INFO0902 - tutoriel C

Jean-Michel Begon, Romain Mormont

Dernière mise à jour le 22 février 2019

Affichage (printf)

Code :

```c
// file 'main.c'
#include <stdio.h>

int main() {
  double xref = 2.25, yref = 1.125;
  printf("reference: (%lf, %lf)\n", xref, yref);
  return 0;
}
```

Compilation :

```
gcc main.c -o main -Wall --std=c99
```

Résultat :

```
reference: (2.250000, 1.125000)
```

Condition

Objectif : afficher le quadrant dans lequel se trouve le point de référence.

```c
#include <stdio.h>

int main() {
  double xref = 2.25, yref = 1.125;
  printf("reference: (%lf, %lf)\n", xref, yref);
  // completer ici...
  return 0;
}
```

Résultat attendu :

```
reference: (2.250000, 1.125000)
quadrant: 1
```

Faire varier les coordonnées du points de référence pour s'assurer que le résultat est correct.

Structures conditionnelles

Choix binaire

```
if (expr) {
    ...
}

if (expr) {
    ...
} else {
    ...
}
```

Choix multiple

```
if (expr1) {
    ...
} else if (expr2) {
    ...
} else if (expr3) {
    ...
} else {
    ...
}
```

Expression conditionnelle

```
expr1 ? expr2 : expr3;
```

Condition (solution)

```c
#include <stdio.h>

int main() {
  double xref = 2.25, yref = 1.125;
  printf("reference: (%lf, %lf)\n", xref, yref);

  printf("quadrant: ");
  if (xref >= 0 && yref >= 0) {
      printf("1\n");
  } else if (xref < 0 && yref >= 0) {
      printf("2\n");
  } else if (xref < 0 && yref < 0) {
      printf("3\n");
  } else {
      printf("4\n");
  }

  return 0;
}
```

Variable

Objectif : en se basant sur le code initial, créer deux nouvelles variables `x` et `y` de type `double`, leur assigner respectivement les valeurs -1 et 5 puis afficher ces valeurs.

```c
// file 'main.c'
#include <stdio.h>

int main() {
  double xref = 2.25, yref = 1.125;
  printf("reference: (%lf, %lf)\n", xref, yref);
  // completer ici...
  return 0;
}
```

Résultat attendu :

```
reference: (2.250000, 1.125000)
point: (-1.000000, 5.000000)
```

Typage

Toute variable doit être déclarée en spécifiant son type et peut être initialisée au moment de sa déclaration.

Type primitif :

`bool`	true ou false (ISO-C99, avec stdbool.h)
`char`	caractère signé
`int`	entier signé
`size_t`	entier non-signé représentant une taille ou un indice
`float`	nombre réel (précision simple)
`double`	nombre réel (précision double)

Le typage du C est *statique* (le type d'une variable est déterminé à la compilation) et *faible* (une valeur peut être convertie implicitement vers le type adéquat).

Variable (solution)

```c
// file 'main.c'
#include <stdio.h>

int main() {
  double xref = 2.25, yref = 1.125;
  printf("reference: (%lf, %lf)\n", xref, yref);

  double x = -1, y = 5;
  printf("point: (%lf, %lf)\n", x, y);

  return 0;
}
```

Opérations

Objectif : compléter le code précédent pour calculer la distance euclidienne entre les points (`x`, `y`) et (`xref`, `yref`). Formule :

$$d_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Conseil : pour calculer la racine, utilisez la librairie `math.h`.
Compilation :

```
gcc main.c -o main -Wall --std=c99 -lm
```

Résultat attendu :

```
reference: (2.250000, 1.125000)
point: (-1.000000, 5.000000)
distance: 5.057482
```

Opérateurs

Par ordre de précédence :

postfixe	`[] . -> expr++ expr--`
préfixe	`++expr --expr +expr -expr ~ !`
	`&expr *expr sizeof (type)expr`
multiplicatifs	`* / %`
additifs	`+ -`
décalages	`<< >>`
comparaisons	`< > <= >=`
égalité	`== !=`
ET binaire	`&`
OU exclusif binaire	`^`
OU binaire	`|`
ET logique	`&&`
OU logique	`||`
conditionnel	`?:`
affectations	`= += -= *= /= %= <<= >>= \&= ^= |=`

Opérations (solution)

```c
#include <stdio.h>
#include <math.h> // math.h for sqrt

int main() {
  double xref = 2.25, yref = 1.125;
  printf("reference: (%lf, %lf)\n", xref, yref);

  double x = -1, y = 5;
  printf("point: (%lf, %lf)\n", x, y);

 // Compute the distance
  double dx = x - xref;
  double dy = y - yref
  double distance = sqrt(dx*dx + dy*dy);

  printf("distance: %lf\n", distance);
  return 0;
}
```

Fonctions

Objectif : isoler le calcul de distance dans une fonction dont le prototype est donné ci-dessous. Ensuite, appeler cette fonction depuis la fonction `main` pour calculer la distance entre (`x`, `y`) et (`xref`, `yref`) :

```
double dist(double x1, double x2,
            double y1, double y2);
```

Résultat attendu :

```
reference: (2.250000, 1.125000)
point: (-1.000000, 5.000000)
distance: 5.057482
```

Fonctions : syntaxe

```
int fct1(int a, int b) {
    ...
    return 4;
}
int fct2(void) {
    int a,b;
    ...
    return a+b;
}
void fct3(float b) {
    ...
    [return;]
}
```

- Une fonction peut prendre zéro, un ou plusieurs arguments.
- Les arguments sont passés par valeur.
- Chaque fonction renvoie une valeur d'un type donné, ou void.
- Si une fonction renvoie une valeur, elle doit posséder un instruction `return` correspondant au bon type.

Fonctions (solution)

```c
// file 'main.c'
#include <stdio.h>
#include <math.h>

double dist(double x1, double x2, double y1, double y2)
{
  double dx = x1 - x2, dy = y1 - y2;
  return sqrt(dx*dx + dy*dy);
}

int main() {
  double xref = 2.25, yref = 1.125;
  printf("reference: (%lf, %lf)\n", xref, yref);

  double x = -1, y = 5;
  printf("point: (%lf, %lf)\n", x, y);

  // Compute the distance
  printf("distance: %lf\n", dist(x, xref, y, yref);

  return 0;
}
```

Tableau–création

Objectif : supprimer `x` et `y` et créer un tableau `xs` et un tableau `ys` pour stocker respectivement l'abscisse et l'ordonnée des points $(-1, 5)$, $(-2, -2)$ et $(4, -3)$. Afficher les coordonnées du premier point et la distance à la référence

Résultat attendu :

```
reference: (2.250000, 1.125000)
point: (-1.000000, 5.000000)
distance: 5.057482
```

Tableau : syntaxe

```
int[5] a;
a[0] = 1;
a[4] = 42;
a[-1] = 10; // Bug!
a[5] = -5;  // Bug!
```

- Un tableau est un type de données indexable contenant des éléments du même type.
- Les éléments sont indexés à partir de 0 et jusqu'à N-1.
- Les tableaux sont passés aux fonctions par pointeurs (voir plus loin). Leurs modifications sont donc répercutées à l'appelant.

Tableau–création (solution)

```c
// file 'main.c'
#include <stdio.h>
#include <math.h>

double dist(double x1, double x2, double y1, double y2)
{
  double dx = x1 - x2, dy = y1 - y2;
  return sqrt(dx*dx + dy*dy);
}

int main() {
  double xref = 2.25, yref = 1.125;
  printf("reference: (%lf, %lf)\n", xref, yref);

  double xs[3] = {-1, -2, 4};
  double ys[3] = {5, -2, -3};
```

```
  printf("point: (%lf, %lf)\n", xs[0], ys[0]);

  // Compute the distance
  printf("distance: %lf\n", dist(xs[0], xref, ys[0],
         yref);

  return 0;
}
```

Tableau

Objectif : supprimer l'affiche du point $(-1, 5)$, ainsi que de sa distance. Calculer la distance entre le point de référence et tous les points. Afficher les coordonnées de ceux-ci, ainsi que la distance.
Résultat attendu :

```
reference: (2.250000, 1.125000)
point 0: (-1.0, 5.0) -- distance with ref.: 5.057482
point 1: (-2.0, -2.0) -- distance with ref.: 5.275237
point 2: (4.0, -3.0) -- distance with ref.: 4.480862
```

Boucle for

```
for (expr1; expr2; expr3) {
    ...
}
```

où

`expr1` une expression qui est exécutée avant le début de la boucle.

`expr2` le gardien de la boucle qui est testé avant chaque itération.

`expr3` une expression qui est exécutée à la fin de chaque itération.

`break` et `continue` :

- L'instruction `break` permet de quitter la boucle courante.
- L'instruction `continue` permet de passer à l'itération suivante, sans exécuter le restant de l'itération courante.

Tableau (solution)

```c
// file 'main.c'
#include <stdio.h>
#include <math.h>

double dist(double x1, double x2, double y1, double y2)
{
  double dx = x1 - x2, dy = y1 - y2;
  return sqrt(dx*dx + dy*dy);
}

int main() {
  double xref = 2.25, yref = 1.125;
  printf("reference: (%lf, %lf)\n", xref, yref);

  double xs[3] = {-1, -2, 4};
  double ys[3] = {5, -2, -3};
```

```
  double distance;
  for(size_t i = 0; i<3; i++)
  {
    distance = dist(xs[i], xref, ys[i], yref);
    printf("point %zu: (%lf, %lf)", i, xs[i], ys[i]);
    printf(" -- ");
    printf("distance with ref.: %lf\n", distance);
  }

  return 0;
}
```

Tableau et fonction

Objectif : créer une fonction qui calcule la distance entre un point de référence et les points des tableaux `xs` et `ys`. Substituer la fonction dans le code.

Conseil : utiliser un nouveau tableau `distances` pour pouvoir stocker les résultats.

Résultat attendu :

```
reference: (2.250000, 1.125000)
point 0: (-1.0, 5.0) -- distance with ref.: 5.057482
point 1: (-2.0, -2.0) -- distance with ref.: 5.275237
point 2: (4.0, -3.0) -- distance with ref.: 4.480862
```

Tableau et fonction (solution)

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

void arrayDist(double xref, double yref,
               double* xs, double* ys, size_t length,
               double* result)
{
    for(size_t i=0; i<length; i++)
        result[i] = dist(xref, xs[i], yref, ys[i]);
}

double dist(double x1, double x2, double y1, double y2)
{
  double dx = x1 - x2, dy = y1 - y2;
  return sqrt(dx*dx + dy*dy);
}
```

```c
int main() {
  double xref = 2.25, yref = 1.125;
  printf("reference: (%lf, %lf)\n", xref, yref);

  double xs[3] = {-1, -2, 4};
  double ys[3] = {5, -2, -3};

  double distances[3];
  arrayDist(xref, yref, xs, ys, 3, distances);
  for(size_t i = 0; i<3; i++)
  {
    printf("point %zu: (%lf, %lf)", i, xs[i], ys[i]);
    printf(" -- ");
    printf("distance with ref.: %lf\n", distances[i]);
  }

  return 0;
}
```

Allocation dynamique

Objectif : utiliser un tableau alloué dynamiquement pour stocker le résultat.

Résultat attendu :

```
reference: (2.250000, 1.125000)
point 0: (-1.0, 5.0) -- distance with ref.: 5.057482
point 1: (-2.0, -2.0) -- distance with ref.: 5.275237
point 2: (4.0, -3.0) -- distance with ref.: 4.480862
```

Allocation dynamique : syntaxe

```c
// Alloue un bloc de la taille d'un int
int* p = (int*) malloc(sizeof(int));

// Toujours vérifier le succès de malloc
if (!p) {
    printf("Error");
    return 1;
}

// On libère le bloc
free(p);
```

Allocation dynamique (solution)

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

double* arrayDist(double xref, double yref,
                  double* xs, double* ys,
                  size_t length)
{
    double* result = malloc(length*sizeof(double));
    if (!result)
        return NULL;

    for(size_t i=0; i<length; i++)
        result[i] = dist(xref, xs[i], yref, ys[i]);

    return result;
}
```

```
double dist(double x1, double x2, double y1, double y2)
{
  double dx = x1 - x2, dy = y1 - y2;
  return sqrt(dx*dx + dy*dy);
}

int main() {
  double xref = 2.25, yref = 1.125;
  printf("reference: (%lf, %lf)\n", xref, yref);

  double xs[3] = {-1, -2, 4};
  double ys[3] = {5, -2, -3};

  double* distances = arrayDist(xref, yref, xs, ys, 3);
  if (!distances) {
    fprintf(stderr, "Could not allocate array\n");
    return -1;
  }
```

```
  for(size_t i = 0; i<3; i++)
  {
    printf("point %zu: (%lf, %lf)", i, xs[i], ys[i]);
    printf(" -- ");
    printf("distance with ref.: %lf\n", distances[i]);
  }

  free(distances);

  return 0;
}
```

Structure

Objectif : A partir du premier code (sur le slide "Affichage (`printf`)"), utiliser une *structure* à la place de variables pour représenter le point de référence. Ensuite, afficher les coordonnées de ce point.

Résultat :

```
reference: (2.250000, 1.125000)
```

Structure (solution)

```c
// file 'main.c'
#include <stdio.h>

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

int main() {
  Point ref = {2.25, 1.125};
  printf("reference: (%lf, %lf)\n", ref.x, ref.y);
  return 0;
}
```

Structure et fonction

Objectif : modifier la fonction `dist` pour qu'elle prenne deux structures `Point` plutôt que les coordonnées. Utiliser la nouvelle fonction pour calculer et afficher la distance entre le point de référence et le point $(-1, 5)$.

Résultat attendu :

```
reference: (2.250000, 1.125000)
point: (-1.000000, 5.000000)
distance: 5.057482
```

Structure et fonction (solution)

```c
// file 'main.c'
#include <stdio.h>

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

double dist(Point p1, Point p2)
{
  double dx = p1.x - p2.x, dy = p1.y - p2.y;
  return sqrt(dx*dx + dy*dy);
}

int main() {
  Point ref = {2.25, 1.125};
```

```
  Point p = {-1, 5};
  printf("reference: (%lf, %lf)\n", ref.x, ref.y);
  printf("point: (%lf, %lf)\n", p.x, p.y);

  // Compute the distance
  printf("distance: %lf\n", dist(ref, p));

  return 0;
}
```

Structure et tableau

Objectif : allouer un tableau dynamique de structure `Point` pour stocker les coordonnées des points $(-1, 5)$, $(-2, -2)$ et $(4, -3)$. Ré-écrire la fonction `arrayDist` pour calculer les distances entre les points du tableau et le point de référence et afficher ces informations.

Résultat attendu :

```
reference: (2.250000, 1.125000)
point 0: (-1.0, 5.0) -- distance with ref.: 5.057482
point 1: (-2.0, -2.0) -- distance with ref.: 5.275237
point 2: (4.0, -3.0) -- distance with ref.: 4.480862
```

Structure et tableau (solution)

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

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

double dist(Point p1, Point p2)
{
  double dx = p1.x - p2.x, dy = p1.y - p2.y;
  return sqrt(dx*dx + dy*dy);
}
```

```
void arrayDist(Point pref, Point* points,
               size_t length, double* result)
{
    for(size_t i=0; i<length; i++)
        result[i] = dist(pref, points[i]);
}

int main() {
  Point ref = {2.25, 1.125};
  printf("reference: (%lf, %lf)\n", ref.x, ref.y);

  Point* points = malloc(3 * sizeof(Point));

  if (points == NULL) {
    return -1;
  }

  points[0].x = -1; points[0].y = 5;
```

```
points[1].x = -2; points[1].y = -2;
points[2].x = 4; points[2].y = -3;

double* result = malloc(3 * sizeof(double));

if (result == NULL) {
  free(points);
  return -1;
}

arrayDist(ref, points, 3, result);
for(size_t i = 0; i<3; i++)
{
  printf("point %zu: (%.1lf, %.1lf)", i, points[i].x,
         points[i].y);
  printf(" -- ");
  printf("distance with ref.: %lf\n", result[i]);
}
```

```
  free(points);

  return 0;
}
```

Modularité

Objectif : créer des fichiers `Point.h` et `Point.c` implémentant la structure `Point` et les fonctions `dist` et `arrayDist`. Même exercice que le précédent en utilisant cette fois-ci la structure et les fonctions définies dans l'interface `Point.h`.

Compilation :

```
gcc main.c Point.c -o main -Wall --std=c99 -lm
```

Résultat attendu :

```
reference: (2.250000, 1.125000)
point 0: (-1.0, 5.0) -- distance with ref.: 5.057482
point 1: (-2.0, -2.0) -- distance with ref.: 5.275237
point 2: (4.0, -3.0) -- distance with ref.: 4.480862
```

Modularité (solution)

```c
// fichier Point.h
#ifndef POINT_H
#define POINT_H
#include <stdlib.h>

typedef struct point {
  double x, y;
} Point;

Point* createPoint(double x, double y);
void freePoint(Point* p);
double dist(Point* p1, Point* p2);
void arrayDist(Point* pref, Point** points,
               size_t length, double* result);

#endif // POINT_H
```

```c
// fichier Point.c
#include <math.h>
#include "Point.h"

Point* createPoint(double x, double y) {
  Point* p = malloc(sizeof(Point));
  if (p == NULL) {
          return NULL;
  }
  p->x = x;
  p->y = y;
  return p;
}

void freePoint(Point* p) {
  free(p);
}
```

```
double dist(Point* p1, Point* p2) {
  double dx = p1->x - p2->x, dy = p1->y - p2->y;
  return sqrt(dx*dx + dy*dy);
}

void arrayDist(Point* pref, Point** points,
                size_t length, double* result)
{
    for(size_t i=0; i<length; i++)
        result[i] = dist(pref, points[i]);
}
```

```c
// fichier main.c
#include <stdio.h>
#include <stdlib.h>
#include "Point.h"

int main() {
  Point* ref = createPoint(2.25, 1.125);
  printf("reference: (%lf, %lf)\n", ref->x, ref->y);

  Point** points = malloc(3 * sizeof(Point*));

  if (points == NULL) {
    return -1;
  }

  points[0] = createPoint(-1, 5);
  points[1] = createPoint(-2, -2);
  points[2] = createPoint(4, -3);
```

```c
double* result = malloc(3 * sizeof(double));

if (result == NULL) {
  free(points);
  return -1;
}

arrayDist(ref, points, 3, result);
for(size_t i = 0; i<3; i++)
{
  printf("point %zu: (%.1lf, %.1lf)", i,
         points[i]->x, points[i]->y);
  printf(" -- ");
  printf("distance with ref.: %lf\n", result[i]);
}
```

```
  // Free memory resources
  freePoint(ref);

  for (size_t i = 0; i < 3; i++) {
    freePoint(points[i]);
  }
  free(points);

  return 0;
}
```

Opacité des structures

Objectif : changer la structure Point en une structure opaque.

Résultat attendu :

```
point 0 -- distance with ref.: 5.057482
point 1 -- distance with ref.: 5.275237
point 2 -- distance with ref.: 4.480862
```

Opacité de structures (solution)

```c
// fichier Point.h
#ifndef POINT_H
#define POINT_H
#include <stdlib.h>

// structure opaque
typedef struct point Point;

Point* createPoint(double x, double y);
void freePoint(Point* p);
double dist(Point* p1, Point* p2);
void arrayDist(Point* pref, Point** points,
               size_t length, double* result);

#endif // POINT_H
```

```c
// fichier Point.c
#include <math.h>
#include "Point.h"

struct point {
  double x, y;
};

Point* createPoint(double x, double y) {
  Point* p = malloc(sizeof(Point));
  if (p == NULL) {
          return NULL;
  }
  p->x = x;
  p->y = y;
  return p;
}
```

```
void freePoint(Point* p) {
  free(p);
}

double dist(Point* p1, Point* p2) {
  double dx = p1->x - p2->x, dy = p1->y - p2->y;
  return sqrt(dx*dx + dy*dy);
}

void arrayDist(Point* pref, Point** points,
                size_t length, double* result)
{
    for(size_t i=0; i<length; i++)
        result[i] = dist(pref, points[i]);
}
```

Opacité des structures (2)

Objectif : créer un nouveau fichier `PolarPoint.c` qui implémente l'interface `Point.h` en utilisant des coordonnées polaires :

$$r = \sqrt{x^2 + y^2}$$
$$\theta = \arctan_2(x, y)$$
$$d_{i,j} = \sqrt{r_1^2 + r_2^2 - 2 r_1 r_2 \cos(\theta_1 - \theta_2)}$$

Compilation :

```
gcc main.c PolarPoint.c -o main -Wall --std=c99 -lm
```

Résultat attendu :

```
point 0 -- distance with ref.: 5.057482
point 1 -- distance with ref.: 5.275237
point 2 -- distance with ref.: 4.480862
```

```c
// fichier PolarPoint.c
#include "Point.h"
#include <math.h>

struct point {
  double r, theta;
};

Point* createPoint(double x, double y) {
  Point* p = malloc(sizeof(Point));
  if (p == NULL) {
    return NULL;
  }
  p->r = sqrt(x * x + y * y);
  p->theta = atan2(y, x);
  return p;
}
```

```c
void freePoint(Point* p) {
  free(p);
}

double dist(Point* p1, Point* p2) {
  return sqrt(p1->r * p1->r + p2->r * p2->r \
              - 2 * p1->r * p2->r \
              * cos(p1->theta - p2->theta));
}

void arrayDist(Point* pref, Point** points,
               size_t length, double* result) {
  /* Code identique */
}
```