Cours de programmation séquentielle

Polygones réguliers et comptage de références

1 Buts

— Allocation dynamique de mémoire (`malloc()` et `free()`) pour des structures.
— Utilisation de pointeurs partagés entre plusieurs structures.
— Notion de propriété de la mémoire: qui alloue, qui libère?
— Implémentation d'un comptage de références.
— Détection d'erreurs mémoire à l'aide des *sanitizers*.
— Écriture d'une librairie de plusieurs fichiers et son utilisation dans un programme.

2 Énoncé

Il s'agit d'écrire une librairie de géométrie permettant de construire des *polygones réguliers* à n côtés ($n \geq 3$). Un polygone régulier est découpé en n triangles qui ont tous comme sommet commun le centre du polygone (voir fig. 1). Chaque triangle est formé du centre et de deux sommets consécutifs du polygone.

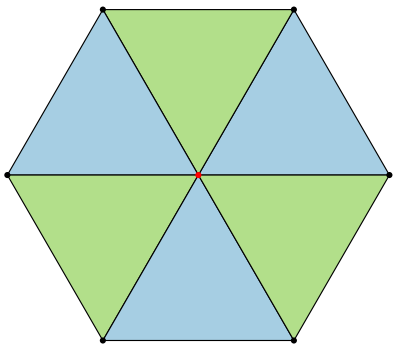


FIGURE 1 – Un hexagone régulier découpé en six triangles. Le centre (en rouge) est partagé par les six triangles, chaque sommet (en noir) par deux triangles.

Les points (le centre et les sommets) sont alloués dynamiquement **une seule fois** et sont **partagés** par les triangles qui les utilisent: les triangles ne contiennent pas de copies des points, mais des *pointeurs* vers ceux-ci. Ainsi, si on déplace le centre du polygone, les n triangles sont modifiés d'un seul coup (voir fig. 2).

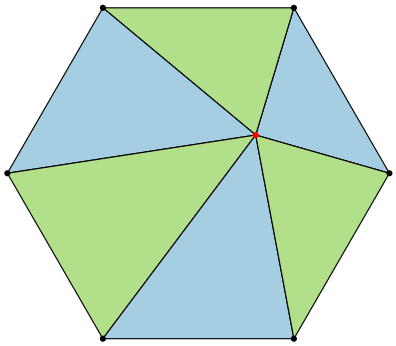


FIGURE 2 – Le même hexagone après avoir déplacé le centre de $(0.3, 0.2)$. Les six triangles ont changé, mais pas le polygone.

Ce partage pose une question délicate: lorsqu'on détruit un triangle, doit-on libérer ses points? Si oui, le centre sera libéré n fois (ce qui est une erreur). Si non, qui s'en charge? Pour résoudre ce problème, vous allez implémenter un **comptage de références**.

Ensuite, il faudra écrire un programme utilisant la librairie et permettant de vérifier que votre code se comporte comme attendu, aussi bien du point de vue des calculs que de la gestion de la mémoire.

2.1 Le comptage de références

Chaque point contient, en plus de ses coordonnées, un compteur `refcount` qui indique combien de pointeurs *détiennent* ce point. La règle est la suivante:

> Chaque endroit qui stocke un pointeur vers un point (une variable, un triangle, ...) détient **une référence** sur ce point. Lorsqu'il n'en a plus besoin, il **retire** sa référence. Le point est libéré lorsque plus personne ne le détient, c'est-à-dire lorsque son compteur tombe à zéro.

Concrètement:

- À sa création, un point a un compteur valant `1`: la référence appartient à celui qui l'a créé.
- *Ajouter* une référence sur un point (`point_incref()`) incrémente son compteur.
- *Retirer* une référence (`point_decref()`) décrémente son compteur et libère le point si le compteur atteint `0`. Retirer une référence ne libère donc **pas** forcément le point!
- On n'appelle **jamais** `free()` directement sur un point en dehors de `point_decref()`.

Par exemple:

```c
struct point *p = point_create(0.0, 0.0); // refcount: 1 (p)
struct point *q = point_create(1.0, 0.0); // refcount: 1 (q)
struct point *r = point_create(0.0, 1.0); // refcount: 1 (r)
struct triangle *t = triangle_create(p, q, r);
                     // p, q, r: refcount 2 (variable et triangle)
point_decref(&p);     // refcount: 1 (t seulement), p vaut NULL
point_decref(&q);
point_decref(&r);
triangle_destroy(&t);  // refcount: 0, les trois points sont libérés
```

3 Cahier des charges

La librairie sera constituée d'un fichier d'en-tête `geometry.h` et d'un fichier d'implémentation `geometry.c`. Les structures à utiliser sont les suivantes:

```c
#define POLYGON_MAX_SIDES 100

struct point {
    double x, y;
    int refcount; // nombre de références détenues sur ce point
};

struct triangle {
    struct point *a, *b, *c;
};

struct polygon {
    int n; // nombre de côtés (et donc de triangles)
    struct triangle *triangles[POLYGON_MAX_SIDES];
};
```

Notez que le nombre de côtés d'un polygone est limité à `POLYGON_MAX_SIDES`: le tableau `triangles` a une taille fixe, seuls les triangles qu'il contient (et leurs points) sont alloués dynamiquement.

3.1 Les points

1. Alloue un point, initialise ses coordonnées et son compteur à `1`. Retourne `NULL` si l'allocation a échoué.
   `struct point *point_create(double x, double y);`

2. Ajoute une référence sur le point `p` (incrémente son compteur) et retourne `p`. `C  struct point *point_incref(struct point *p);`
3. Retire une référence du point `*p`. La fonction reçoit l'**adresse** d'un point dynamiquement alloué puis:
   — décrémente le compteur du point;
   — libère le point si le compteur vaut `0`;
   — met la variable de l'appelant à `NULL`, **que le point ait été libéré ou non**: l'appelant a retiré sa référence, il ne doit plus utiliser ce

pointeur.

Si `p` ou `*p` vaut `NULL`, la fonction ne fait rien.

```
void point_decref(struct point **p);
```

Par exemple, si un point `q` est détenu par la variable `q` et par un triangle (compteur à 2), après l'appel `point_decref(&q);` le point existe toujours (compteur à 1, détenu par le triangle seulement), mais `q` vaut `NULL`.

4. Déplace le point `p` de (dx, dy).

   ```
   void point_move(struct point *p, double dx, double dy);
   ```

5. Retourne la distance euclidienne entre les points `p` et `q`.

   ```
   double point_distance(const struct point *p,
                         const struct point *q);
   ```

6. Affiche les coordonnées du point `p` ainsi que son compteur (p.ex. `(0.2, 0.1) [refcount = 6]`).

   ```
   void point_print(const struct point *p);
   ```

3.2 Les triangles

1. Alloue un triangle qui **ajoute une référence** sur chacun de ses trois sommets. Retourne `NULL` si l'allocation a échoué.

   ```
   struct triangle *triangle_create(struct point *a, struct point *b,
                                    struct point *c);
   ```

2. **Retire** les références sur les trois sommets du triangle `*t`, libère le triangle et met `*t` à `NULL`.

   ```
   void triangle_destroy(struct triangle **t);
   ```

3. Retourne le périmètre du triangle `t`.

   ```
   double triangle_perimeter(const struct triangle *t);
   ```

4. Retourne l'aire du triangle `t`. Pour un triangle de sommets a, b et c, l'aire vaut

   $$A = \frac{1}{2} \left| (x_b - x_a)(y_c - y_a) - (x_c - x_a)(y_b - y_a) \right| . \tag{1}$$

   ```
   double triangle_area(const struct triangle *t);
   ```

5. Retourne un **nouveau** point situé au centre de gravité (centroïde) du triangle `t`,

   $$G = \left(\frac{x_a + x_b + x_c}{3}, \frac{y_a + y_b + y_c}{3} \right) . \tag{2}$$

   ```
   struct point *triangle_centroid(const struct triangle *t);
   ```

   Par exemple:

3.3 Les polygones réguliers

1. Crée un polygone régulier à `n` côtés, de centre (c_x, c_y), dont les sommets se trouvent sur un cercle de rayon `radius`. La fonction retourne `NULL` si `n` n'est pas compris entre 3 et `POLYGON_MAX_SIDES`, si le rayon n'est pas strictement positif, ou si une allocation a échoué (dans ce dernier cas, toute la mémoire déjà allouée doit être libérée).

   ```
   struct polygon *polygon_create(int n, double cx, double cy,
                                  double radius);
   ```

   La construction se fait en trois étapes.

   a. **Créer les points.** Créer le centre et les `n` sommets. Le k-ème sommet $(k = 0, ..., n-1)$ a pour coordonnées

   $$v_k = \left(c_x + r \cos\left(\frac{2\pi k}{n}\right), c_y + r \sin\left(\frac{2\pi k}{n}\right) \right). \tag{3}$$

   Les sommets peuvent être stockés temporairement dans un tableau local `struct point *vertices[POLYGON_MAX_SIDES]`. À ce stade, chaque point a un compteur valant 1: la référence est détenue par les variables locales de la fonction.

   b. **Créer les triangles.** Le k-ème triangle est formé, **dans cet ordre**, du centre, du sommet v_k et du sommet suivant v_{k+1} (le dernier triangle relie v_{n-1} à v_0) (**attention:** l'ordre est important). Chaque triangle ajoute une référence sur ses trois points: le centre a alors un compteur valant `n + 1`, et chaque sommet, qui appartient à deux triangles, un compteur valant 3.

   c. **Retirer les références locales.** Les variables locales disparaissent à la fin de la fonction. Les points ne sont alors plus détenus que par les triangles: le centre a un compteur valant `n` et chaque sommet un compteur valant 2.

2. Détruit tous les triangles du polygone `*p`, libère le polygone et met `*p` à `NULL`. Grâce au comptage de références, tous les points doivent alors être libérés exactement une fois.

   ```
   void polygon_destroy(struct polygon **p);
   ```

3. Retourne le périmètre du polygone `p`, c'est-à-dire la somme des côtés **extérieurs** des triangles (le côté b–c de chaque triangle).

   ```
   double polygon_perimeter(const struct polygon *p);
   ```

4. Retourne l'aire du polygone `p`, c'est-à-dire la somme des aires de ses triangles.

   ```
   double polygon_area(const struct polygon *p);
   ```

5. Retourne un pointeur vers le centre du polygone `p`. Ce pointeur est **emprunté**: aucune référence n'est ajoutée, il ne faut donc pas la retirer, et il devient invalide dès que le polygone est détruit [1].

1. Si on souhaite conserver le centre après la destruction du polygone, il faut ajouter une référence avec `point_incref()` (puis la retirer avec `point_decref()` lorsqu'on n'en a plus besoin).

```
struct point *polygon_center(const struct polygon *p);
```

6. Exporte le polygone `p` dans le fichier `filename` au format SVG (voir plus bas). Retourne `false` si le fichier n'a pas pu être écrit, `true` sinon.
```
bool polygon_to_svg(const char *filename,
                    const struct polygon *p);
```

Pour vérifier vos calculs, le périmètre et l'aire d'un polygone régulier à n côtés inscrit dans un cercle de rayon r valent

$$P = 2nr \sin\left(\frac{\pi}{n}\right), \tag{4}$$

$$A = \frac{nr^2}{2} \sin\left(\frac{2\pi}{n}\right). \tag{5}$$

3.4 L'export SVG

Le format SVG (*Scalable Vector Graphics*) est un format d'image vectorielle textuel qui peut être ouvert dans n'importe quel navigateur web. Afin de pouvoir visualiser vos polygones, une petite librairie d'export vous est fournie: svg.h et svg.c. Elle ne dépend pas de vos structures et ne manipule que des coordonnées:

```
// Ouvre le fichier et y écrit l'en-tête SVG. La zone dessinée est
// le rectangle [xmin, xmax] x [ymin, ymax].
// Retourne NULL en cas d'erreur.
FILE *svg_open(const char *filename, double xmin, double ymin,
               double xmax, double ymax);
// Dessine un triangle de couleur fill (p.ex. "lightblue").
void svg_triangle(FILE *f, double ax, double ay, double bx, double by,
                  double cx, double cy, const char *fill);
// Dessine un point de couleur color (p.ex. "red").
void svg_point(FILE *f, double x, double y, const char *color);
// Termine le fichier SVG et le ferme.
void svg_close(FILE *f);
```

L'axe y est orienté vers le haut, comme en mathématiques. Pour écrire `polygon_to_svg()`, il faut:

1. Calculer la boîte englobante du polygone (les coordonnées minimales et maximales de tous ses points) et lui ajouter une marge (p.ex. 10% de sa taille) pour que le dessin ne touche pas les bords.
2. Ouvrir le fichier avec `svg_open()`.
3. Dessiner chaque triangle avec `svg_triangle()` (en alternant deux couleurs par exemple), puis les sommets et le centre avec `svg_point()`.
4. Fermer le fichier avec `svg_close()`.

Par exemple, le code suivant dessine le triangle de sommets $(0,0)$, $(2,0)$ et $(0,1)$ (voir fig. 3). Sa boîte englobante est $[0,2] \times [0,1]$. Avec une marge de 10% de la plus grande dimension, soit 0.2, la zone dessinée est $[-0.2, 2.2] \times [-0.2, 1.2]$.

```c
FILE *f = svg_open("exemple.svg", -0.2, -0.2, 2.2, 1.2);
if (f == NULL) {
    fprintf(stderr, "Impossible d'ouvrir exemple.svg\n");
    return EXIT_FAILURE;
}
svg_triangle(f, 0.0, 0.0, 2.0, 0.0, 0.0, 1.0, "lightblue");
svg_point(f, 0.0, 0.0, "red");
svg_point(f, 2.0, 0.0, "black");
svg_point(f, 0.0, 1.0, "black");
svg_close(f);
```

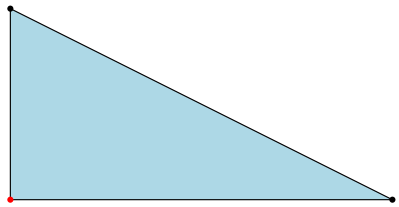


FIGURE 3 – Le fichier `exemple.svg` produit par le code ci-dessus. Le point $(0, 0)$, en rouge, est bien en bas à gauche.

Cet exemple n'est pas général et ne marche que pour un triangle. Il faut réfléchir un peu pour l'étendre au cas plus général du polygone.

3.5 Le programme principal

Votre programme prend quatre arguments à la ligne de commande: le nombre de côtés `n`, le rayon `r` du polygone, et un déplacement `dx dy` à appliquer au centre. Il doit:

1. Vérifier les arguments et afficher un message d'erreur sur `stderr` (avec un code de retour non nul) s'ils ne sont pas valides.
2. Créer le polygone centré en $(0, 0)$, afficher son centre (avec son compteur), son périmètre, son aire et les valeurs théoriques correspondantes, puis le périmètre P, l'aire A et le centroïde G de chaque triangle. L'exporter dans le fichier `avant.svg`.
3. Déplacer le centre du polygone de (dx, dy), afficher à nouveau les mêmes informations, et exporter le polygone dans le fichier `apres.svg`.
4. Détruire le polygone.

Par exemple:

```
$ ./geometrie 6 1 0.2 0.1
Polygone régulier à 6 côtés, rayon 1
Centre: (0, 0) [refcount = 6]
Périmètre: 6.000000000 (théorique: 6.000000000)
Aire:      2.598076211 (théorique: 2.598076211)
  triangle  0: P = 3.000000, A = 0.433013, G = (0.5, 0.288675)
```

```
  triangle  1: P = 3.000000, A = 0.433013, G = (1.11022e-16, 0.57735)
  ...

Déplacement du centre de (0.2, 0.1)
Centre: (0.2, 0.1) [refcount = 6]
Périmètre: 6.000000000 (théorique: 6.000000000)
Aire:      2.598076211 (théorique: 2.598076211)
  triangle  0: P = 2.628901, A = 0.321410, G = (0.566667, 0.322008)
  triangle  1: P = 2.860363, A = 0.383013, G = (0.0666667, 0.610684)
  ...
```

4 Vérification de la mémoire: les *sanitizers*

Les erreurs de gestion de la mémoire (fuites, double libération, utilisation d'une zone déjà libérée) ne provoquent pas forcément d'erreur l'exécution: le programme peut sembler fonctionner parfaitement. De même on peut avoir des erreurs incompréhensibles au premier abord (SegFault ou autre). Pour détecter et aider à remonter à la source du problème, on utilise les *sanitizers* du compilateur. Il suffit de compiler **et** de faire l'édition des liens avec les options suivantes (ici sous la forme de variables d'un `Makefile`):

```
-std=c11 -Wall -Wextra -pedantic -g -fsanitize=address,undefined # compilation
-fsanitize=address,undefined -lm # édition des liens
```

Attention: si vous n'utilisez pas `-g` il sera impossible de localiser la source des erreurs (cela aide à afficher les lignes où les erreurs se produisent).