PoP C++ Série 4

H1 :
a) Usage de static à l'échelle d'un module
b) Elaboration d'un type paramétré

H2 : MOOC MOOC Introduction à la programmation orientée objet (en C++)

Série semaine 3: Surcharge des opérateurs

Dans document séparé ; l'exercice 11 n'est pas au programme du cours

---

Première partie :

a) static à l'échelle d'un module

Exercice 1.1(niveau 0) : **module student gérant un ensemble de variables Student.**

Le document séparé illustre l'usage de **static** pour un type concret mis en œuvre avec une structure.

Exercice 1.2 (niveau 2) : gestion d'un agenda

On testera les modules **date** et **event** au fur et à mesure des questions avec un module de niveau supérieur **test** qui inclut leur interface et appelle les fonctions exportées.

a) Méthode de travail : Commencer par écrire le fichier **makefile** et les **fichiers .cc et .h dans une version minimale** qui compile pour produire l'exécutable **test**, avant de préciser les classes **date** et **event** avec les questions suivantes. On mettra en oeuvre les header guard pour les interfaces des modules (série3 PoP).

b) module **date** gérant une classe **Date** de calendrier
Ecrire un module responsable d'une classe Date avec 3 attributs privés : **jour, mois, annee.**

Un seul constructeur sera défini pour initialiser simultanément les 3 champs en vérifiant que la date est correcte pour le calendrier grégorien (la série1 fourni le code source qui effectue ces vérifications). Un message est affiché en cas d'erreur.

c) Une méthode publique de calcul du *nombre de jours entre deux dates* sera aussi fournie (cf série1).

d) Ecrire une surcharge des opérateurs == et != pour la classe **Date**

e) module **event** gérant une classe **Event** et un ensemble d'instance de ce type dans un tableau dynamique **agenda** restant confidentiel au niveau du module event.
La classe Event contient les attributs privés : **nom** et **lieu** de type string, et **date** de type Date.

Un constructeur doit permettre d'initialiser une instance à partir de la valeur de tous les attributs.

f) Ecrire une surcharge des opérateurs == et != pour la classe **Event**

g) Une méthode publique **add_to_agenda** doit permettre d'ajouter la valeur d'une variable de type Event à l'ensemble des Event mémorisés dans **agenda** au niveau du module **event**. Les conditions à remplir sont les suivantes :
- L'agenda ne doit pas contenir déjà cet **Event**
- il ne doit pas y avoir d'Event déjà prévu pour le même lieu à la même *date plus ou moins 6 jours*
Un message d'erreur est affiché en cas d'erreur.

h) Une méthode publique doit permettre d'afficher l'Event sur lequel elle est appelée.
Une méthode de classe doit permettre d'afficher l'ensemble des Event mémorisés dans l'agenda.

Exercice 1.3 (niveau 2) : Gestion d'une famille d'entités avec un type paramétré

Le concept de type paramétré sert à intégrer la gestion d'entités d'une même famille au sein d'une seule classe. Ce concept de type paramétré est destiné à être remplacé par la notion de *hiérarchie de classe* présentée la semaine prochaine mais dont le plein potentiel sera réalisé seulement avec le concept du *Polymorphisme* présenté dans deux semaines. L'exercice sera à nouveau abordé avec ces concepts plus avancés à ce moment là.

Cet exercice vise également à aborder la décomposition d'une tâche à plusieurs niveaux d'abstraction dans une architecture modulaire. Il s'agit de la lecture de fichier de configuration introduite dans la série0 de la semaine précédente.

a) L'architecture du programme est la suivante*, du plus bas niveau vers le plus haut niveau* (Fig1):
   - Module **form**: définit la classe **Form** avec un type paramétré permettant de gérer 3 formes distinctes (cercle, carré, rectangle). Ce module offre aussi le type **Bbox** (comme *bounding box*) qui est le plus petit rectangle englobant d'une Form (Fig2).
   - Module **formset**: inclut **form.h** pour définir la classe **Formset** qui gère un ensemble de **Form** et tient à jour la **Bbox** de cet ensemble. L'ensemble de tous les **Formset** déjà créés est caché dans ce module (cf série0).
   - Module de plus haut niveau **test** : inclut **formset.h**. Contient main() ; il lit un fichier en effectuant une vérification après la lecture de chaque **Formset**. Il faut vérifier si **Bbox** du **Formset** intersecte l'une des **Bbox** des autres **Formset** de l'ensemble caché dans le module formset.cc. Si c'est le cas un message d'erreur est affiché ainsi que la valeur des **Bbox** des 2 **Formset**.

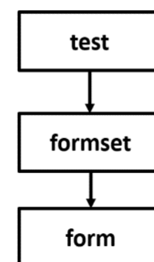

Fig 1: architecture modulaire

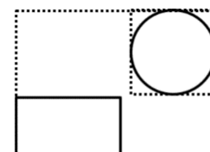
Fig 2: **Bbox** d'un ensemble de 2 **Form**

Commencer par écrire le fichier **Makefile** et les **fichiers .cc et .h** dans une version minimale avec seulement les directives **include** et une fonction **main** vide afin de produire l'exécutable **test**. On mettra en oeuvre les header guard pour les interfaces des modules (série3 PoP). On précisera les classes **Form, Formset** et le type **Bbox** à partir des questions suivantes.

b) Définition et test unitaire de la classe **Form**. Par definition d'un type paramétré la classe Form doit avoir un attribut définissant le type de la forme. Un type défini avec **enum** est parfait, comme dans le code fourni en exemple (form.h)

```
enum Category {CIRCLE, SQUARE, RECTANGLE};

class Form;         // pré-déclaration de la class Form
typedef Bbox Form; // le type Bbox est synonyme de Form

class Form
{
public:
    Form(Category c=CIRCLE,double x=0,double y=0,double p1=0);
    Form(Category c,double x,double y,double p1,double p2 );
    void affiche() ;
    …
private:
    Category category;
    double x,y ; coordonnées du centre de la Form
    std::vector<double> param{std::vector<double>(1,0.)};
};
```

La définition des méthodes doit être externalisée dans l'implémentation form.cc . On fournit déjà celle des 2 constructeurs et de la méthode affiche visibles ci-dessus. Les deux constructeurs vérifient que la valeur des paramètres est positive ou nulle, sinon un message d'erreur est affiché et on appelle exit(EXIT_FAILURE).

On aura aussi besoin au moins d'un accesseur à la catégorie d'une Form.

Surcharger l'opérateur << pour obtenir un affichage des attributs dans le terminal.

Tester les constructeurs en faisant afficher la valeur d'instances de **Form** par un programme de test.

Ecrire une fonction **form_lecture** qui accepte un parametre **istringstream** et une valeur de type **Category** et qui, selon ce second parametre, lit (x,y) et le ou les paramètres et renvoie la valeur d'une instance de **Form** initialisée avec cette catégorie et valeurs.

Tester **form_lecture** par le programme de test.

c) L'interface form.h utilise **typedef** pour établir que le nouveau type **Bbox** est synonyme de **Form**. Le nom **Bbox** vient de l'abréviation de ***bounding box*** *(boite englobante)*. Ce type doit être une **Form** de la catégorie RECTANGLE sur laquelle nous ferons des opérations spécialisées ; en effet, cette notion de boite englobante est très utilisée pour réduire de coût calcul des programmes qui dessinent des formes complexes.

Ajouter une méthode qui renvoie une **Bbox** pour chaque catégorie de **Form**. RECTANGLE renvoie simplement sa valeur courante tandis que SQUARE doit créer un RECTANGLE dont la largeur = hauteur = coté du carré. Enfin, CIRCLE doit créer un RECTANGLE dont largeur = hauteur = 2x rayon du cercle.

Tester ces méthodes par le programme de test.

Ajouter une méthode **Bbox merge_bbox(Bbox& other);** qui calculi et renvoie la plus petite **Bbox** qui contient l'instance et le paramètre **other** (fig2). On obtient les valeurs selon x et y comme suit:

- Min_merge = MIN(instance, other)
- Max_merge= MAX(instance, other)

Il faut ensuite mettre à jour le centre de la **Bbox** résultante avec la moyenne des valeurs min et max selon x et selon y.

Ajouter une méthode **bool intersect_bbox(Bbox& other);** qui renvoie un booléen true s'il y a intersection de l'instance et du paramètre **other** (renvoie false sinon). Un test possible est le suivant: soit **dx** et **dy** les distances entre les deux centres des Bbox et **cx** et **by** la moyenne de leurs cotés. Il y a intersection si (**dx < cx ET dy < cy**).

Fig 3: deux cas produisant un booléen true

Les deux méthodes merge et intersect doivent vérifier la catégorie RECTANGLE de l'instance et du paramètre avant de commencer ses calculs

Tester ces méthodes par le programme de test avec au moins 4 à 6 configurations qui produisent des familles de cas distincts (position relative, les 2 cas de Form incluse dans l'autre, les 2 Form de taille nulle).

d) Le module **formset** gère la classe **Formset** qui regroupe un ensemble de **Form** et tient à jour la **Bbox** de cet ensemble (Fig 3). Nous supposons ici que le module **Form** a été vérifié avec les tests des questions précédentes.

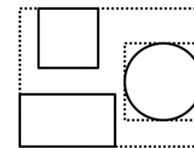

Fig 4: Bbox d'un Formset

Ecrire un constructeur par défaut puis une méthode **add** qui ajoute une **Form** au **Formset** en mettant à jour la **Bbox** (pour cette mise à jour il faut distinguer 2 cas: celui où on ajoute une **Form** à un **Formset** vide et les autres cas).

Ecrire une méthode affiche qui est utilisée par la surcharge externe de l'opérateur << pour obtenir un affichage des attributs dans le terminal. On délégue à la classe Form l'affichage de chaque Form. Puis tester dans un programme de test le constructeur et l'ajout en faisant afficher la valeur du **Formset** après chaque ajout d'une **Form**.

Ecrire une méthode **bool intersect_formset(Formset & other);** qui determine si la **Bbox** du **Formset** passé en paramètre intersecte la **Bbox** de l'instance sur laquelle on l'appelle. Renvoie un booléen *true* s'il y a intersection et faux sinon. Cette méthode délègue la vérification au module **form**.

Ecrire une fonction **formset_lecture** qui accepte en parametre un **ifstream** sur un fichier qu'on suppose déjà ouvert avec succès. Son but est de lire les lignes de ce fichier correspondant à *un seul* **Formset** avec son automate de lecture. Elle modifie un paramètre **Formset** passé par

reference. Pour cela, elle va contenir la boucle qui extrait une ligne du fichier en filtrant les séparateurs (série3 niveau0):

```
while(getline(fichier >> ws, line)
```

Le format attendu pour la lecture de la partie d'un fichier de configuration pour un **Formset** est d'avoir d'abord les 3 nombres entiers des nombres de cercle, de carrés et de rectangle (voir fichiers fournis). Ensuite, dans le même ordre, il y a autant de lignes que les nombres annoncés. Ces lignes contiennent les valeurs (x,y) suivies par un ou deux paramètres de la Form.

Selon la valeur des nombres d'éléments l'automate va faire évoluer son état entre lecture de cercle, de carré et de rectangle. La lecture d'une **Form** est déléguée à la fonction **form_lecture** du module form à qui on passe un **istreamstring** initialisé avec la string **line**. La valeur de **Form** renvoyée est ajoutée au paramètre **Formset** avec la méthode **add**.

Le test de **form_lecture** est l'objectif du module de plus haut niveau de cet exercice comme décrit à la question e).

Enfin, écrire une fonction qui ajoute un **Formset** à un **vector** de **Formset** caché dans le module **formset**.

e) Le module de plus haut niveau **test** a les responsabilités suivantes:

main() : récupère un nom de fichier de configuration passé en argument sur la ligne de commande. Elle passe une string initialisée avec ce nom de fichier à la fonction **lecture** qui se charge de l'ouvrir et de gérer la boucle de lecture des **Formset**.

lecture : ouverture du fichier pour initialiser un **ifstream** et gestion du nombre de **Formset**. Cette fonction gère seulement le plus haut niveau du décodage du fichier de configuration: la boucle de lecture des **Formset**. Cette fonction s'attend au format de fichier de configuration suivant (voir exemples fournis):

D'abord un entier indiquant le nombre de Formset
Ensuite les données de chaque Formset

Cette fonction va donc contenir un boucle d'extraction de ligne de fichier telle que:

```
while(getline(fichier >> ws, line)
```

Une fois décodée la ligne indiquant le nombre de Formset, on quitte le while ci-dessus pour gérer la boucle de lecture des **Formset** ; chaque lecture de **Formset** appelle **formset_lecture**. Après la lecture de chaque **Formset**, on autorise son ajout au vector des **Formset** caché dans le module **Formset** seulement si le nouveau **Formset** n'intersecte pas ceux qui sont déjà dans le vector caché du module **formset.** Cette verification doit être déléguée autant que possible à la classe **Formset** et au module **formset.** Faire afficher un petit message de succès pour chaque ajout de **Formset** à ce vector caché.