POLYTECH PARIS-SUD

Et3 - Tronc commun

Programmation C

TP6 - Structures de données

(Correction)

1 Objectifs :

- Savoir définir une structure dans le bon fichier
- Savoir déclarer une structure selon les deux manières possibles, en connaissant les différences et leurs avantages.
- Savoir gérer les structures imbriquées
- Savoir gérer les structures via des pointeurs

2 Rappel

N'oubliez pas de répartir votre code dans plusieurs fichiers, en regroupant les fonctions de manière logique. Ici, nous vous proposons de séparer comme suit :

- main.c
- point.c, point.h
- segment.c, segment.h
- rect.c, rect.h

N'oubliez pas de faire les `include` dans l'ordre nécessaire et de correctement renseigner les fichiers *.h !

Commentaires : *Rappel sur les structures :*

```
struct personne{
        char[10] nom;
        int age;
};
```

(Attention au ";"!)

Qui permet de déclarer de cette manière une structure :

```
struct personne pers1;
pers1.nom = ''Pierre'';
pers1.age = 24;
```

ou

```
struct personne pers2 = {''Paul'', 65};
```

Cependant, il faut à chaque fois écrire ***struct****. Pour y pallier, on redéfinit* ***struct personne*** *en* ***personne*** *grâce à* ***typedef*** *comme suit :*

```
typedef struct personne{
        char[10] nom;
        int age;
}personne;
```

Qui permet de déclarer une structure de cette manière :

```
personne pers2;
pers2.nom = ''Paul'';
pers2.age = 65;
```

Comme nous avons besoin probablement besoin des pointeurs, voyons comment ça fonctionne. Pour en déclarer un, il faut :

```
personne* pers2 = (personne*) malloc(sizeof(personne));
```

et pour accéder aux champs

```
(*pers2).nom = ``Paul'';
(*pers2).age = 65;
```

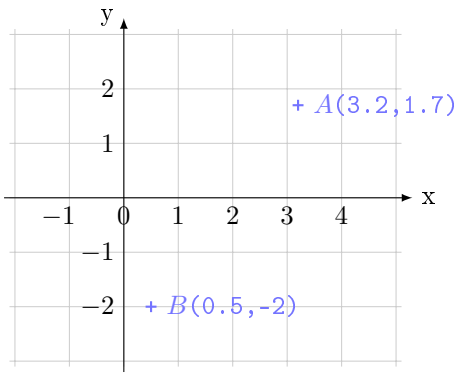
Mais c'est long comme notation. Une abréviation a été crée : la flèche -> :

```
pers2->nom = ``Paul'';
pers2->age = 65;
```

---

3 Exercice 1 : Des Points

Nous souhaitons définir une structure `point_t` contenant deux `double` `x` et `y`, les coordonnées d'un point, respectivement l'abscisse et l'ordonné, dans un plan muni d'un repère orthonormal. Exemple ici, le `A` de coordonnées `(3.2,1.7)` et le `B` de coordonnées `(0.5,-2)`.



1. Définissez cette structure `point_t`.

Correction :

```
// Dans point.h
typedef struct point_t{
        double x, y;
}point_t;
```

---

2. Pour créer une structure, deux possibilités s'offrent à nous :
   - Faire une fonction `point_create` qui va créer un point en fonction des paramètres `x` et `y` que nous lui auront fournis. Il faut de plus ajouter une procédure `point_delete` pour libérer la mémoire.
   - ou utiliser le raccourcis `point_t A = {3.2,1.7};`

   Quel est l'avantage de l'un par rapport à l'autre ? N.B. : Utilisez ici la deuxième solution.

Commentaires : *Il faut bien comprendre la différence : ils peuvent peuvent être tous deux utilisés car il n'y a pas d'invariant dans la création de notre point.*

Cependant, la deuxième solution est ici plus simple. Surtout qu'il n'y a pas besoin de libérer la mémoire après.

On verra plus loin avec le segment et le rectangle comment faire une fonction `create`.

Correction : *Si deuxième solution :*

```
// Dans point.c
point_t q = {1.65, 3.89};
```

Si première solution :

```
// Dans point.c
point_t* point_create(double abscisses, double ordonnes){
        point_t* p = (point_t*) malloc(sizeof(point_t));
        p->x = abscisses;
        p->y = ordonnes;
        return p;
}
```

Attention : Ici le retour est donc obligatoirement un ***point_t**** *Et donc pour appeler on fait :*

```
// Dans main.c
point_t* A = point_create(3.2, 1.7);
```

De plus, dans ce cas, il faut aussi une procédure ***point_delete*** *:*

```
// Dans point.c
void point_delete(point_t* p){
   free(p);
};
```

Appelée comme suit :

```
// Dans main.c
point_delete(A);
```

3. Nous voulons maintenant afficher les informations contenues dans un point. Écrivez la fonction `point_affiche(point_t const* p)`

Commentaires : *Nous utilisons* ***const*** *car dans cette fonction, nous n'avons pas besoin de modifier le* ***point_t*** *et nous voulons donc le protéger grâce à* ***const****. Nous pourrions faire* ***point_t const* const p*** *qui protège et le pointeur (le deuxième) et le* ***point_t*** *(le premier) mais dans les faits, modifier le pointeur, on s'en fout vu que c'est déjà une copie grâce à la fonction. Donc en ressortant, il ne sera pas modifié. Alors que le* ***point_t*** *aurait pu.*

Correction :

```
// Dans point.c
void point_affiche(point_t const*  p){
        printf("Point {%.2f, %.2f}\n", p->x, p->y);
}
```

Attention : pour faire l'appel, selon comment vous avez déclaré les ***point_t****, vous devrez passer le pointeur ou l'adresse de la variable :*

```
// Dans main.c
// Avec un create
point_affiche(p);
// Avec le raccourci
point_affiche(&q);
```

4. Nous voulons maintenant calculer la distance entre deux points. La signature de la fonction devra être la suivante :
`double point_distance(point_t const* p1, point_t const* p2)`.
Rappels :
   - $dist_{AB} = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$
   - Vous aurez besoin d'une bibliothèque de mathématique : à vous de trouver laquelle. Pour l'utiliser, dans la ligne de compilation, n'oubliez pas d'ajouter -lm.

Testez avec des valeurs connues !

Correction :

```c
// Dans point.c
double point_distance(point_t const* p1,  point_t const* p2){
        double dx = (p1->x - p2->x);
        double dy = (p1->y - p2->y);
        double distance = sqrt(dx*dx + dy*dy);
        return distance;
}
```

5. Nous souhaitons vérifier si deux points sont égaux. Votre fonction devra renvoyer 1 si oui, 0 sinon et prendra en paramètre deux `point_t const*`. (Deux points sont égaux si leurs x sont égaux ET leurs y).

Correction :

```c
// Dans point.c
int point_equal(point_t const*  p1, point_t const*  p2){
    int is_equal = 0;

    if((p1->x == p2->x)&& (p1->y == p2->y)){
        is_equal = 1;

    }
    return is_equal;
}
```

6. Enfin, nous voulons appliquer une translation à un point p d'une amplitude dx en abscisse et dy en ordonnés. Écrivez la fonction `void point_translate(point_t* p, double dx, double dy)`

Correction :

```c
// Dans point.c
void point_translate(point_t* p, int dx, double dy){
    p->x += dx;
    p->y += dy;
}
```

4 Exercice 2 : Puis des segments

Maintenant, nous allons créer des segments. Un `segment` sera défini par une structure contenant elle-même deux structures `point_t`.

1. Définissez cette structure `segment`.

Commentaires : *Notez qu'ici, nous utilisons des* `point_t` *et non des* `point_t*`*. Sinon, on peut modifier l'intérieur de la structure avec les pointeurs qu'on avait passé en paramètre de la fonction* `create` *!*

Correction :

```c
// Dans segment.h
typedef struct segment{
        point_t begin, end;
}segment;
```

2. Cette fois-ci, nous allez créer la fonction `segment* segment_create(point_t const* begin, point_t const* end)` pour tester les deux méthodes. Dans cette fonction, vous devez créer le `segment*`,

l'allouer puis remplir les champs de la structure avec les `point_t*` passés en paramètre. N.B. : Vous remarquerez que nous avons ici fait un choix sur les paramètres, mais qu'on aurait pu prendre 4 coordonnées en entrée.

Correction :

```c
// Dans segment.c
segment* segment_create(point_t* begin, point_t* end){
        segment* seg = (segment*) malloc(sizeof(segment));
        seg->begin = *begin;
        seg->end = *end;
        return seg;
}
```

3. Chaque structure avec un `create` doit obligatoirement avoir la procédure `delete` correspondante. La signature de ovtre fonction devra être `void segment_delete(segment* seg)`.

Correction :

```c
// Dans segment.c
void segment_delete(segment* seg) {
    free(seg);
}
```

4. Pour pouvoir vérifier les valeurs entrées dans un `segment`, définissez une procédure `void segment_affiche(segment const* seg)`.
Remarque : Pensez à utiliser la fonction déjà faite pour afficher les points.

Correction :

```c
// Dans segment.c
void segment_affiche(segment const* seg) {
    printf("\nSegment\n");
    point_affiche(&(seg->begin));
    point_affiche(&(seg->end));
}
```

5. Maintenant, nous voulons calculer la longueur d'un segment passé en argument de la fonction. Le résultat de votre fonction sera stocké sous forme d'un `double`.
Remarque : Pensez à utiliser la fonction déjà faite pour les points.

Correction :

```c
// Dans segment.c
double segment_longueur(segment const* seg){
    double distance = point_distance(&(seg->begin), &(seg->end));
    return distance;
}
```

6. De la même manière que pour les points, nous voulons savoir si deux segments sont égaux. Écrivez la fonction `int segment_equal(segment const* seg1, segment const* seg2)`. Attention, ici nos segments ont un début et une fin et sont donc orientés.
Remarque : Pensez à utiliser la fonction déjà faite pour les points.

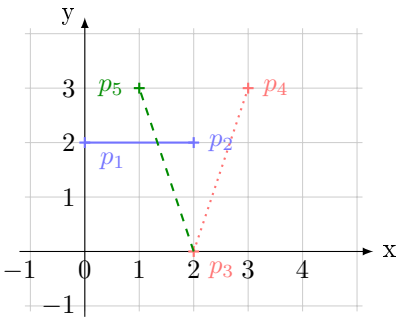
Correction :

```c
// Dans segment.c
int segment_equal(segment const* seg1, segment const* seg2){
    int is_equal = 0;

    if(point_equal(&(seg1->begin), &(seg2->begin))
       && point_equal(&(seg1->end), &(seg2->end))){
        is_equal = 1;
    }
    return is_equal;
}
```

7. Pour finir avec les segments, nous voulons définir si deux segments ont une intersection et si oui, renvoyer un `point_t*`. Pour se faire, nous vous proposons la méthodologie suivante pour votre fonction :
   - Tout d'abord calculer l'équation des droites (De la forme $y = a * x + b$). Si les coefficients directeurs ne sont pas égaux, cela signifie que les droites ne sont pas parallèles et donc qu'il y a une intersection.
   - Ensuite, il faut calculer le point d'intersection. (Défini par : $x_{inter} = (b' - b)/(a - a')$ et $y_{inter} = a * x_{inter} + b$)
   - Enfin, comme l'illustre le schéma ci-dessous, nous travaillons sur des segments et non des droites. Par exemple, le segment vert et bleu ainsi que le vert et rouge sont sécants tandis que le rouge et bleu ne sont pas sécants alors que leur équation de droite l'est. Il faut donc vérifier que le point calculé est bien sur les deux segments : vérifier que le x_{inter} est bien compris entre le x_{begin} et le x_{end} de chaque segment. De même pour le y.
     N.B. : Cette dernière partie peut aussi se faire avec des produits scalaires.... A vous de voir !

   Remarque :
   - Attention aux cas où le dénominateur peut être égal à 0 !
   - Vous aurez probablement besoin de créer des sous-fonctions.
   - Nous ne prenons pas en compte le cas où les deux segments sont sur la même droite, et peuvent donc avoir soit un point, soit un segment en commun.



Correction :

```c
// Dans segment.c
point_t* segment_intersect(segment const* seg1, segment const* seg2){
    point_t* intersect = NULL;
    // Si les dorites ne sont pas égales
    if(segment_equal(seg1, seg2) == 0){
        // Calcule des équations des droites
        // Pour seg 1 :
        // on a y = a1x + b1
        // Ou a1 = (x end - x begin) / (y end - y begin)
        // /!\ si ybegin = yend -> a tout de suite à 0
        double a1 = seg1->end.x - seg1->begin.x;
        double denominateur = seg1->end.y - seg1->begin.y;
        if(denominateur != 0){
           a1 = a1 / denominateur;
        }else{
            a1 = 0;
        }
        // et b = y begin - a1 * x begin
        double b1 = seg1->begin.y - a1 * seg1->begin.x;
        // Pour seg 2 :
        //on a y = a2 * x + b2 sur le même schéma :
        double a2 = seg2->end.x - seg2->begin.x;
        denominateur = seg2->end.y - seg2->begin.y;
        if(denominateur != 0){
           a2 = a2 / denominateur;
        }else{
            a2 = 0;
        }
```

```
        double b2 = seg2->begin.y - a2 * seg2->begin.x;
        // Si a != a' (Sinon, elles sont parallèles -> pas d'intersection)
        if(a1 != a2){
            double x_inter;
            double y_inter;
            // l'intersection est alors au point :
            // xj = (b' - b) / (a - a')
            x_inter = (b2 - b1);
            denominateur = a1 - a2;
            if(denominateur != 0){
                x_inter = x_inter / denominateur;
            }else{
                x_inter = 0;
            }
            // yj = a * xj + b
            y_inter = a1 * x_inter + b1;
            //printf("%f, %f", x_inter, y_inter);

            // Et à la fin, le point d'intersection doit bien être entre les 4
                points !!
            // Sinon pas d'intersection
            // Pour se faire, il faut vérifier que le x_inter et le y_inter sont
                bien entre respectivement les x begin et les x end, les y begin et
                les y end :
            if(entre_deux_val(seg1->begin.x, seg1->end.x, x_inter) == 1 &&
                entre_deux_val(seg1->begin.y, seg1->end.y, y_inter) == 1){
                if(entre_deux_val(seg2->begin.x, seg2->end.x, x_inter) == 1 &&
                    entre_deux_val(seg2->begin.y, seg2->end.y, y_inter) == 1){
                    intersect = (point_t*) malloc(sizeof(point_t));
                    intersect->x = x_inter;
                    intersect->y = y_inter;
                }
            }
         }
    }
    return intersect;
}

int entre_deux_val(double val1, double val2, double valentre){
    if((val1<=valentre && valentre<=val2) || (val2<=valentre && valentre<=val1)){
        return 1;
    }else{
        return 0;
    }
}
```

---

5 Exercice 3 : Et enfin des rectangles !

1. Nous définissons maintenant la structure `rect` qui est composée de 4 `point_t`. Les 4 points correspondent aux 4 sommets et sont stockés dans l'ordre horaire en commençant n'importe où.
   Note : Vous pouvez aussi choisir de stocker 2 segments, 1 segment et 2 points, etc. Mais chaque façon de faire a ses inconvénients.

Correction :

```
// Dans rect.h
typedef struct rect{
        point_t point1, point2, point3, point4;
}rect;
```

---

2. Cette fois, nous sommes obligés de créer une fonction `rect_create` qui prendra 4 `point_t*` en paramètre. En effet, nous avons ici un invariant à vérifier : les points sont-ils dans le bon ordre ? Et est-ce vraiment un rectangle ? Les points sont-ils tous distincts ?
   Pour ce faire, il vous suffit de vérifier s'il y a au moins 3 angles droits.
   Indice : nous pouvons utiliser le produit scalaire qui nous dit deux choses :

$$\vec{AB}.\vec{AC} = x_{AB} * x_{AC} + y_{AB} * y_{AC} \quad (1)$$

$$\vec{AB}.\vec{AC} = |AB| * |AC| * \cos\theta \quad (2)$$

Note : si $\cos\theta = 0$ alors $\theta = 90^\circ$

Remarque : nous vous conseillons de créer une sous-fonction `produit_scalaire(point_t const* A, point_t const* O, point_t const* B)` qui calcule la valeur $\cos\theta$ entre les vecteurs $\vec{OA}.\vec{OB}$.

Correction :

```c
// Dans rect.c
rect* rect_create(point_t const* p1, point_t const* p2, point_t const* p3, point_t
    const* p4){
    rect* rectangle = (rect*) malloc(sizeof(rect));
    // On doit enlever les cas où au moins 2 sont les mêmes points.
    if(!point_equal(p1, p2) && !point_equal(p1, p3) &&
       !point_equal(p1, p4) && !point_equal(p2, p3) &&
       !point_equal(p2, p4) && !point_equal(p3, p4)){
        //Nous allons maintenant, partant du principe que les points sont donnés
            dans un ordre (Aiguille d'une montre),
        // vérifier qu'il y a bien 3 angles droits pour valider que c'est un
            rectangle. (Cad que les valeur du cosinus sont bien = 0)
        if(produit_scalaire(p1, p2, p3) == 0 && produit_scalaire(p2, p3, p4) == 0
           && produit_scalaire(p3, p4, p1) == 0){

            rectangle->point1 = *p1;
            rectangle->point2 = *p2;
            rectangle->point3 = *p3;
            rectangle->point4 = *p4;
        }else{
            rectangle= NULL;
            printf("Ce n'est pas un rectangle !!\n ");
        }
    }
    return rectangle;
}

// Renvoie la valeur du cosinus de l'angle AOB
double produit_scalaire(point_t const* A, point_t const* O, point_t const* B){
    double resultat = 0;

    //Calcul des vecteurs OA et OB
    double xOA = A->x - O->x;
    double xOB = B->x - O->x;
    double yOA = A->y - O->y;
    double yOB = B->y - O->y;
    // On sait que vec(OA).vec(OB) = xOA.xOB + yOA.yOB
    // et que vec(OA).vec(OB) = |OA|.|OB|.cos(theta)
    // d'où on peut déduire cos(theta).
    // si = 0 c'est que théta = 90 et donc c'est un angle droit !
    resultat = point_distance(O, A)*point_distance(O, B);
    if(resultat != 0){
        resultat = (xOA * xOB + yOA * yOB)/resultat;
    }
    return resultat;
}
```

3. N'oubliez pas la règle d'or : à chaque `create` son `delete` !

Correction :

```c
// Dans rect.c
void rect_delete(rect* rectangle){
    free(rectangle);
}
```

4. Nous souhaitons afficher les informations contenues dans un rectangle. Écrivez la fonction `void rect_affiche(rect const* rectangle)`

Correction :

```c
// Dans rect.c
void rect_affiche(rect const* rectangle){
    if(rectangle != NULL){
        printf("\n Rectangle\n");
        printf("Point 1 : {%.2f, %.2f}\n", rectangle->point1.x, rectangle->point1.y)
            ;
        printf("Point 2 : {%.2f, %.2f}\n", rectangle->point2.x, rectangle->point2.y)
            ;
        printf("Point 3 : {%.2f, %.2f}\n", rectangle->point3.x, rectangle->point3.y)
            ;
        printf("Point 4 : {%.2f, %.2f}\n", rectangle->point4.x, rectangle->point4.y)
            ;
        printf("\n");
    }else{
        printf("Ce n'est pas un rectangle. \n");
    }
}
```

---

5. Nous voulons déterminer si deux rectangles sont égaux. Écrivez la fonction, sans oublier de pivoter les rectangles.

Correction :

```c
// Dans rect.c
int rect_equal(rect const* rectangle1, rect const* rectangle2){
    int resultat = point_equal(&(rectangle1->point1), &(rectangle2->point1)) ||
        point_equal(&(rectangle1->point1), &(rectangle2->point2)) || point_equal(&(
        rectangle1->point1), &(rectangle2->point3)) || point_equal(&(rectangle1->
        point1), &(rectangle2->point4));
    resultat = resultat && (point_equal(&(rectangle1->point2), &(rectangle2->point1)
        ) || point_equal(&(rectangle1->point2), &(rectangle2->point2)) ||
        point_equal(&(rectangle1->point2), &(rectangle2->point3)) || point_equal(&(
        rectangle1->point2), &(rectangle2->point4)));
    resultat = resultat && (point_equal(&(rectangle1->point3), &(rectangle2->point1)
        ) || point_equal(&(rectangle1->point3), &(rectangle2->point2)) ||
        point_equal(&(rectangle1->point3), &(rectangle2->point3)) || point_equal(&(
        rectangle1->point3), &(rectangle2->point4)));
    resultat = resultat && (point_equal(&(rectangle1->point4), &(rectangle2->point1)
        ) || point_equal(&(rectangle1->point4), &(rectangle2->point2)) ||
        point_equal(&(rectangle1->point4), &(rectangle2->point3)) || point_equal(&(
        rectangle1->point4), &(rectangle2->point4)));
    return resultat;
}
```

---

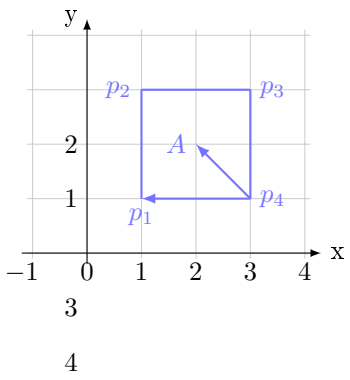
6. Calculez maintenant la surface d'un rectangle.
Remarque : N'oubliez pas que vous avez déjà codé des fonctions !

Correction :

```c
// Dans rect.c
double rect_surface(rect const* rectangle){
    double surface = point_distance(&(rectangle->point1), &(rectangle->point2)) *
        point_distance(&(rectangle->point2), &(rectangle->point3));
    return surface;
}
```

---

7. Nous voulons vérifier qu'un point est compris ou non dans un rectangle. Ecrivez la fonction.
Indices : Soit un rectangle défini par les 4 points p1, p2, p3 et p4 et un point A dans le plan, les 4 $\cos\widehat{Ap_4p_1}$, $\cos\widehat{Ap_1p_2}$, $\cos\widehat{Ap_2p_3}$ et $\cos\widehat{Ap_3p_4}$ sont compris entre entre 0 et 1 si et seulement si A est compris dans le rectangle. Pour les calculer, nous vous recommandons d'utiliser les formules des produits scalaires données précédemment.



Correction :

```c
// Dans rect.c
int rect_contains(rect const* rectangle, point_t const* point){
    // point est dans rectangle si et seulement si le cosinus de l'angle du produit
        scalaire est bien compris entre 0 et 1 inclus)
    int resultat = entre_deux_val(0,1,produit_scalaire(point, &(rectangle->point1),
        &(rectangle->point2)));
    resultat = resultat && entre_deux_val(0,1,produit_scalaire(point, &(rectangle->
        point2), &(rectangle->point3)));
    resultat = resultat && entre_deux_val(0,1,produit_scalaire(point, &(rectangle->
        point3), &(rectangle->point4)));
    resultat = resultat && entre_deux_val(0,1,produit_scalaire(point, &(rectangle->
        point4), &(rectangle->point1)));
    return resultat;
}
```

8. Pour finir, nous voulons déterminer si deux rectangles sont bien disjoints : `int rect_disjoint(rect const* rectangle1, rect const* rectangle2)`.
   Veillez à bien vérifier tous les cas.

Correction :

```c
// Dans rect.c
int rect_disjoint(rect const* rectangle1, rect const* rectangle2){
    //Il suffit de vérifier qu'il n'y a pas de sommet de l'un des triangles dans l'
        autre, mais aussi inversement ! (Cf cas du rectabgle complétement inclu)
    int resultat = 1;
    if(rect_equal(rectangle1, rectangle2)){
        resultat = rect_contains(rectangle2, &(rectangle1->point1)) || rect_contains
            (rectangle2, &(rectangle1->point2)) || rect_contains(rectangle2, &(
            rectangle1->point3)) || rect_contains(rectangle2, &(rectangle1->point4))
            ;
        resultat = rect_contains(rectangle1, &(rectangle2->point1)) || rect_contains
            (rectangle1, &(rectangle2->point2)) || rect_contains(rectangle1, &(
            rectangle2->point3))|| rect_contains(rectangle1, &(rectangle2->point4));
    }
    return !resultat;
}
```