange(n,p);
  printf("\nAprès appel de echange() : %d, %d\n",n,p);
}
```

Code conversion.c

```
#include <stdio.h>
#include <math.h>
float pi=3.1415926535;

void car_to_pol(/* compléter ... */) {
  /* compléter ... */
}

void pol_to_car(/* compléter ... */) {
```

```
  /* compléter ... */
}

main() {
  float x,y;
  float rho,phi;
  int i;

  printf("Choisissez la conversion ...\n");
  printf("  1 : cartésien -> pôlaire\n");
  printf("  2 : pôlaire -> cartésien\n");
  scanf("%d",&i);
  switch (i) {
    case 1 :
       printf("Entrez x et y\n");
       scanf("%f %f",&x,&y);
       car_to_pol(/* compléter ... */);
       printf("rho = %f ; phi = %f deg.\n",rho,phi);
       return;
    case 2 :
       printf("Entrez rho et phi (deg.)\n");
       scanf("%f %f",&rho,&phi);
       pol_to_car(/* compléter ... */);
       printf("x = %f ; y = %f\n",x,y);
       return;
  }
}
```

Code balistique.c

```
#include <stdio.h>
#include <math.h>
#define g 9.81


position(/* compléter ... */) {
  /* compléter ... */
}

int main (void) {
  double x0,y0,v0;
  double alpha,pi=3.14159;
  double x,y,t,dt;
  FILE* fp;
  double tlim,tmax,delta;
  double portee,altitude;

  /* Initialisation */
  x0 = -10.;
  t = 0.;
  dt = 0.05;
  v0 = 15.;
```

```
  printf("angle alpha (en degrés) ?\n");
  scanf("%lf",&alpha);
  alpha = alpha*pi/180.;
  printf("ordonnée y ?\n");
  scanf("%lf",&y0);

  fp = fopen("trajectoire","w");
  fprintf(fp," %8.4f  %8.4f\n",x0,y0);

  x = x0;
  y = y0;
  do {
    t = t + dt;
    position(/* compléter ... */);
    fprintf(fp," %8.4f  %8.4f\n",x,y);
  } while (y>=0);

  tmax = v0*sin(alpha)/g;
  delta=v0*v0*sin(alpha)*sin(alpha) + 2*g*y0;
  tlim = (v0*sin(alpha) + sqrt(delta)) /g;
  position(tmax,/* compléter ... */);
  position(tlim,/* compléter ... */);
  portee = portee – x0;
  printf("portee = %f\naltitude max. = %f\n",portee,altitude);
}
```