TD4 : Classe et objet (2/2)

Exercice 1 : Constructeur et destructeur en mémoire

Soit le programme C++ suivant :

```
#include <iostream>
using namespace std;

class Point2D {
  public:
    float x,y;
    Point2D () {x = 0.0; y = 0.0; cout << "Point2D nul créé\n";}
    Point2D (float _x, float _y) {
      x = _x; y = _y;
      cout << "Point2D créé\n";
    }
    Point2D (const Point2D & p) {
      x = p.x; y = p.y;
      cout << "Point2D copié\n";
    }
    ~Point2D () {cout << "Point2D détruit\n";}
};

int main() {
  Point2D pt1;
  Point2D pt2 (1.0,2.5);
  Point2D * ppt3 = new Point2D (pt2);
  delete ppt3;
  return 0;
}
```

a. Dessiner l'état de la mémoire après chaque construction d'objet en supposant que la valeur de retour du main est stockée à l'adresse 3 987 546 988 et qu'il n'y a pas de problème d'allocation mémoire.

Après la construction de pt1 (constructeur sans paramètre) :

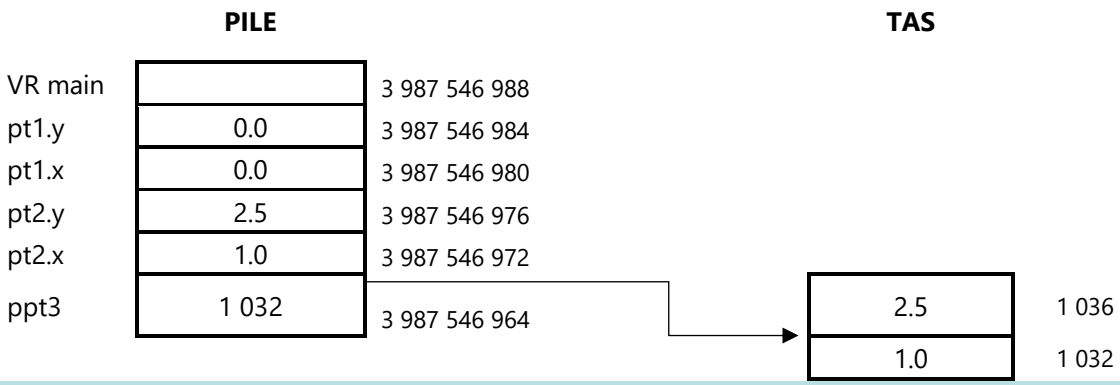
	PILE		TAS
VR main		3 987 546 988	
pt1.y	0.0	3 987 546 984	
pt1.x	0.0	3 987 546 980	

Après la construction de pt2 (constructeur avec paramètres x,y) :

	PILE		TAS
VR main		3 987 546 988	
pt1.y	0.0	3 987 546 984	
pt1.x	0.0	3 987 546 980	

pt2.y	2.5	3 987 546 976
pt2.x	1.0	3 987 546 972

Après la construction de ppt3 (constructeur par copie sur le tas) :



b. Donner la trace écran de l'exécution de ce programme.

```
Point2D nul créé              (pt1)
Point2D créé                  (pt2)
Point2D copié                 (ppt3)
Point2D détruit               (ppt3)
Point2D détruit               (pt2)
Point2D détruit               (pt1)
```

c. Au lieu de définir un constructeur sans paramètre et un autre avec comme paramètres les données membres, vous pouvez utiliser des valeurs par défaut. Ecrivez un constructeur avec paramètres par défaut pouvant remplacer les deux autres.

```
Point2D (float _x = 0.0, float _y = 0.0) {
  x = _x; y = _y;
  cout << "Point2D créé\n";
}
```

Exercice 2 : Appel à des fonctions membres

Soit le programme C++ suivant :

```
#include <iostream>
using namespace std;

class Point2D {
  public:
    float x,y;

    Point2D (float _x, float _y) {x = _x; y = _y;}
    ~Point2D () {}

    float distanceOrigine() const {return sqrt(x*x+y*y);}
    float distancePoint (const Point2D & p) const {
       return sqrt((p.x-x)*(p.x-x)+(p.y-y)*(p.y-y));
    }
```

```
}

int main() {
  Point2D pt1 (2.6,7.5);
  Point2D pt2 (1.4,3.8);
  float distpt1O;
  float distpt2pt1;
  distpt1O = pt1.distanceOrigine();
  distpt2pt1 = pt2.distancePoint(pt1);
  return 0;
}
```

a. Dessiner l'état de la mémoire avant les suppressions de frame (sauf constructeurs) en supposant que la valeur de retour du main est stockée à l'adresse 3 987 546 988.

Avant la sortie de pt1.distanceOrigine() :

	PILE	**TAS**
VR main		3 987 546 988
pt1.y	7.5	3 987 546 984
pt1.x	2.6	3 987 546 980
pt2.y	3.8	3 987 546 976
pt2.x	1.4	3 987 546 972
distpt1O		3 987 546 968
distpt2pt1		3 987 546 964
	appel à pt1.distanceOrigine	
VR float	7.937883849	3 987 546 960
this	3 987 546 980	3 987 546 952

Avant la sortie de pt2.distancePoint(pt1) :

	PILE	**TAS**
VR main		3 987 546 988
pt1.y	7.5	3 987 546 984
pt1.x	2.6	3 987 546 980
pt2.y	3.8	3 987 546 976
pt2.x	1.4	3 987 546 972
distpt1O	7.937883849	3 987 546 968
distpt2pt1		3 987 546 964
	appel à pt2.distancePoint	
VR float	3.889730068	3 987 546 960
this	3 987 546 972	3 987 546 952
p	3 987 546 980	3 987 546 944

Avant la sortie du main :

	PILE	TAS
VR main	0	3 987 546 988
pt1.y	7.5	3 987 546 984
pt1.x	2.6	3 987 546 980
pt2.y	3.8	3 987 546 976
pt2.x	1.4	3 987 546 972
distpt1O	7.937883849	3 987 546 968
distpt2pt1	3.889730068	3 987 546 964

b. La fonction distanceOrigine n'est finalement qu'un cas particulier de distancePoint. Réécrivez le code de distanceOrigine afin d'utiliser le code de distancePoint.

```
float distanceOrigine() const {
  Point2D origine (0.0,0.0);
  return distancePoint(origine);   // ou bien : origine.distancePoint(*this)
}
```

Cette fonction peut aussi s'écrire en une instruction : `return distancePoint(Point2D(0.0,0.0));`