Langages à objets

Définitions

Traduction des méthodes en C++

Polymorphisme, affectation et héritage

Accès aux attributs

Accès aux méthodes

Tables virtuelles

Héritage multiple

Gestion d’exceptions

Définitions

Objets :

regroupent les structures de données et les opérations

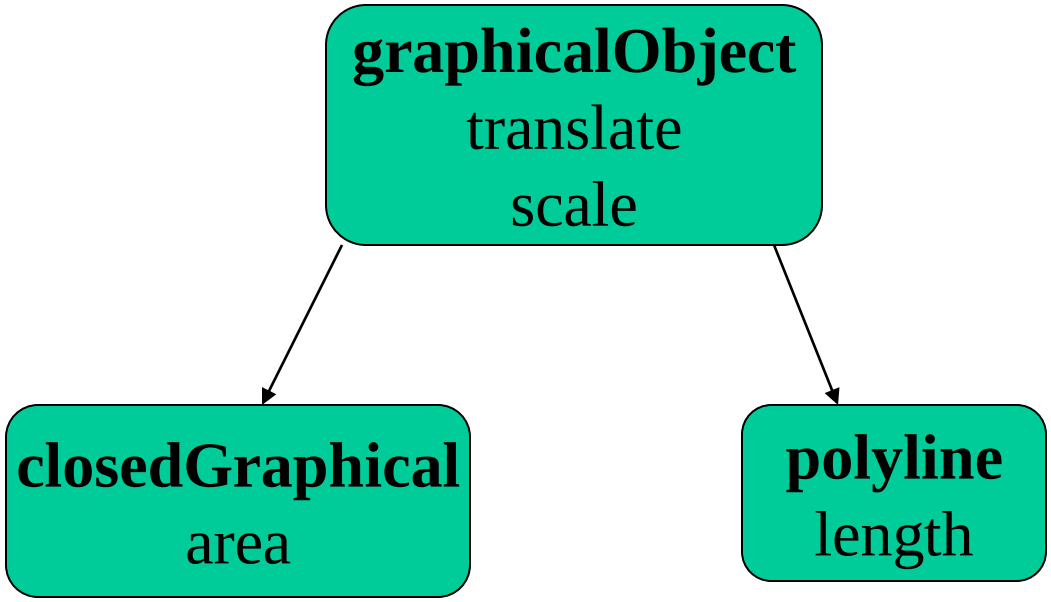
Attributs : données sur un objet

Méthodes : fonctions sur un objet

Classe : type d'objets

Héritage :

relation entre une classe et ou sous-classe ou classe dérivée (moins d'objets, plus d'informations)



Hiérarchie d'héritage

Les objets des classes dérivées ont les attributs et méthodes de la classe de base, plus d'autres

Traduction des méthodes en C++

Nom : `scale__15graphicalObject`

Paramètres : on ajoute `graphicalObject & this`

Appel : l'objet sur lequel on appelle la méthode devient 1er paramètre

Corps de la méthode

On accède aux attributs et méthodes de l'objet par l'intermédiaire de `this.`

`m__5a__1a` : `m__5a` dans `a` ou `m` dans `a__1a` ?

Polymorphisme et héritage

a := b

La syntaxe garantit que b peut être vu comme un a

Un type dérivé peut être vu comme le type de base (on perd les informations spécifiques au type dérivé)

Le type statique de a est déduit de la déclaration

Le type propre ou type dynamique de a est connu seulement à l'exécution

Certaines opérations dépendent du type propre (méthodes virtuelles)

Polymorphisme et héritage

Paramètres des fonctions

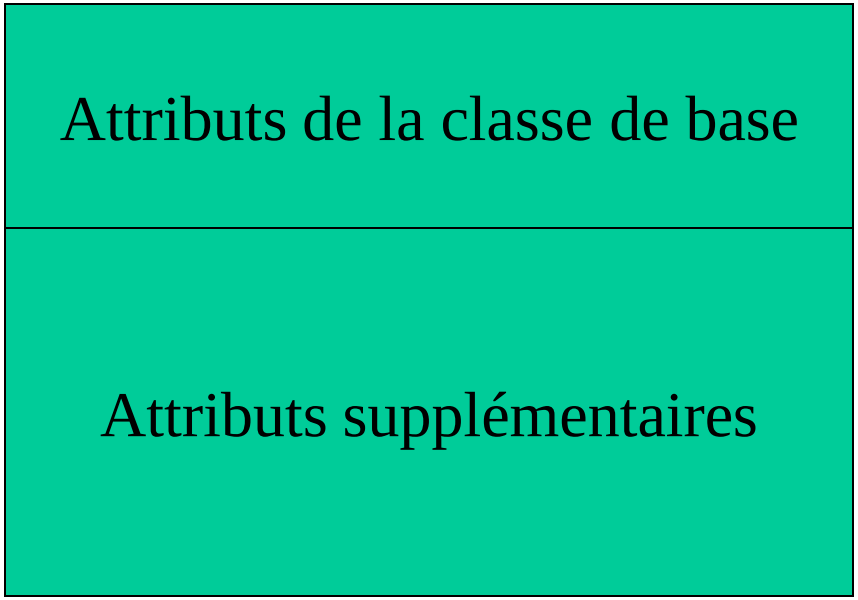
```
rectangle dist(point a) ;
dist(b)              // b peut être vu comme un a
```

Valeur de retour

```
rectangle dist(point a) {
// ...
return c ; }
          // c peut être vu comme un rectangle
```

Accès aux attributs

Les attributs définis dans la classe de base conservent la même adresse relative dans les classes dérivées. Cette adresse est connue à la compilation.



Méthodes statiques

Peuvent être héritées mais non redéfinies

Méthodes virtuelles ou dynamiques

On peut les redéfinir lors d'un héritage

Exemple. Dans la classe `polyline`, la méthode `scale` provoque un changement d'échelle.

```
class rectangle : public polyline {
   double side1_length, side2_length ; }
```

Il faut redéfinir `scale` pour qu'elle change les deux longueurs des côtés. Pour cela, la méthode `scale` doit avoir été définie `virtual` dans la classe de base

Sélection des méthodes

Sélection statique : par le type statique

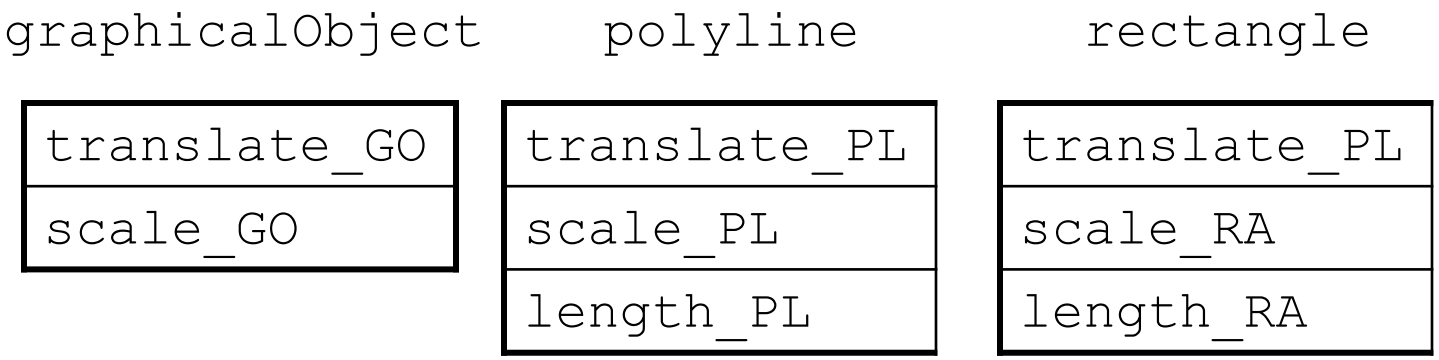
Sélection dynamique

La sélection de la méthode à appeler dépend du type propre de l'objet, connu seulement à l'exécution

L'objet a un lien vers un descripteur de classe

Le descripteur de classe a les liens vers le code des méthodes virtuelles (table virtuelle)

Tables virtuelles



L'adresse relative des liens est la même pour la définition de base et pour les redéfinitions

Le contenu du lien est l'adresse du code de la méthode. Il dépend du type propre

Traduction d'un appel

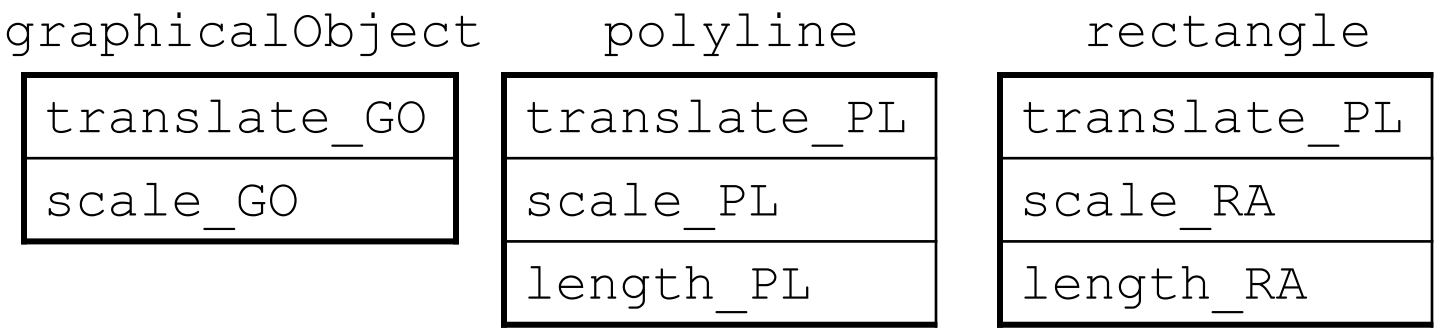
```
image.scale(factor)
```

Traduction :

```
vt=*image;
vm=vt[2];
(*vm)(image, factor) ;
```

Le compilateur relie les noms des méthodes virtuelles (`scale`) aux adresses relatives dans la table virtuelle (`2`)

Construction des tables virtuelles



On copie la table virtuelle de la classe de base

On remplace les méthodes redéfinies par l'adresse du nouveau code

On ajoute les nouvelles méthodes virtuelles

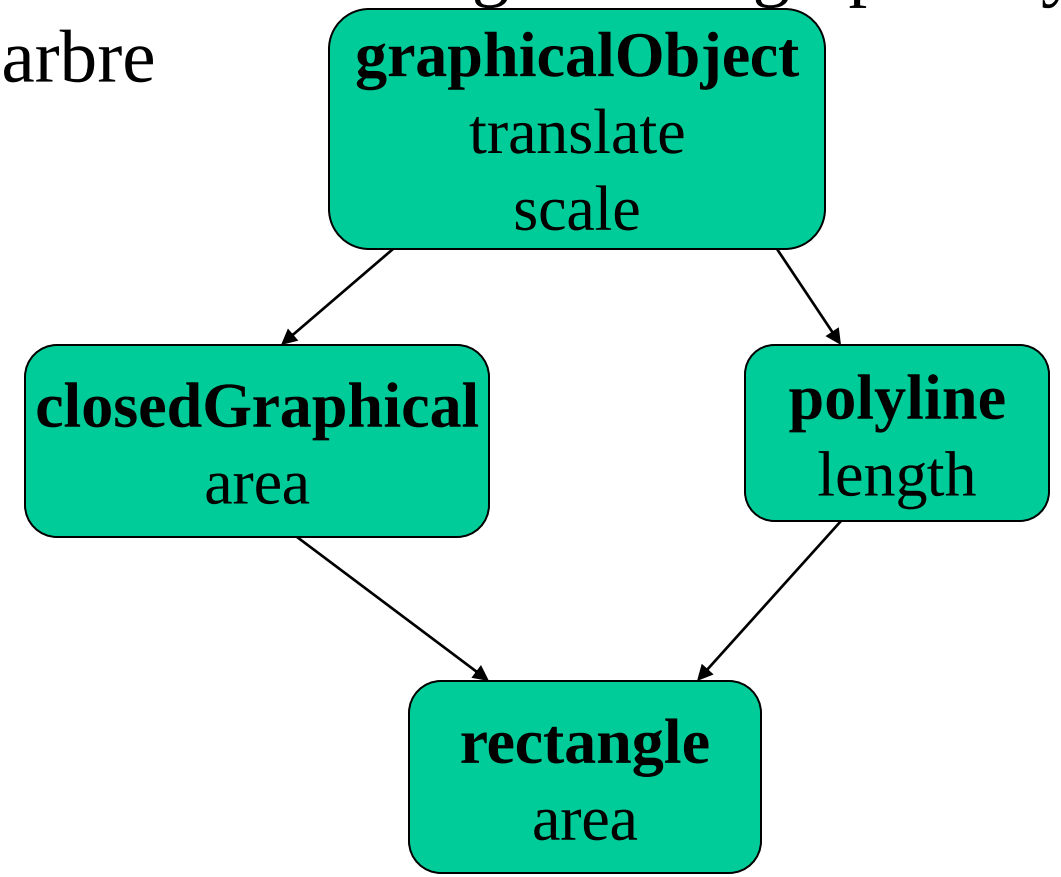
Coût de la sélection dynamique

- pour chaque objet, le pointeur vers la table virtuelle
- pour chaque classe, la table virtuelle
- à chaque appel de méthode, 1 indirection + 1 accès dans la table

Héritage multiple

Une classe peut avoir plusieurs classes mères

La hiérarchie d'héritage est un graphe acyclique, non un arbre



Conflits de noms

si deux classes mères utilisent le même nom pour des attributs ou méthodes

En C++, les conflits sont résolus par un préfixe de résolution de portée

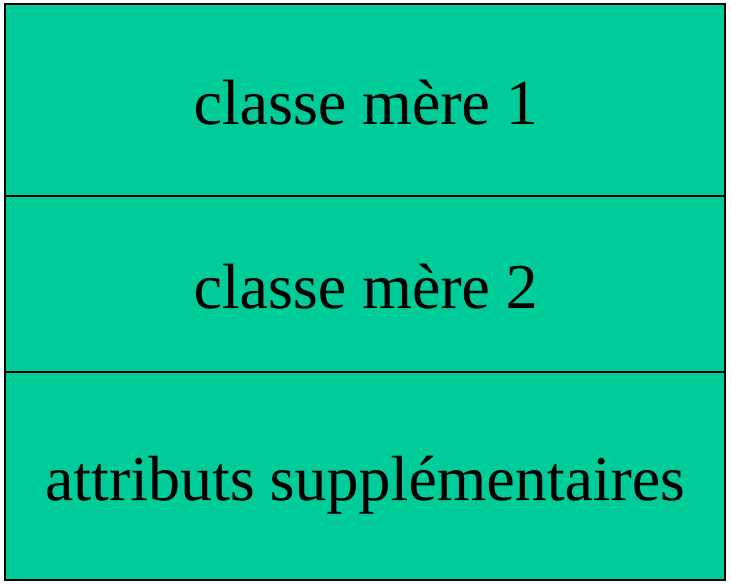
Héritage répété

Une classe dérivée hérite d'une autre classe plusieurs fois, directement ou indirectement

En C++, la classe mère est instanciée plusieurs fois dans la classe dérivée (la même méthode héritée plusieurs fois est utilisée pour des objectifs différents)

Compilation de l'héritage multiple

Pour chaque classe mère, le compilateur doit pouvoir voir l'objet comme un objet de la classe mère




```
myPolyline = myRectangle ;
```

le compilateur crée une vue `polyline` de `myRectangle`

Système de gestion d'exceptions

Objectif

Récupération après erreur

Exemple : une transaction commerciale échoue, mais le système propose au client de réessayer ou d'essayer autre chose

Principe

Bloc d'essai (`try`) : la gestion d'exceptions est active pendant l'exécution de ce bloc

Bloc de traitement d'erreur (`catch`)

Levée d'une exception (`throw`)

Gestion d'exceptions : syntaxe

Blocs

Le bloc d'essai et le bloc de traitement d'erreur sont des blocs au sens de la portée des variables

Exemple : les variables déclarées dans le bloc d'essai ne sont pas disponibles dans le bloc de traitement d'erreur

Bloc de libération (`finally`)

même chose

Gestion d'exceptions : syntaxe

Blocs

Un bloc d'essai est obligatoirement suivi

- d'un ou plusieurs blocs de traitement d'erreur
- ou d'un bloc de libération
- ou des deux

Imbrication de blocs

N'importe quel bloc peut contenir un bloc d'essai

Gestion d'exceptions à l'exécution

Propagation

Entre la levée de l'exception (`throw`) et son traitement (`catch`)

L'exception est comparée à plusieurs blocs de traitement d'erreur, jusqu'à en trouver un qui traite cette exception

Les blocs peuvent appartenir à des niveaux d'imbrication différents

Ils peuvent appartenir à des méthodes différentes

Gestion d'exceptions à l'exécution

```
try {
   // initialisations
   for (int i = 0 ; i < max ; i ++) {
        try {
               // essayer
               }
        catch(ArithmeticException e) {
               // réparer les dégâts
               }
        // terminé pour cette valeur de i
        }
   }
catch(RuntimeException e) {
   // on abandonne cette itération
   }
```

Gestion d'exceptions à l'exécution

```
public static void main(String[] args) {
   try { aventureux() ; }
   catch(RuntimeException e) {
        // réinitialiser
        }
private void aventureux() {
   try {
        // essayer
        }
   catch(ArithmeticException e) {
        // réparer les dégâts
        }
   // sauvegarder le résultat
   }
```

Gestion d'exceptions à l'exécution

Propagation

À chaque niveau, le bloc de libération (`finally`) est exécuté s'il existe

Gestion d'exceptions à l'exécution

```
try {
   // initialisations
   for (int i = 0 ; i < max ; i ++) {
        try {
               // essayer
               }
        finally {
               // libérer les ressources allouées pour i
               }
        // terminé pour cette valeur de i
        }
   }
finally {
   // libérer les ressources allouées pour l'itération
   }
```

Retour dans un bloc d'essai

Si une instruction de retour dans un bloc d'essai ou dans un bloc de traitement d'erreur est exécutée

Pas d'exception levée, mais le fonctionnement ressemble

Le contrôle remonte tous les niveaux de blocs d'essai dans la méthode

À chaque niveau, le bloc de libération (`finally`) est exécuté s'il existe

Gestion d'exception à l'exécution

Quand le bloc de traitement d'erreur qui traite cette exception est trouvé,

- l'exception est désactivée
- le bloc de traitement d'erreur est exécuté
- le bloc de libération est exécuté s'il existe
- la propagation s'arrête

L'exécution reprend après le bloc trouvé

Levée d'exception à l'exécution

Levée d'une exception (`throw`)

- calculer l'adresse du premier bloc de traitement (`catch`, BT) à essayer, et de l'enregistrement d'activation (EA) correspondant
- sauvegarder l'exception
- dépiler les EA plus récents si nécessaire

Recherche du BT à l'exécution

On connaît l'exception, et le premier BT à essayer

On saute les BT de ce niveau qui ne correspondent pas au type dynamique de l'exception

Si on trouve un BT qui correspond, on l'exécute et on désactive l'exception

Exécuter le bloc de libération (`finally`)

Si l'exception est toujours active, on la propage

Compilation : levée

Objectif : traduire la levée d'une exception

Calcul de l'adresse du premier BT à essayer, et de l'adresse de l'EA correspondant

Sauvegarde de l'exception

Saut inconditionnel vers le BT

Dépilement des EA plus récents si nécessaire

Compilation : levée

Calcul de l'adresse du BT et de l'EA

Si l'instruction courante est dans un bloc d'essai

Prendre le BT correspondant : son adresse est connue statiquement (table statique)

Prendre l'EA courant

Sinon

Le BT dépend de l'appel de la méthode courante

Il faut l'avoir indiqué dans l'EA courant

Compilation : appel de méthode

Objectif : préparer les levées d'exceptions qui se produiront pendant l'activation, en indiquant l'adresse du BT et de l'EA dans l'EA de la méthode appelée

Si l'appel est dans un bloc d'essai

Prendre le BT correspondant

Prendre l'EA de la méthode appelante

Sinon

Copier les informations (BT, EA) indiquées dans l'EA de la méthode appelante

Compilation : bloc d'essai

Objectif : préparer les levées d'exceptions qui se produiront pendant l'exécution du bloc d'essai

Dans la table statique, pour la zone de code correspondant au bloc d'essai, indiquer l'adresse du premier BT

Pour la zone de code correspondant aux BT et au bloc de libération, l'adresse doit être celle du bloc d'essai englobant s'il existe

Cela permet de relever une exception depuis un BT

Compilation : blocs de traitement

Objectif : traduire des blocs de traitement

Avant chaque bloc de traitement, comparaison du type de l'exception

Après chaque bloc, inactivation de l'exception traitée et saut inconditionnel vers le bloc de libération s'il existe

Compilation : bloc de libération

Objectif : traduire un bloc de libération

Après le bloc, propagation de l'exception si elle est toujours active

Compilation : définition de méthode

En Java, quand une méthode peut propager des exceptions qui ne sont pas du type RuntimeException, on doit les déclarer dans la définition (`throws`)

Vérifier que les exceptions levées dans le corps de la méthode (ou déclarées dans les définitions des méthodes appelées) sont déclarées dans définition de la méthode courante