# INF1600 : TP4

## Assembleur et C++

Imane Hafnaoui <imane.hafnaoui@polymtl.ca>
Giovanni Beltrame <giovanni.beltrame@polymtl.ca>

École Polytechnique de Montréal

Automne 2014

# Introduction

Dans ce quatrième travail pratique d'INF1600, nous vous demandons à nouveau de faire valoir vos compétences en assembleur. Cette fois, vous aurez à manipuler des objets du langage C++ à l'aide du langage assembleur afin de comprendre comment procède C++ pour implémenter les principes du paradigme orienté-objet. Assurez-vous de bien comprendre les concepts d'héritage et de polymorphisme avant d'entreprendre ce travail. Les notes de cours au sujet de l'assembleur vous seront très utiles.

# Remise

Voici les détails concernant la remise de ce travail pratique :

- **méthode** : sur Moodle (une seule remise par équipe) ;
- **échéance** : avant 23h50, le 23 novembre 2014 pour les groupes du Lundi B1, le 30 novembre 2014 pour les groupes du Lundi B2.
- **format** : le fichier `tp4_s.s`
- **langue écrite** : français, English ;
- **distribution** : les deux membres de l'équipe recevront la même note.

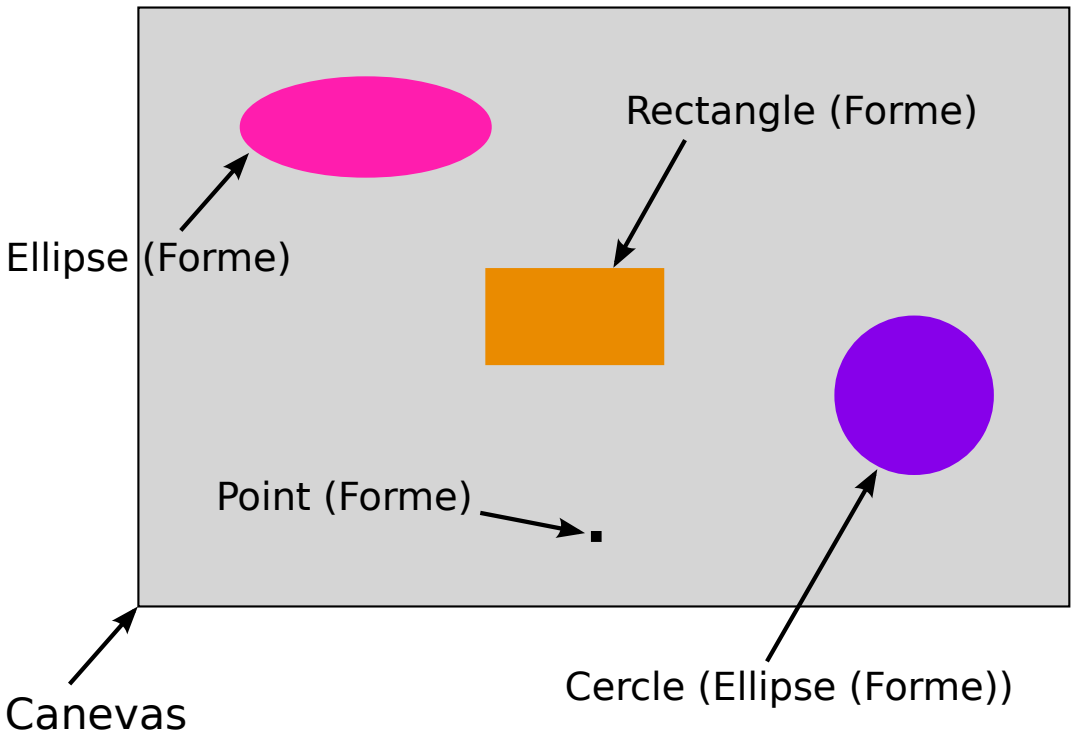
# Barème

| ***Contenu*** | ***Points du cours*** |
|---|---|
| Bonne implementation de `nouveauCercle` | 1 |
| Bonne implementation de `faireDessin` | 2 |
| Bonne implementation de `desallouerForme` | 1 |
| Bonne implémentation de `Canevas::dessinerForme` | 1 |
| Bonne implémentation de `Canevas::assignerPixel(x, y)` | 2 |
| Bonne implémentation de `Canevas::assignerPixel(Coord)` | 1 |
| Allure générale du code (clarté, commentaires, choix des noms de variables/étiquettes, etc.) | 2 |
| Langage écrit erroné (dans les commentaires) | jusqu'à -1 |
| Format de remise erroné (irrespect du noms de fichiers demandé, fichiers superflus, etc.) | jusqu'à -1 |
| Retard | -1 par jour |

## Présentation

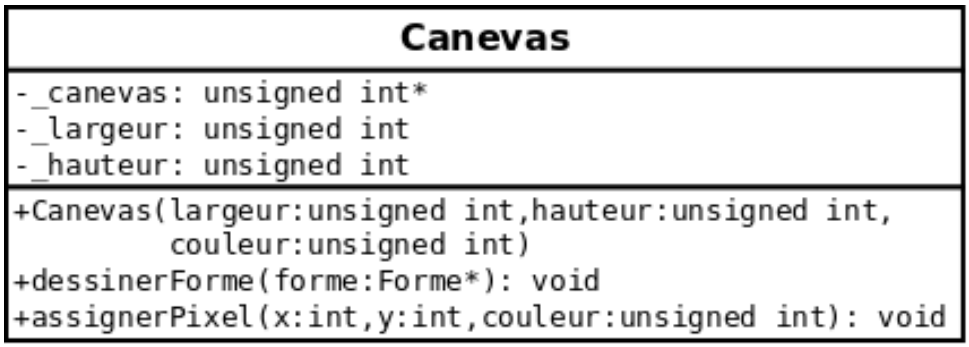
Le code qui vous est fourni dans `inf1600_tp4.zip` implémente un simple engin de rendu graphique permettant de dessiner des formes géométriques sur un canevas. Cependant, certaines fonctions ou méthodes importantes sont manquantes et vous aurez bien sûr à les implémenter. Une présentation non-exhaustive des classes présentes et de leurs propriétés est faite dans la présente section, mais il est conseillé de se référer aux fichiers sources où toutes les méthodes et attributs sont documentés.

Pour résumer, le programme que vous allez réaliser permet de créer une image, et d'y dessiner des points, des lignes, des cercles, etc. Le code déjà présent dans l'archive permet de produire une animation en affichant une suite d'images à l'écran. **Lorsque vous aurez implémenté les fonctions manquantes, vous pourrez voir cette animation** – et pourquoi pas créer la vôtre.

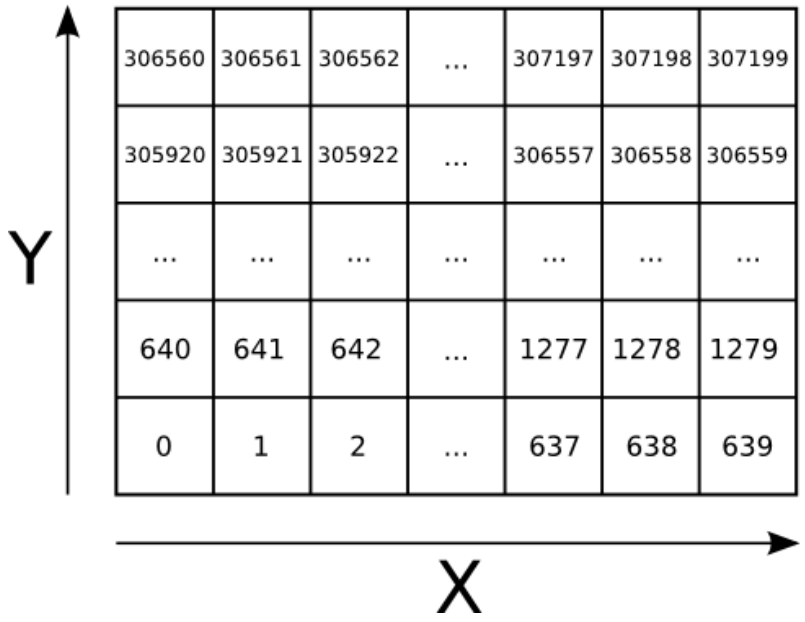


### *Classe Canevas*

Un canevas représente une image. Il peut être vu comme un tableau de points en deux dimensions où chaque point possède une valeur de couleur. La classe `Canevas` représente donc un canevas sur lequel on peut dessiner en assignant des valeurs de couleurs aux points (pixels). Chaque pixel est représenté par un entier non signé (`unsigned int`) dont les trois derniers octets contiennent respectivement les composantes rouges, vertes et bleues de la couleur. L'octet de poids fort est quant à lui inutilisé.

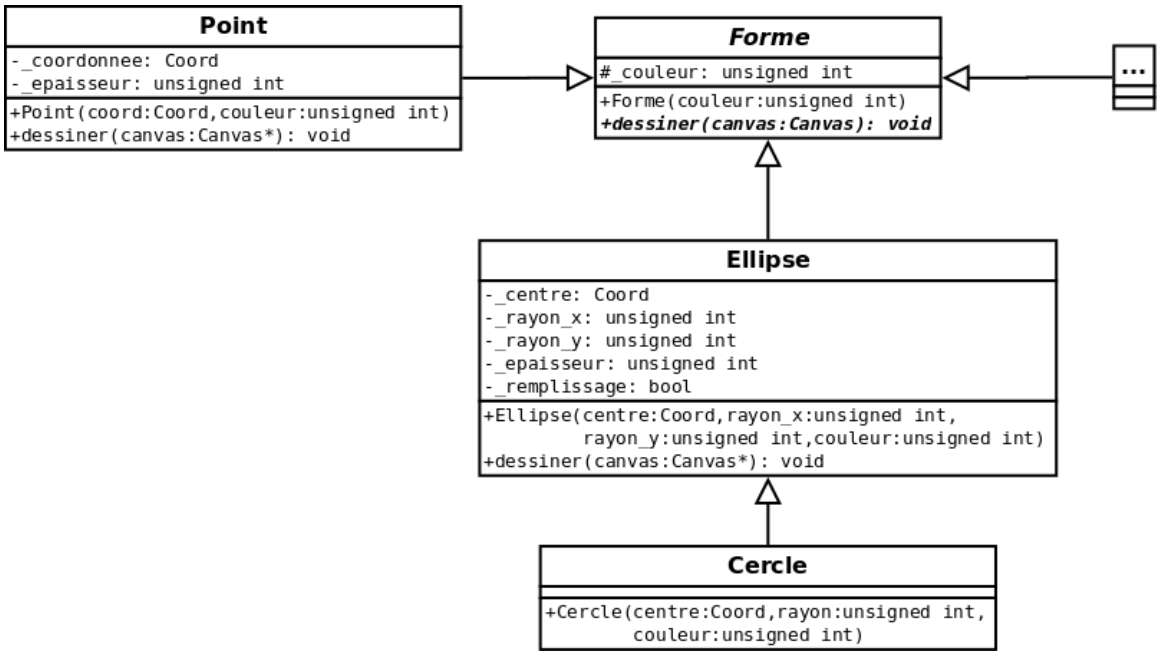



Le constructeur de `Canevas` prend en paramètres la hauteur et la largeur désirée et alloue un espace mémoire en conséquence pour entreposer l'ensemble des pixels. La convention utilisée ici est de placer le pixel de coordonnée `(x, y)` = `(0, 0)` en bas à gauche du canevas. Puis, les rangées de pixels sont concatenées dans le tableau `_canevas` de la plus basse jusqu'à la plus haute. Les index du tableau `_canevas` sont donc reliés aux pixels comme illustré sur la figure suivante, en supposant un canevas de 640 pixels de largeur par 480 pixels de hauteur). Le troisième paramètre du constructeur permet de spécifier la couleur avec laquelle seront initialisés tous les pixels du canevas.



### *Classe abstraite Forme et classes dérivées*

Une multitude de classes représentant des formes géométriques ont été implémentées. Toutes héritent de la même classe de base abstraite `Forme`. L'image suivante représente trois sortes de `Forme`s concrètes (`Point`, `Ellipse` et `Cercle`), leurs attributs particuliers ainsi que leur relation avec la classe `Forme`. On remarque aussi que la class `Cercle` n'est qu'un cas particulier de l'`Ellipse`, puisqu'il s'agit d'une `Ellipse` dont les deux axes sont de même longueur.




La classe `Forme` possède une méthode virtuelle pure dessiner qui prend en paramètre un pointeur vers un objet `Canevas` sur lequel tracer les éléments constituant la forme. Chaque forme implémente donc cette méthode comme bon lui semble afin d'assigner les bons pixels du canevas à la bonne couleur pour dessiner le résultat attendu. De plus, la classe `Canevas` présentée plus haut propose aussi une méthode `dessinerForme` qui prend en paramètre un pointeur vers une `Forme` quelconque. Cette méthode ne fera qu'appeler par polymorphisme la méthode dessiner correspondante en lui passant l'adresse du canevas.

Notez que la structure `Coord` qui se retrouve dans les différentes formes représente une coordonnée en `x` et en `y`.

Après avoir dessiné différentes formes sur le canevas, il est possible par exemple d'exporter l'image dans un fichier afin de la garder en souvenir. Dans ce TP, nous avons choisi de créer une animation en faisant appel à l'engin de rendu plusieurs fois par seconde (en modifiant légèrement les propriétés des formes entre chaque rendu) et en affichant le résultat de chaque génération directement à l'écran. Vous n'avez pas à vous soucier de cette partie, mais vous pourrez tout de même apprécier l'animation résultante une fois que vous aurez complété le travail demandé.

## Travail demandé

On vous demande d'implémenter en langage assembleur IA-32 les méthodes suivantes.

### *Cercle* nouveauCercle(Coord centre, unsigned int rayon, unsigned int couleur);*

Cette fonction permet de créer une forme de type `Cercle` de façon dynamique. Sont passés à la fonction les coordonnées du centre du cercle, son rayon ainsi que sa couleur. Votre fonction doit donc allouer de façon dynamique un objet `Cercle` ayant ces propriétés et retourner l'adresse de l'objet nouvellement créé. Vous pouvez vous inspirer des fonctions analogues à celle-ci présentes dans le fichier `main.cpp`.

### *void faireDessin(Forme** formes, unsigned int n);*

La fonction `faireDessin` prend en paramètres un tableau de pointeurs de formes ainsi que le nombre d'éléments dans ce tableau. La tâche de cette fonction est de créer un objet `Canevas` et d'y appliquer toutes les formes qui sont fournies à l'aide de la méthode `Canevas::dessinerForme`. Les définitions de la taille du canevas et de la couleur par défaut se trouvent dans le fichier `tp4.hpp`. Par la suite, vous devez appeler la fonction `ecrireFrame` en lui passant un pointeur vers votre canevas. La fonction `ecrireFrame` est déjà implémentée et permettra de copier les pixels du canevas dans un tampon d'OpenGL afin de les afficher à l'écran.

### *void desallouerForme(Forme* forme);*

Cette fonction prend en paramètre un pointeur vers une `Forme` quelconque. Elle doit tout simplement détruire la forme en question et libérer la mémoire associée.

### *void Canevas::dessinerForme(Forme* forme);*

Cette méthode de la classe `Canevas` reçoit en paramètre un pointeur vers une forme quelconque. Sa tâche est de faire appel à la méthode dessiner de la forme en lui passant en paramètre un pointeur vers le `Canevas` courant (le fameux pointeur `this`). Il s'agit de la méthode mentionnée plus haut dans la section de présentation.

### *void Canevas::assignerPixel(int x, int y, unsigned int couleur);*

Cette méthode permet d'assigner une valeur de couleur à une pixel du canevas. Votre fonction devra d'abord vérifier que la position `(x,y)` passée en paramètre se trouve à l'intérieur du canevas, puis devra déduire l'index du tableau `_canevas` correspondant à cette coordonnée pour enfin aller placer la valeur de la couleur spécifiée au bon endroit. Si la coordonnée

spécifiée en paramètre n'est pas valide (en dehors du canevas), aucune modification n'est apportée au canevas.

### *void Canevas::assignerPixel(Coord coord, unsigned int couleur);*

Cette méthode est équivalente à la dernière, mais peut être plus pratique dans certaines situations puisqu'elle reçoit un objet `Coord` pour définir la coordonnée plutôt que deux entiers séparés. Pour l'implémenter, vous pouvez soit exécuter la même logique que la méthode précédente ou bien tout simplement faire un appel à la méthode précédente en lui passant les coordonnées de façon individuelles.

## Méthode de travail

Il est conseillé d'implémenter vos fonctions en C++ avant tout afin de vous assurer que vous comprenez bien la logique du programme. Par la suite, vous pouvez traduire une fonction à la fois vers l'assembleur. Commentez simplement votre fonction C++ au même moment où vous travaillez sur la version en assembleur. Les fonctions (et méthodes) implémentées en C++ doivent être placées dans le fichier `tp4_cpp.cpp` alors que celles en assembleur doivent se retrouver dans `tp4_s.S`. **Seul votre code est assembleur sera évalué.**

La seule commande nécessaire à la compilation est `make`. Pour lancer votre programme compilé, utilisez `./tp4`.

Il est interdit d'utiliser des variables globales (section `.data`).

Conseils :

- Vous aurez besoin de trouver quelques informations à propos des classes par vous même, donc n'hésitez pas à consulter directement les fichiers sources de celles-ci.
- N'oubliez pas que tout objet créé localement dans une fonction (donc sur la pile) doit être détruit convenablement à la fin de la fonction afin de libérer de la mémoire qui lui est potentiellement associée.

Si vous voulez travailler sur vos ordinateurs personnels, vous aurez besoin d'installer la librairie freeglut. Sous Debian/Ubuntu :

```
$ sudo apt-get install libstdc++6:i386 freeglut3-dev freeglut3-dev:i386
```

Sous Fedora :

```
$ sudo yum install gcc-c++ libstdc++.i686 freeglut-devel.i686 mesa-
libGL-devel.i686 mesa-libGLU-devel.i686
```

# Rappels techniques

## *Décoration des noms de fonction (mangling)*

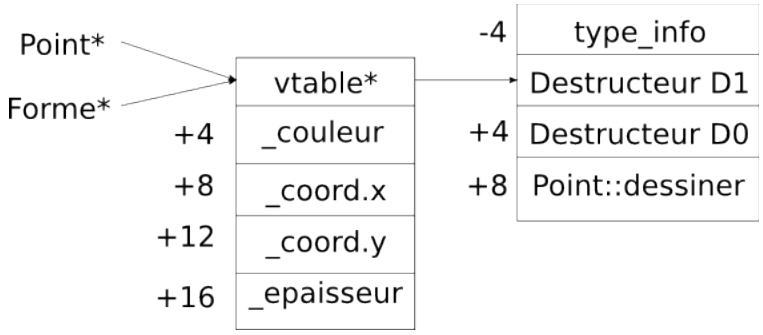
En langage C, lorsqu'une fonction est déclarée, un symbole portant le même nom que la fonction est généré. Il est donc impossible d'effectuer de la surcharge de fonctions puisque deux fonctions avec le même nom généreraient le même nom de symbole. En C++, il est toutefois possible de définir deux fonctions de même nom mais qui prennent des arguments différents. Ceci est obtenu en générant le nom du symbole en se basant sur le nom de la fonction ainsi que ses arguments. Référez-vous aux notes de cours afin de déduire le nom décoré des fonctions que vous devez implémenter.

L'outil `c++filt` peut vous aider à vérifier la validité de vos noms décorés en vous redonnant le nom original de la fonction :

```
$ c++filt _ZN5Ligne17assignerEpaisseurEj
Ligne::assignerEpaisseur(unsigned int)
```

## *Table de méthodes virtuelles (vtable)*

Un des points clés de ce TP est l'utilisation des tables de méthodes virtuelles, ou vtable. L'objectif n'est pas ici d'expliquer le principe de la vtable, mais de clarifier quelque peu sa structure. Voici donc un exemple de la structure d'un objet `Point`.




Le premier élément de l'objet `Point` est le pointeur vers la vtable. Dans cette table, on retrouve d'abord un pointeur vers le destructeur `D1` de `Point` (`_ZN5PointD1Ev`) qui exécute simplement le code que nous avons défini dans le destructeur en C++. Puis se trouve un pointeur vers le destructeur `D0` (`_ZN5PointD0Ev`), qui lui effectue le même travail que `D1`, mais en plus fait un appel à l'opérateur delete pour désallouer l'objet de façon dynamique. Ce destructeur est donc pratique à utiliser au lieu de faire un appel au destructeur suivi d'un appel à l'opérateur `delete`. On trouve finalement les différentes méthodes virtuelles de l'objet, dans notre cas `dessiner` étant la seule.