## Plan Pédagogique du cours

**Module:** Programmation Orientée Objet

**Section :** informatique

**Niveau :** 3ème niveau (gestion, industriel, réseau)

**Volume Horaire :** 22,5 heures Cours Intégrés + 45 Travaux Pratiques

**Coefficient :** 3

**Evaluation :** Interrogations Orale - Travaux dirigés – Travaux Pratiques – Interrogations Ecrites

**Suivi des versions du support :**

| Version | Date | Rédigé Par | Raison |
|---|---|---|---|
| 1.0 | Juin 2004 | Tahar Haouet | Création du document |
| 2.0 | Juin 2006 | Tahar Haouet | Corrections et remise en forme |

**Objectif général du cours :**

L'objectif général de ce cours est de donner les concepts de base de la programmation orientée objets. Il s'appuie sur le langage C++.

Les principaux points abordés sont :

- Les origines et les attentes de la programmation orienté objets et les concepts qui en découlent.
- Le concept de classes et d'objets, les membres, fonctions membres, fonctions amies et le cas particulier très important des constructeurs et du destructeur.
- La notion d'héritage, simple puis multiple avec les notions de polymorphisme.

## Plan du cours

## Leçon 1 : Critère de qualité d'un logiciel

**Prérequis :**

Programmation I et II

**Objectif du chapitre :**

Connaître les critères de qualité d'un logiciel et les origines de l'approche objet

**Durée : 1h30**

**Eléments du Contenu :**

Critère de qualité du logiciel et du module, Principe de définition d'un module, la réutilisabilité, surcharge - généricité

## 1-1- Introduction

La conception par objet trouve ses fondements dans une réflexion menée autour de la vie du logiciel. D'une part, le développement de logiciels de plus en plus importants nécessite l'utilisation de règles permettant d'assurer une certaine qualité de réalisation. D'autre part, la réalisation même de logiciel composée de plusieurs phases, dont le développement ne constitue que la première partie. Elle est suivie dans la majorité des cas d'une phase dite de maintenance qui consiste à corriger le logiciel et à le faire évoluer. On estime que cette dernière phase représente 70 % du coût total d'un logiciel, ce qui exige plus encore que la phase de développement doit produire du logiciel de qualité.

La conception objet est issue des réflexions effectuées autour de cette qualité. Celle-ci peut-être atteinte à travers certains critères :

**– La validité** : c'est-à-dire le fait qu'un logiciel effectue exactement les tâches pour lesquelles il a été conçu.

**– l'extensibilité** : C'est-à-dire, la capacité à intégrer facilement de nouvelles spécifications (demandées par les utilisateurs ou imposées par un événement extérieur)

**– la réutilisabilité** : Les logiciels écrits doivent pouvoir être réutilisables, complètement ou en partie. Ceci impose lors de la conception une attention particulière à l'organisation du logiciel et à la définition de ses composantes ;

**– la robustesse** : c'est-à-dire l'aptitude d'un logiciel à fonctionner même dans des conditions anormales.

Bien que ce critère soit plus difficile à respecter, les conditions anormales étant par définition non spécifiées lors de la conception d'un logiciel, il peut être atteint si le logiciel est capable de détecter qu'il se trouve dans une situation anormale.

Nous détaillons dans un premier temps les critères utilisés pour concevoir du logiciel de qualité, tels que la modularité ou encore la réutilisabilité.

## 1-2- La Modularité

Les critères énoncés au paragraphe précédent influent sur la façon de concevoir un logiciel, et en particulier sur l'architecture logicielle. En effet, beaucoup de ces critères ne sont pas respectés lorsque l'architecture d'un logiciel est obscure. Dans ces conditions, le moindre changement de spécification peut avoir des répercutions très importantes sur le logiciel, imposant une lourde charge de travail pour effectuer les mises à jour.

On adopte généralement une architecture assez flexible pour parer ce genre de problèmes, basée sur les modules. Ceux-ci sont des entités indépendantes intégrées dans une architecture pour produire un logiciel.

L'ensemble des modules utilisés, ainsi que les relations qu'ils entretiennent entre eux, sont nommé système. L'intérêt de ce type de conception est de concentrer les connaissances liées à une entité logique à l'intérieur d'un module qui est seul habilité à exploiter ces connaissances. L'une des conséquences immédiates est que lorsqu'une maintenance est à effectuer sur une entité logique, celle-ci ne doit concerner qu'un seul module, ce qui confine la maintenance.

### 1-2-1. Deux méthodes de conception de modules

Si la définition de modules est une approche communément admise, il faut également une méthode de construction de systèmes qui permette de déduire quels sont les bons modules. Il existe deux grandes familles de méthodes modulaires :

– Les méthodes descendantes qui procèdent par décomposition de problème. Un problème est ainsi divisé en un certain nombre de sous-problèmes, chacun de complexité moindre. Cette division est ensuite appliquée aux sous-problèmes générés, et ainsi de suite, jusqu'à ce que chacun des sous-problèmes soit trivial.

– Les méthodes ascendantes qui procèdent par composition de briques logicielles simples, pour obtenir des systèmes complets. C'est en particulier le cas des bibliothèques de sous-programmes disponibles avec tous les systèmes, langages, environnements...

Les deux méthodes ne sont pas automatiquement à opposer, et sont souvent utilisées en même temps lors de la conception d'un logiciel. On peut cependant noter que l'approche descendante ne favorise pas toujours la réutilisabilité des modules produits.

### 1-2.2 Quelques critères de qualités

En dehors de la démarche même menant aux modules, il est bon de préciser quelques critères de qualité à respecter lors de la définition des modules :

**– Compréhensibilité modulaire** : les modules doivent être clairs et organisés de manière compréhensible dans le système. Ceci implique que les modules doivent communiquer avec peu de modules, ce qui permet de les «situer» plus facilement.

**– Continuité modulaire** : ce critère est respecté si une petite modification des spécifications n'entraîne qu'un nombre limité de modifications au sein d'un petit nombre de modules, sans remettre en cause les relations qui les lient.

**– Protection modulaire** : Ce critère signifie que toute action lancée au niveau d'un module doit être confinée à ce module, et éventuellement à un nombre restreint de modules. Ce critère ne permet pas de corriger les erreurs introduites, mais de confiner autant que possible les erreurs dans les modules où elles sont apparues.

### 1.2-3 Les principes de définition

À partir des critères exposés ci-dessus, quelques principes de conception ont été retenus pour la réalisation de modules :

**– Interface limitée** : le but n'est pas de borner les actions associées à un module, mais de se restreindre à un nombre limité d'actions bien définies, ce qui supprime une part des erreurs liées à l'utilisation de modules.

**– Communications limitées** : les communications entre modules, réalisées via leur interface, doivent être limitées de façon quantitative. Ceci est une conséquence du principe de modularité, qui est d'autant mieux respecté que les modules jouent leur rôle. Si les échanges sont trop importants, la notion même de module devient flou, limitant l'intérêt de cette technique.

**– Interface explicites**: les communications entre modules doivent ressortir explicitement.

**– Masquage de l'information** : toutes les informations contenues dans un module doivent être privées au module, à l'exception de celles explicitement définies publiques. Les communications autorisées sont ainsi celles explicitement définies dans l'interface du module, via les services qu'il propose.

– Les modules définis lors de la conception doivent correspondre à des unités modulaires syntaxiques liées au langage de programmation. En clair, **le module spécifié ne doit pas s'adapter au langage de programmation, mais au contraire le langage de programmation doit proposer une structure permettant d'implanter le module tel qu'il a été spécifié**. Par exemple, si le langage de programmation ne permet pas d'effectuer le masquage de l'information (comme le langage C), il n'est pas adéquat pour implanter les modules de manière satisfaisante.

Ce genre de critères proscrit ainsi des comportements tels que l'utilisation de variables globales par exemple, qui se contredit avec les principes énoncés. En effet, les variables globales peuvent être utilisées et modifiées par n'importe quelle composante d'un programme, ce qui complique d'autant la maintenance autour de ce genre de variables.

## 1-3- La réutilisabilité

La réutilisabilité n'est pas un concept nouveau en informatique et a été utilisée dès les premiers bafouillages.

En effet, les types de données à stocker sont toujours construits autour des mêmes bases (tables, listes, ensembles) et la plupart des traitements comportent des actions atomiques telles que l'insertion, la recherche, le tri, ... qui sont des problèmes résolus en informatique. Il existe une bibliographie assez abondante décrivant des solutions optimales à chacun de ces problèmes. La résolution des problèmes actuels passe par la composition des solutions de chacun de ces problèmes basiques.

Les bibliothèques (systèmes, mathématiques, etc.) sont des bons exemples de réutilisabilité et sont couramment utilisés par les programmeurs. Elles montrent cependant parfois leurs limites. En effet, les fonctions qu'elles comportent ne sont pas capables de s'adapter aux changements de types ou d'implantation.

La solution dans ce cas est de fournir une multitude de fonctions, chacune adaptée à un cas particulier, ou d'écrire une fonction prenant tous les cas en considération. Dans un cas comme dans l'autre, ce n'est que peu satisfaisant. C'est pourquoi la conception objet se propose de formaliser un peu plus cette notion de réutilisabilité et de proposer de nouvelles techniques pour l'atteindre pleinement.

### 1-3.1 Les principes de la réutilisabilité

Le paragraphe précédent a introduit la notion de module, en insistant sur les avantages de la conception modulaire, mais n'a pas donné de détails sur la conception même d'un module. On conviendra ici qu'un « bon » module est un module réutilisable, c'est-à-dire conçu dans l'optique d'être placé dans une bibliothèque à des fins de réutilisation. Affin de marier modularité et réutilisabilité, quelques conditions nécessaires à la conception de bons modules ont été définis :

– un module doit pouvoir manipuler plusieurs types différents. Un module de listes par exemple doit pouvoir manipuler aussi bien des entiers que des types composites.

– de même, un module doit pouvoir s'adapter aux différentes structures de données manipulées dotées de méthodes spécifiques. Il devra ainsi par exemple pouvoir rechercher de la même manière une information contenue dans un tableau, une liste, un fichier.

– un module doit pouvoir offrir des opérations aux clients qui l'utilisent sans que ceux-ci connaissent l'implantation de l'opération. Ceci est une conséquence directe du masquage de l'information préconisé. C'est une condition essentielle pour développer de grands systèmes : les clients d'un module sont ainsi protégés de tout changement de spécifications relatif à un module.

- Les opérations communes à un groupe de modules doivent pouvoir être factorisées dans un même module. Ainsi par exemple, les modules effectuant du stockage de données, tels que les listes, les tables, etc. doivent être dotés d'opérations de même nom permettant d'accéder à des éléments,

d'effectuer un parcours, de tester la présence d'éléments. Ceci peut permettre entre autres de définir des algorithmes communs, tels que la recherche, quelle que soit la structure de données utilisée pour stocker les données.

### 1-3.2 De nouvelles techniques

L'idée, afin de faire cohabiter les principes inhérents à la modularité et à la réutilisabilité, est d'utiliser la notion de paquetage, introduite par des langages tels que Ada ou Modula-2. Un paquetage correspond à un regroupement, au sein d'un même module, d'une structure de données et des opérations qui lui sont propres. Ceci satisfait en particulier les critères de modularité, en isolant chaque entité d'un système, ce qui la rend plus facile à maintenir et à utiliser. En ce qui concerne les critères de réutilisabilité, il est possible d'aller encore un peu plus loin. Nous introduisons ici de nouvelles notions qui apparaissent avec les paquetages et vont permettre de franchir ce pas :

**– La surcharge** : cette notion prévoit que des opérations appartenant à des modules différents peuvent être associées au même nom. Les opérations ne sont donc plus indépendantes, elles prennent leur signification contextuellement en fonction du cadre dans lequel elles sont utilisées. Parmi ces opérations, on trouve les fonctions, mais également les opérateurs. Cela peut permettre par exemple de définir une fonction insérer dans chaque module de stockage, permettant d'écrire de manière uniforme : insérer (elt, container) quelque soit le type de container (liste, tableau, fichier...)

**– La généricité** : cette notion permet de définir des modules paramétrés par le type qu'ils manipulent.

Un module générique n'est alors pas directement utilisable : C'est plutôt un patron de module qui sera « instancié » par les types paramètres qu'il accepte. Cette notion est très intéressante, car elle va permettre la définition de méthodes (façon de travailler) plus que de fonctions (plus formelles).

Ces définitions et ces nouveaux outils vont nous permettre de définir de nouvelles manières de concevoir des systèmes informatiques.

## Leçon 2 : Principes de conception Objet

**Prérequis :**

Leçon 1, Programmation I et II

**Objectif du chapitre :**

Connaître les limites de l'approche de développement fonctionnelle et introduction de la notion Orientée Objet

**Durée : 1h30**

**Eléments du Contenu :**

Le découpage fonctionnel d'un problème informatique
Introduction à la conception par objet

## 2.1 Introduction

Après avoir énuméré les qualités souhaitables nécessaires à l'élaboration d'un système de qualité, il nous reste maintenant à déterminer les règles de construction de tels systèmes. D'un point de vue général, la construction d'un système informatique se résume par la formule :

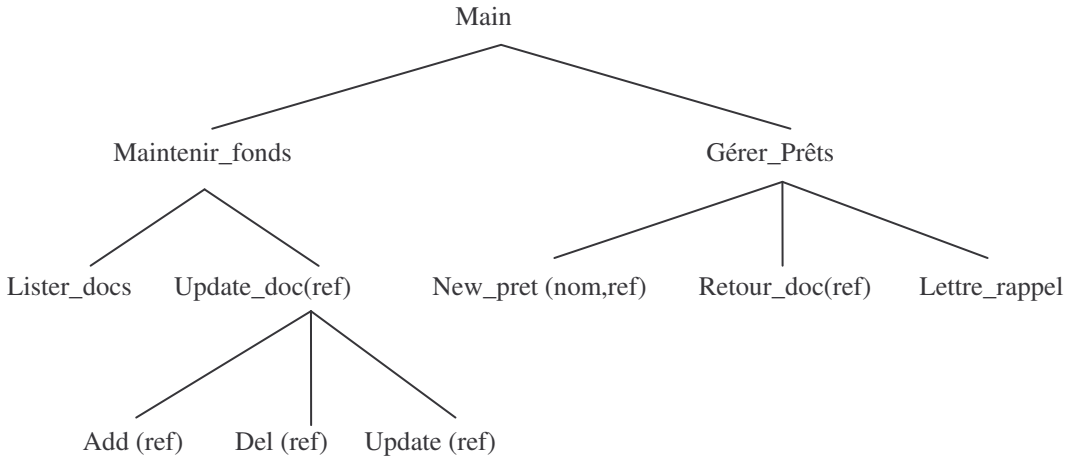
Algorithmes + Structures de données = Programme

Le concepteur d'un système informatique a donc deux grandes options pour l'architecture d'un système :

Orienter sa conception en se basant sur les données ou sur les traitements.

Dans les méthodes de conception par traitements, qui constituent l'approche traditionnelle, la base de la réflexion est effectuée autour des traitements. Le concepteur considère ainsi les tâches que doit accomplir le programme, en décomposant celui-ci en une série de tâches simples (approche descendante) ou en le construisant par composition de traitements (fonctions) disponibles (approche ascendante)

**Exemple :** (Découpage fonctionnel d'un problème informatique)
Gestation d'une bibliothèque




**critiques :**

- Approche intuitive
- Hiérarchie de fonctions réalisant les services désirés
- On peut réutiliser les fonctions déjà définies pour créer de nouvelles fonctions (la fonction « update(ref) » peut être utilisée par « new_pret » et « retour doc »
- L'évolution du logiciel est délicate car les fonctions deviennent interdépendantes (modifier « update(ref) » entraîne en cascade des modifications dans « update_doc », « new_pret » et « retour_doc »
- Les données et les fonctions (travaillant sur ces données) sont dispersées.

→Solution : regrouper les données et le code travaillant sur ces données dans une même entité. (**Objet**)

**Remarques**

On peut faire les remarques suivantes sur ce type d'approche :

– Les modules trouvés par cette approche sont adaptés au type de problème posé au départ et ne permettant que peu d'extensibilité du système obtenu et que peu de réutilisabilité.

– Les traitements définis ne prennent pas assez en considération les structures de données sous-jacentes, qui se retrouvent partagées entre plusieurs modules. Il devient difficile dans ces conditions de maintenir et de protéger ces structures de données.

– Les traitements sont généralement beaucoup moins stables que les données. En effet, un programme une fois terminé se verra souvent étendre par de nouvelles fonctionnalités. Les données en revanche sont beaucoup plus stables, et si des modifications interviennent, elles ne changent pas radicalement leurs représentations.

Bien sûr, les approches basées sur les traitements ont quand même quelques avantages, dont celui d'être relativement intuitives et facilement applicables. Elles peuvent à ce titre être utilisées pour la réalisation d'applications de taille raisonnable. Elles se révèlent cependant rapidement inadaptées lors de la réalisation de systèmes plus conséquents ou sensibles aux changements de spécification.

## 2.2 La conception par objets

Afin d'établir de façon stable et robuste l'architecture d'un système, il semble maintenant plus logique de s'organiser autour des données manipulées, les objets. En effet, les données étant de par leur nature plus stables que les traitements, la conception en est simplifiée. De plus il apparaît que la conception par traitement ne favorise pas l'utilisation des principes de qualité mis en évidence, tels que la modularité ou la réutilisabilité. Il reste maintenant à éclaircir les fondements mêmes de la conception par objets. Dans cette optique, voici une première définition de la conception par objets:

« La conception par objet est la méthode qui conduit à des architectures logicielles fondées sur les OBJETS que tout système ou sous-système manipule » ou encore «Ne commencez pas par demander ce que fait le système, demandez À QUOI il le fait !»

La spécification d'un système va donc maintenant s'axer principalement sur la détermination des objets manipulés. Une fois cette étape réalisée, le concepteur n'aura plus qu'à réaliser les fonctions de haut-niveau qui s'appuient sur les objets et les familles d'objets définis. C'est cette approche, préconisant de considérer d'abord les objets (c'est-à-dire en première

approche les données) avant l'objectif premier du système à réaliser, qui permet d'appliquer les principes de réutilisabilité et d'extensibilité.

## 2-2-1 Objectifs de la programmation par objets

**- Modélisation du monde réel**

Le monde réel est constitué d'objets (caractéristiques et comportement) en interaction. Modélisation proche du monde réel

**Exploitation de la redondance** : Dans le monde réel on a de nombreux représentants mais peu de concepts différents.

Il y a 6 milliards d'êtres humains sur terre mais une seule espèce :
« homo sapiens »

- **Réutilisation de composants logiciels :**

Utilisation de composants préexistants (objets) comme :
En électronique : composants, circuits intégrés, modules,…
Constructions automobiles : moteurs, roues,etc…
Jeu d'enfants : les Legos.

- **Réduction de l'impact des modifications et extensions**

Grâce au confinement dans des petites unités (objets)

- **Réduction et explicitation du couplage entre différentes parties d'une application**

Définition d'une interface minimale et clairement définie entre objets

- **Protection et abstraction des entités**

En séparant l'interface de l'implémentation

## 2-2-2 Détermination des Objets

Pour la détermination même des objets, il faut simplement se rattacher aux objets physiques ou abstraits qui nous entourent.

Typiquement, dans un programme de Bibliothèque, les ouvrages, les étudiants, les enseignants, etc. sont des objets. Les objets vont couvrir tous les types d'entités, des plus simples aux plus complexes, et sont à ce titre partiellement similaires aux structures utilisées dans des langages tels que C ou Pascal. Mais ils sont également dotés de nombreuses propriétés supplémentaires très intéressantes.

Le monde dans lequel nous vivons est constitué d'objets matériels de toutes sortes. Par extension on peut définir d'autres objets, sans masse, comme les comptes en banque, …ces objets correspondent plutôt à des concepts qu'à des entités physiques.

Les objets informatiques définissent une représentation abstraite des entités du monde réel ou virtuel, dans le but de les piloter ou de les simuler comme les êtres vivants, les objets du monde réel qui nous entourent naissent, vivent et meurent ; la conception objet permet de représenter le cycle de vie des objets au travers de leurs interactions.

## 2-3- Conclusion

Pour conclure, faisons le point sur les objectifs souhaités pour obtenir du logiciel de qualité et les techniques introduites pour atteindre ce but :

**– modularité** : cette technique permet de découper un système complet en un ensemble de modules qui sont indépendant ;
**– réutilisabilité** : Les modules produits peuvent être regroupés en bibliothèques et être réutilisés.
**– abstraction de données et masquage de l'information:** les modules n'indiquent pas la représentation physique des données qu'elles utilisent, mais se contentent de présenter les services qu'elles offrent.
Le concept de généricité permet encore d'accroître cette abstraction, en proposant des modules qui sont paramétrées par des types de données ;
**– extensibilité** : les modules sont définis en terme de services. Dès lors, un changement de représentation interne de données ou une modification de celles-ci n'altère pas la façon dont les autres classes les utilisent.

**– lisibilité** : l'interface (documentée) permet d'avoir un mode d'emploi clair et précis de l'utilisation d'un module, qui est d'autant plus clair que l'implantation des modules est cachée.

## Leçon 3 : L'objet

**Prérequis :**

Leçons précédents, Programmation I et II

**Objectif du chapitre :**

Introduire la notion d'objet en programmation et ses caractéristiques

**Durée : 1h30**

**Eléments du Contenu :**

Caractéristiques fondamentales d'un objet (état, comportement, identité)

## 3-1 Introduction

Le monde dans lequel nous vivons est constitué d'objets matériels de toutes sortes. Par extension, il est possible de définir d'autres objets, sans masse, comme les comptes en banque, les polices d'assurance... Ces objets correspondent plutôt à des concepts plutôt qu'à des entités physiques.

## 3-2 Les objets

L'objet est une unité atomique formée de l'union d'un état et d'un comportement. Il fournit une abstraction qui assure à la fois une cohésion interne très forte et un faible couplage entre le dit objet et l'extérieur. L'objet révèle son vrai rôle et sa vraie responsabilité lorsque, par l'intermédiaire de l'envoi de message, il s'insère dans un scénario de communication.

### 3-2-1 Caractéristiques fondamentales d'objet

Tout objet présente les trois caractéristiques suivantes: un état, un comportement et une identité.

| *OBJET = ETAT + COMPORTEMENT + IDENTITE* |
|---|

**Exemple :**

**Identité** : un nom qui permet de distinguer un objet d'un autre objet

**Un état** : ensemble de valeurs associées à des propriétés et qui caractérisent l'objet à un instant T

**Un comportement** : ensemble de services ou d'opérations que peut rendre un objet ou qui modifient son état

| *Fusée rouge : voiture* |
|---|
| Marque = « Ferrari »<br>Couleur = « rouge »<br>Proprio = «toto »<br>Vitesse = 290 km/h |
| Démarrer ( )<br>Tourner ( )<br>Accélérer ( )<br>Ralentir ( )<br>Arrêter ( ) |

**L'état :**

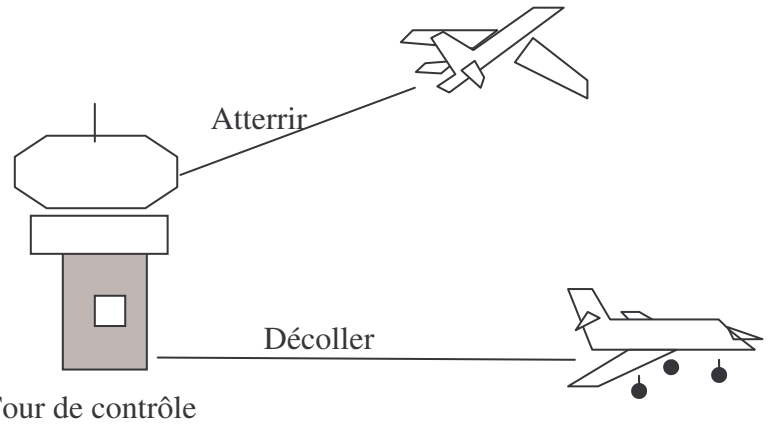
L'état regroupe les valeurs instantanées de tous les attributs d'un objet (c'est à dire ses propriétés) sachant qu'un attribut est une information qualifiant l'objet qui le contient. Chaque attribut peut prendre une valeur dans un domaine de définition donné. L'état d'un objet, à un instant donné, correspond à une sélection de valeurs, parmi toutes les valeurs possibles des différents attributs.

Cet état évolue au cours du temps; ainsi, lorsqu'une voiture roule, la quantité de carburant diminue, les roues s'usent. Certaines composantes de l'état peuvent être constantes, c'est le cas de la marque,… Toutefois, en règle générale, l'état d'un objet est variable et peut être vu comme la conséquence de ses comportements passés.

**Le comportement**

Le comportement regroupe toutes les compétences d'un objet et décrit les actions et les réactions de cet objet. Chaque atome de comportement et appelé opération (ou méthode) Les opérations sont déclenchées suite à une simulation externe, représenté sous la forme d'un message envoyé par un autre objet

L'état et le comportement sont liés; en effet, le comportement à un instant donné dépend de l'état courant, et l'état peut être modifié par le comportement Il est possible de faire atterrir un avion à la condition qu'il soit en train de voler; en d'autres termes, le comportement atterrir n'est valide que si l'information en vol est valide. Après l'atterrissage, l'information en vol devient invalide et l'opération atterrir n'a plus




de sens

**L'identité**

En plus de son état, un objet comporte une identité qui caractérise son existence propre. L'identité permet de distinguer tout objet de façon non ambigu, et cela indépendamment de son état. Ainsi, il est possible de distinguer deux objets dont toutes les valeurs d'attributs sont identiques L'identité est un concept ; elle ne se présente pas de manière spécifique en modélisation. Chaque objet possède une identité attribuée de manière implicite à la création de l'objet et jamais modifiée en phase de réalisation, l'identité est souvent construite à partir d'un identifiant issu naturellement du domaine du problème. Ces identifiant uniques sont appelés clef naturelle et peuvent être rajoutés dans l'état des objets afin de les distinguer.

## Leçon 4 : La classe

**Prérequis :**

Leçons précédents, Programmation I et II

**Objectif du chapitre :**

Définir une classe, ses propriétés et ses méthodes

**Durée : 3 h**

**Eléments du Contenu :**

Introduire la notion de classe.

Connaître la notion d'abstraction, d'encapsulation

Propriétés privées, publiques et protégées

Déclaration d'une classe en C++

## 4-1- Notion de Classe

Le monde réel est constitué de très nombreux objets en interaction. Ces objets sont des mélanges souvent trop complexes pour être compris du premier coup dans leur intégralité. Pour réduire cette complexité ou du moins pour la maîtriser, et comprendre ainsi le monde qui l'entoure, l'être humain a appris à regrouper les éléments qui se ressemblent et à distinguer des structures de plus haut niveau d'abstraction, débarrassées de détails inutiles

### 4-1-1. La démarche d'abstraction

La démarche d'abstraction procède de l'identification des caractéristiques communes à un ensemble d'éléments, puis de la description condensée des ces caractéristiques dans ce qu'on appelle une classe.

### 4-1-2 Définition :

La classe décrit le domaine de définition d'un ensemble d'objets. Chaque objet appartient à une classe. Les généralités sont contenues dans la classe et les particularités sont contenues dans les objets. Les objets informatiques sont construits à partir de la classe, par un processus appelé instanciation. De ce fait, tout objet est une instance de classe.

Une classe est l'abstraction d'un ensemble d'objet qui possède une structure identique (Liste des attributs) et un même comportement (liste des fonctions)

## 4-2. Les attributs et les opérations

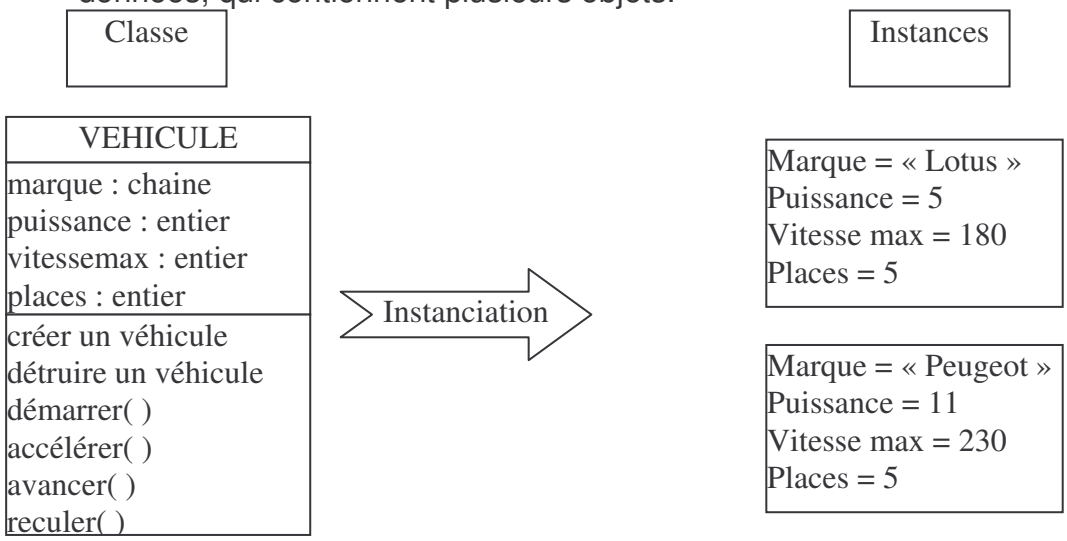
Les attributs d'une classe correspondent aux propriétés de la classe. Ils sont définis par un nom, un type et éventuellement une valeur initiale. Chaque objet, instance d'une classe, donne des valeurs particulières à tous les attributs définis dans sa classe et fixe par-là même son état.

La spécification du comportement d'un objet est définie par les opérations décrites dans sa classe. La réalisation du comportement est exprimée dans les méthodes.

Il existe cinq principales catégories d'opérations :

- Les constructeurs qui créent les objets;
- Les destructeurs qui détruisent les objets;
- Les sélecteurs (ou opération de consultation) qui renvoient tout ou partie de l'état d'un objet;
- Les modificateurs qui changent tout ou partie de l'état d'un objet;

- Les itérateurs qui visitent l'état d'un objet ou le contenu d'une structure de données, qui contiennent plusieurs objets.




## 4-3. Description des classes

La description des classes est séparée en deux parties :

- La spécification d'une classe qui décrit le domaine de définition et les propriétés des instances de cette classe, qui correspond à la notion de type.
- La réalisation qui décrit comment la spécification est réalisée et qui contient le corps des opérations et les données nécessaires à leur fonctionnement.

La séparation entre la spécification et la réalisation des classes participe à l'élévation du niveau d'abstraction. Les traits remarquables sont décrits dans les spécifications alors que les détails sont confinés dans les réalisations.

Le regroupement de la description de la structure des attributs (propriétés) et de la description des fonctions (opérations) dans une même classe porte le nom d'**encapsulation**. Les détails de l'implémentation d'un objet sont masqués aux autres objets du système.

Par défaut, les valeurs d'attributs d'un objet sont encapsulés dans l'objet et ne peuvent pas être manipulées directement par les autres objets. Toutes les interactions entre objets sont effectuées suite au déclenchement de diverses opérations déclarées dans la spécification de la classe et accessibles depuis les autres objets.

Les règles de visibilité viennent compléter ou préciser la notion d'encapsulation. Il existe trois niveaux distincts d'encapsulation couramment rencontrés.

- **Public** (+) C'est le niveau le plus faible, qui est obtenu en plaçant les attributs et/ou les opérations dans la partie publique de la classe. Cela revient à se passer de la notion d'encapsulation et à rendre visibles les attributs pour toutes les classes.
- **Protégé** (#) Il est possible de relâcher légèrement le niveau d'encapsulation, en plaçant certains attributs dans la partie protégée de la classe; Ces attributs sont alors visibles à la fois pour les amis et les classes dérivées de la classe fournisseur; Pour toutes les autres classes, les attributs restent invisibles.

Remarque : On verra en détail la notion de « protected » dans les leçons suivantes

- **Privé** (-) C'est le niveau le plus fort; la partie privée de la classe est alors totalement opaque et seuls les amis peuvent accéder aux attributs placés dans la partie privée.

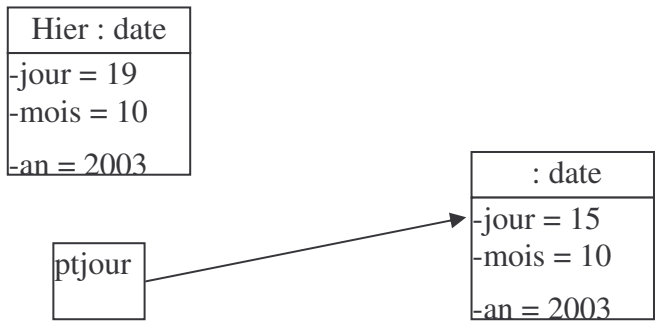
L'ensemble des éléments (attributs et opérations) d'une classe, rendus visibles aux autres porte le nom **d'interface**.
L'interface de la classe correspond à l'ensemble des méthodes publiques de la classe.

## 4-4 . Exemple : classe Date

- Création d'une classe Date :
  - 3 attributs entiers privés :
    jour, mois, année
  - 3 méthodes publiques :
    - initialiser la date
    - lire la date
    - afficher la date

-Création de deux instances :
  - statique : hier
  - dynamique : ptjour

| **Date** |
|---|
| **-jour : entier**<br>**-mois : entier**<br>-an : entier |
| **+ initialiser (E entier,entier, entier) :entier**<br>**+ lire_date (S entier,entier, entier) : entier**<br>**+afficher_date( )** |



En C++ :

```
# include <iostream>
using namespace std;
class date
{
private:
      int jour, mois, an ;
public:
      int initialiser (int j, int m, int a);
      void lire_date (int &, int &, int &) ;
      void afficher () const;
} ;

int date ::initialiser(int j, int m, int a)
{
      if (j >= 1 && j <= 31) jour = j ;
      else return 0 ;
      if (m >= 1 && m<=12) mois = m;
      else return 0;
      if (a > 0) an = a;
      else return 0;
return 1;
}

void date :: lire_date (int& j, int& m, int& a)
{
      j = jour ;
      m= mois ;
      a = an ;
}

void date :: afficher () const
{
      cout << jour<<"  "<<mois<<"   "<<an<<endl;
}

//-----------------------------------------------
main()
{
      date hier ;
      date *ptjour = new date ;
if (hier.initialiser(19,10,2003) )
      hier.afficher();
if (ptjour->initialiser(15,10,2003))
      ptjour->afficher();
}
```

## 4-5- Exemple : ratio (a/b)

```
//--------------------ratio.h---------------------------------
// Déclaration de la classe ratio
class ratio
{
private :
      int num, den;
public:
      void affecter (int,int);
      double valeur_reelle();
      void ecrire() ;
      void inverser() ;
} ;
// implémentation des méthodes membres
//-------------------------ratio.cpp-------------------------
void ratio : affecter(int numerateur , int denominateur)
{
      num = numerateur ;
      den = denominateur ;
}
double ratio ::valeur_reelle()
{
      return((double) num)/den ;
}
void ratio ::inverser()
{
      int tmp =num ;
      num=den ;
      den=tmp ;
}
void ratio::ecrire()
{
      cout<<num<<'/'<<den ;
}
//------------------------------main.cpp-----------------------
main()
{
      ratio r ;
      r.affecter(3,4);
      cout<<"r=";
      r.ecrire();
      out<<r="<<r.valeur_reelle()<<endl ;
}
```

```
// fin programme
resultat de l'execution :
r=3/4=0.75
```

### Affectation d'objets

En C++, Il est possible d'affecter des objets de même type. Cette affectation correspond à la recopie des valeurs des membres données, que ceux-ci soient publics ou non.

```
      ratio r1,r2 ;
      r1.affecter(3,4);
      r1=r2
```

### Données membres statiques

Lorsque dans un même programme, on crée différents objets d'une même classe, chaque objet possède ses propres membres données.

```
ratio r1,r2 ;
r1.affecter(3,4);
r2.affecter(1,2);
```

conduit à la situation suivante:

r1.num = 3 r2.num = 1

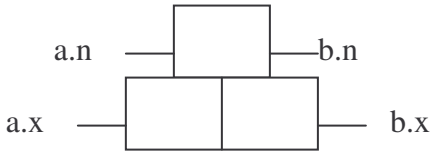
r1.den = 4 r2.den = 2

Pour permettre à plusieurs objets de partager des données. On déclare avec le spécificateur **static** les membres données qu'on souhaite voir exister en un seul exemplaire pour tous les objets de la classe.

Exemple

```
class r
{
      float x;
      static int n ;
      …
} ;
```

Conduit à une situation qu'on peut schématiser ainsi :




On peut dire que les membres données statiques sont des variables globales, dont la portée est limitée a la classe

**Initialisation des membres de données statiques**

Un membre donné statique est déclaré explicitement par une instruction telle que :

```
Int r ::n = 5 ;
```

Valable pour les membres privés ou publics

## Leçon 5 : Les Propriétés des fonctions membres

**Prérequis :**

Leçons précédentes, Programmation I et II

**Objectif du chapitre :**

Connaître les différentes techniques pour définir des fonctions membres de classes

**Durée :** 2 séances (2x 1h30)

**Eléments du Contenu :**

Surdéfinition des fonctions membres

Arguments par défauts de fonction

Objets en arguments de fonction

Autoréférence

Fonction membre constante

Fonction d'accès aux membres privés

## 5-1- Surdéfinition des fonctions membres

C++ Offre la possibilité de surdéfinir des fonctions ordinaires, cette possibilité s'étend aussi au fonctions membres d'une classe.

On peut appeler la fonction affecter(...) de deux manières. le compilateur analyse le nombre et les types d'arguments et appelle la fonction correspondante.

```
class ratio
{private :
int num, den;
public:
void affecter(int n, int d);
void affecter (int d);
…};
ratio :: affecter (int n, int d)
{
num =n;
den =d
}
ratio :: affecter (int d)
{
num =1;
den =d
}
```

## 5-2- Arguments par défauts

Comme le fonctions ordinaires, les fonctions membres peuvent disposer d'arguments par défauts. Voici comment on peut modifier l'exemple précédent pour que la classe ratio ne possède plus qu'une seule fonction affecter()

```
class ratio
{private :
int num, den;
public:
void affecter(int n=0, int d=1);
…};
ratio :: affecter (int n, int d)
{num =n; den =d;}
```

## 5-3- Objets transmis en argument d'une fonction membre

Une fonction membre peut recevoir un ou plusieurs arguments du type de sa classe.

**Exemple:**

On veut introduire au sein de la classe point une fonction membre nommée coïncide, chargée de détecter la coïncidence éventuelle de deux points.

Son appel au sein du main() sera sous la forme:

a.coïncide(b)

a et b étant des objets du type point.

La fonction coïncide doit être écrite de la façon suivante:

```
int point::coïncide (point pt)
Le corps de coïncide se présentera alors ainsi:
if ((pt.x ==x) && (pt.y  == y) ) return 1;
else return 0;
```

## 5-4- Modes de transmission des objets en arguments:

Dans l'exemple précédent **pt** était transmis classiquement à coïncide (par valeur)

Il est possible aussi de prévoir le passage des arguments par référence et par adresse

### 5-4-1 Transmission de l'adresse d'un objet

Il est possible de transmettre explicitement l'adresse d'un objet. Dans ce cas la fonction coïncide s'écrit de la façon suivante:

```
int point::coïncide (point *adpt)
{
if ((adpt-->x ==x) && (adpt->y  == y) ) return
1;
else return 0;
}
et l'appel de coïncide au sein du main devient:
a. coïncide (&b)
```

### 5-4-2 Transmission par référence d'un objet

L'emploi des références permet de mettre en place une transmission par adresse, sans avoir à en prendre en charge la gestion.

Voici une adaptation de coïncide dans laquelle son argument est transmis par référence.

```
int point::coïncide (point& pt)
{
if ((pt.x ==x) && (pt.y == y) ) return 1;
else return 0;
}
L'appel de coïncide au sein du main devient:
a.coïncide (b)
```

## 5-5- Objet résultat d'une fonction

Ce que nous avons vu à propos des arguments des fonctions membres d'une classe, s'applique également pour sa valeur de retour.

Cette dernière peut être un objet et on peut choisir l'un des modes de transmissions (par valeur, par adresse, par référence)

```
Exemple:
point point ::symétrique()
{
point res;
res.x = -x;
res.y = -y;
```

```
return res;
}
```

Il était nécessaire de créer un objet res au sein de la fonction

## 5-6- Autoréférence: le mot clé « this »

Dans certaines conditions particulières, il est nécessaire de disposer d'un moyen de designer depuis une fonction membre, non pas les données membres, mais l'instance elle-même de la classe sur laquelle la méthode membre est appelée

Le mot clé **« this »** permet de désigner l'adresse de l'instance sur laquelle la fonction membre a été appelée.

Exemple:

```
point* point::symétrique()
{
x = -x;
y = -y;
return (this);
}
main()
point a(1,2);
point pa;
pa = a.symétrique();
```

## 5-7- Fonctions membres statiques

On a vu que C++ permet de déclarer des données membres statiques, qui existent en un seul exemplaire (pour une classe donnée)
D'une manière analogue, on peut imaginer que certaines fonctions membres d'une classe, aient un rôle totalement indépendant d'un quelconque objet. Ce serait une fonction qui se contenterait d'agir sur des données membres statiques.

**Exemple :**

```
class cpte_obj
{
private:
static int crt ; // compteur du nombre d'objets crées
public:

cpte_obj();
~cpte_obj();
static void compte() ; // pour afficher le nombre d'objets crées
} ;
int cpte_obj ::crt = 0 ; // initialisation de membre statique crt
cpte_obj ::cpte_obj()
{
++crt ;
}
cpte_obj ::~cpte_obj()
{
--crt ;
}
void cpte_obj ::compte()
{
```

```
cout <<crt <<endl;
}
```

## 5-8- Fonctions membres constantes

C++ étend le concept de constance des variables aux classes. Ce qui signifie qu'on peut définir des objets constants.

Dans le cas des variables ordinaires, le compilateur peut facilement identifier les opérations interdites (celles qui peuvent en modifier la valeur)

Dans le cas d'un objet, les choses sont moins faciles, car les opérations sont généralement réalisées avec des fonctions membres. L'utilisateur doit spécifier parmi ces fonctions membres, lesquelles sont autorisées à opérer sur des objets constants.

On utilise le mot clé **const** dans leur déclaration

**Exemple :**

```
class point
{
int x,y ;
public :
point(..) ;
void affiche() const ;
void deplace( … ) ;
…
} ;
```

La fonction affiche est déclarée constante :

Elle est utilisable pour un objet déclaré constant.

```
point a ;

const point c ;
```

Les instructions suivantes sont correctes :

```
a.affiche() ;

c.affiche() ;

a.deplace(…) ;
```

Les instructions suivantes sont incorrectes

c.deplace(…) // c est const alors que déplace ne l'est pas

La même remarque s'applique à un objet reçu en argument

```
void f( const point p)

{

p.affiche(); //ok
```

```
    p.deplace(…); //incorrect

    }
```

Les instructions figurant dans sa définition ne doivent pas modifier la valeur des membres de l'objet point.

```
    void affiche() const

    {

    x++ //erreur, car affiche a été declaré
const

    }
```

Les membres statiques font exception à cette règle car ils ne sont pas associés à un objet particulier

## 5-9- Fonctions d'accès aux membres privés :

On reprend la classe ratio :

**Problème :**

On veut interdire la modification directe de num et den depuis l'extérieur de la classe tout en autorisant leurs lectures ?

**Solution :**

Limiter l'accès en donnant une fonction publique de lecture d'un membre privé :

**Exemple :**

```
    //ratio.h

    class ratio {

    private :

    int num,den;

    public:

    int numérateur() {return(num); };

    int denominateur() {return (den);};

    };
```

**Problème :**

Comment éviter de modifier la classe par erreur durant le déroulement de la fonction d'accès ?

**Solution :**

Considérer l'instance de classe comme **constante** vis-à-vis de la fonction :

```
class ratio
{
private :
 int num,den;
public:
 int numérateur() const;
 int denominateur() const;
}

int ratio::numerateur() const
{
return num;
};

int ratio::denominateur() const
{
return den;
};
```

## Leçon 6 : Construction et Destruction d'objets

**Prérequis :**

Leçons précédentes, Programmation I et II

**Objectif du chapitre :**

Connaître les différentes techniques de construction et de destruction d'objets

**Durée :** 1 séances (1h30)

**Eléments du Contenu :**

Construction et initialisation des objets

Les Constructeurs spéciaux (par défaut - par copie)

Les destructeurs

## 6-1- Les constructeurs

Il est souvent nécessaire d'initialiser les objets au moment de leur création. Dans le cas de la classe Date en particulier, on souhaite pouvoir initialiser le jour, le mois et l'année à toute nouvelle date créée. Une solution pourrait être de définir pour toutes ces classes une méthode init qui réalise les initialisations nécessaires. Pour toute création de nouvel objet, deux actions vont être nécessaires (déclaration + init) alors que la création d'un objet est apparemment une action atomique.

Pour résoudre ce problème, on doit avoir un mécanisme d'initialisation automatique d'objets de classe.

Une ou plusieurs méthodes particulières, appelées ***constructeurs***, sont appliquées implicitement dès qu'un objet est défini. Ces constructeurs généralement publics, portent le même nom que la classe à laquelle ils appartiennent.

## 6-2 Remarques :

- Les constructeurs n'ont aucun type de retour (même pas void) et ne sont jamais appelés explicitement par le programmeur.
- Les constructeurs portent le même nom que la classe
- Il peut avoir des paramètres et éventuellement des valeurs par défaut.

## 6-3 Exemple :

Au lieu d'utiliser affecter( { classe ratio}, on peut écrire un (voire plusieurs) constructeur(s) pour la classe. Un constructeur est chargé de créer une instance de la classe.

```
// Déclaration de constructeurs
class ratio
{private :
int num, den;
public:
ratio(int n, int d);
ratio(int n);
…};
```

Les définitions se font dans le fichier « .cpp » comme pour les autres fonctions membres.

```
//Définition de constructeurs
ratio ::ratio(int n, int d)
{
num = n ;
den = d ;
}
ratio::ratio (int n)
{
num = n;
den = 1;
}
```

ou encore mieux:

```
//ratio.h
ratio(int    n=0,    int
d=1) ;
```

```
// ratio.cpp
ratio ::ratio (int n, int d)
{num = n; den =d; }
```

## 6-4 Instanciation avec les constructeurs :

On peut désormais utiliser nos constructeurs et écrire plus simplement :

```
// main.cpp
main()
{
…
ratio r1 (1,3) ; // r1 = 1/3
ratio r2 (4); // r2 = 4/1 =4
ratio r3; // r3 = 0/1 =0
….
}
```

## 6-5 Liste d'initialisation des données membres

Un meilleur moyen d'affecter des valeurs lors de la construction aux données membres de la classe est :

```
//ratio.cpp
ratio ::ratio(int n, int d) :
num(n), den(d)
{}
```

Les constructeurs de num et den sont appelés au moment de la construction de Ratio, avant le constructeur de Ratio. Les liste d'initialisation permettent d'utiliser le constructeur de chaque donnée membre, et ainsi d'éviter une affectation après coup.

L'ordre d'initialisation doit correspondre à celui de la déclaration dans la classe.

## 6-6 Constructeurs spéciaux :

Deux constructeurs sont toujours automatiquement crées par le compilateur dans toute nouvelle classe :

- Le constructeur par défaut
- Le constructeur de copie

## 6-6-1 Constructeurs par défaut :

Le constructeur par défaut généré automatiquement par le compilateur est :

A ::A() ;

Il crée une nouvelle instance non initialisée si aucun autre constructeur fourni par l'utilisateur n'est applicable.

→ Eviter absolument de se servir de ce constructeur si possible puisqu'il crée des instances non initialisées.

### 6-6-2 Constructeur de copie

Le constructeur de copie généré automatiquement pour une classe A serait :

A ::A(A& autre_a)

{

// recopie de la zone mémoire utilisée par autre_a dans la zone de mémoire utilisée //par l'instance courante de la classe A.

}

Si la simple recopie de mémoire n'est pas adaptée à la recopie d'une telle classe alors il est impératif que le programmeur fournisse un constructeur de copie correct pour remplacer celui fait par le compilateur (appelé constructeur de recopie par défaut)

Le constructeur de copie est utilisé lors du passage de paramètre de type A par valeur, donc presque au moindre appel de fonction !

### 6-7- Destructeurs

Le destructeur est la méthode membre appelée lorsqu'une variable cesse d'exister en mémoire.

Son rôle est de libérer toutes les ressources qui ont été acquises lors de la construction

(Typiquement libérer la mémoire allouée dynamiquement pour la classe dans le constructeur)

**Remarques**

- Un destructeur n'a pas de paramètres
- Pas de surcharge possible
- Pas de valeur de retour

**Exemple :**

```
class ratio
{
~ratio() ;
…} ;
```

## Leçon 7 : Les Fonctions amies

**Prérequis :**

Leçons précédentes, Programmation I et II

**Objectif du chapitre :**

Apprendre une technique pour accéder aux membres privés des objets

**Durée :** 1 séances (1h30)

**Eléments du Contenu :**

Notion et situations d'amitiés

## 7-1-Problème :

Comment faire pour qu'une fonction f() et/ou une classe B puisse accéder aux données membres privées d'une classe A ?

**Exemple :**

```
class ratio
{
private :
int num, den ;
public:
ratio(int n, int d);
void affiche();
float val_reel();
}
ratio somme (ratio r1, ratio r2);
```

**Solution:**

- **Rendre publiques les données membres des classes.**

Inconvénient : on perd leurs protections.

- **Ajout de fonctions d'accès aux membres privés.**

Inconvénient : temps d'exécution pénalisant.

- **Fonctions amies :**

Il est possible de déclarer qu'une ou plusieurs fonctions (extérieurs à la classe), sont des « amies » ; une telle déclaration d'amitié les autorise alors à accéder aux données privées au même titre que n'importe quelle fonction membre.

L'avantage de cette méthode est de permettre le contrôle des accès au niveau de la classe concernée.

```
Déclaration :
Class A
{
private :
int i ;
friend class B ;
friend void f();
}
class B
{
//tout ce qui appartient à B,
//peut se servir des données membres privés de A
// comme si elle était de A.
...
}
void f(A& a)
{
a.i = 10 ;
}
```

## 7-2 Situation d'amitiés :

- Fonction indépendante amie d'une classe.
- Fonction membre d'une classe amie d'une autre classe.
- Fonction amie de plusieurs classes
- Toutes les fonctions membres d'une classe, amies d'une autre classe.

### 7-2-1 Exemple de fonction indépendante amie d'une classe :

```
classe point
{
private :
int x,y ;
public :
point (int abs= 0, int ord = 0)
{
x= abs ;
y= ord ;
} ;
// declaration d'une fonction amie indépendante
friend int coincide (point p, point q) ;

} ;
int coincide (point p, point q)
{
if ((p.x == q.x) && (p.y == q.y) )
return (1);
else return (0);
} ;
main()
{
point a(1,0), b (1), c ;
if  ( coincide (a,b)  )
cout<<"A coincide avec B"<<endl;
else
cout<<"A et B sont différents"<<endl;
} ;
```

### 7-2-2- Fonction membre d'une classe amie d'une autre classe :

On considère deux classes A et B

int f(char, A) fonction membre de B

f doit pouvoir accéder aux membres privés de A, elle sera déclarée amie au sein de la classe A :

friend int B :: f(char, A) ;

**Exemple :**

```
class A
{
private :
// partie privée
```

```
 public :
// partie publique
friend int B :: f (char, A) ;
...
...
} ;
class B
{
private :
// partie privée
 public :
// partie publique
int f (char, A) ;
...
...
} ;

int B :: f (char, A) ;
{
// on a ici accès aux membres privés de tout objet de type A

} ;
```

### 7-2-3 Fonction amie de plusieurs classes :

Rien n'empêche qu'une même fonction (indépendante ou membre) fasse l'objet de déclaration d'amitié dans différentes classes.

Toutes les fonctions d'une classe sont amies d'une autre classe

```
friend class B ; // dans la classe A
```

### 7-2-4 Toutes les fonctions d'une classe sont amies d'une autre classe.

```
friend class B ; //dans la classe A
```

# Leçon 8 : Surcharge d'opérateurs

**Prérequis :**

Leçons précédentes, Programmation I et II

**Objectif du chapitre :**

Connaître les différentes techniques d'intégration des opérateurs connus à des classes d'objets.

**Durée :** 1 séances (1h30)

**Eléments du Contenu :**

Mécanisme de surdéfinition

Surcharge d'opérateurs par des fonctions amies

Surcharge d'opérateurs par des fonctions membres

Surdéfition de manière générale

## 8-1- Introduction : surcharge d'opérateurs

C++ autorise la surdéfinition de fonctions, qu'il s'agisse de fonctions membres ou de fonctions indépendantes.

**Exemple :**

```
class ratio

{

private :

int num, den ;

public :

ratio ( n=0, d=1) ;

ratio(n) ;

} ;
```

Attribuer le même nom à des fonctions différentes, lors de l'appel, le compilateur fait le choix de la bonne fonction suivant le nombre et les types d'arguments.

C++ permet aussi la surdéfinition d'opérateurs

**Exemple :**

(Même dans C) ;

a+b ;

Le symbole + peut designer suivant le type de a et b :

L'addition de deux entiers

addition de deux réels

Addition de deux doubles.

+ est interprété selon le contexte.

En C++ on peut surdéfinir n'importe quel opérateur existant (unaire ou binaire). Ce qui va nous permettre de créer par le biais de classes, des types avec des opérateurs parfaitement intégrés.

**Exemple :**

On peut donner une signification à des expressions comme a+b, a-b, a*b et a/b pour la classe ratio

## 8-2- Mécanisme de surdéfinition :

Considérons la classe point :

```
class point
{
private :
int x, y ;
public :
// partie publique
...
} ;
```

Supposons que nous souhaitons définir l'opérateur (+) afin de donner une signification à l'expression tels que, a+b, a et b sont de type point.

On considère que la somme de deux points est un point dont les coordonnées sont la somme de leurs coordonnées.

Pour surdéfinir (surcharger) cet opérateur en C++, il faut définir une fonction de nom :

**operatorxx** où **xx** est le symbole de l'opérateur.

La fonction **operator+** doit disposer de deux arguments de types point et fournir une valeur de retour du même type.

Cette fonction peut être définie en tant que fonction membre de la classe (méthode) ou fonction indépendante (fonction amie)

On va examiner les deux solutions.

## 8-3- Surcharge d'opérateurs par des fonctions amies.

Le prototype de notre fonction sera :

**point operator+ (point, point)**

Les deux opérandes correspondent aux deux opérandes de l'opérateur +

Le reste du travail est classique.

Déclaration d'amitié

Définition de la fonction

**Exemple :**

```
class point
{
private :
int x, y ;
public :
point (int abs =0, ord =0) ;
friend point operator+ (point , point) ;
```

```
void afficher ( );
} ;
point operator+ (point a, point b)
{
point p;
p.x =a.x +b.x;
p.y = a.y + b.y;
return p;
}
main()
{
point a (1,2);
point b (2,5);
point c;
c= a+b; c.afficher();
c= a+b+c; c.afficher();
}
```

On a surdéfini (surchargé) l'opérateur **+** pour des objets de type point en employant une fonction amie.

## 8-4- Surcharge d'opérateurs par des fonctions membres :

Dans ce cas, la première opérande sera l'objet ayant appelé la fonction membre

**Exemple :**

L'expression a+b sera interprétée par le compilateur comme a.operator+ (b) ;

Le prototype de notre fonction membre est alors

point operator+ (point) ;

**Exemple:**

```
class point

{

private:

int abs, ord;

public:

point (int abs, int ord);

point operator+ (point a);

void affiche();

};

point point ::operator+ (point a);

{

point p;
```

```
    p.x = x + a.x;

    p.y = y + a.y;

    return (p) ;

    }

    main()

    {

    point a (1,2);

    point b (2,5);

    point c;

    c= a+b; c.afficher();

    c= a+b+c; c.afficher();

    }
```

**Remarque :**

Dans ce cas

c = a+b; c = a.operator+ (b);

Dans le 1er cas :

a= operator+(a,b);

## 8-5 Surdéfinition en général:

On a vu un exemple de surdéfinition de l'opérateur + lorsqu'il reçoit deux opérandes de type point. Ces de deux façons :

- fonctions amies
- fonctions membres

**Remarques :**

- Il faut se limiter aux opérateurs existants. Le symbole qui suit le mot clé **operator** doit obligatoirement être un opérateur déjà défini pour les types de base. Il n'est pas permis de créer de nouveaux symboles.
- Certains opérateurs ne peuvent pas être redéfinis de tout (.)
- Il faut conserver la pluralité de l'opérateur. unaire ou binaire.

binaire : + - / * -> ( )

unaire -- ++ new delete

- Il faut se placer dans un contexte de classe :

On ne peut surdéfinir un opérateur que s'il comporte au moins un argument de type classe →

Une fonction membre.

Une fonction indépendante ayant au moins un argument de type classe. (Fonction amie)

Cette règle permet d'interdire de surdéfinir un opérateur portant sur des types de base (3+5) devient (3 ou 5)

## 8-6 Choix entre fonction membre et fonction amie

Si un opérateur doit absolument recevoir un type de base en 1er argument, il ne peut pas être défini comme une fonction membre (puisque celle-ci reçoit implicitement un premier argument de type sa classe)

L'opérateur = ne peut être surdéfini que par une fonction membre.

## Leçon 9 : Agrégation et composition

**Prérequis :**

Leçons précédentes, Programmation I et II

**Objectif du chapitre :**

Réutiliser des objets par composition - agrégation

**Durée :** 1 séances (1h30)

**Eléments du Contenu :**

Objet membre d ‘une classe

Construction et destructions d’objets composés

## 9-1- Introduction :

L'agrégation est une relation « composé-composant » dans laquelle les objets représentant les composants d'une chose sont associés à un objet représentant l'assemblage (agrégation)

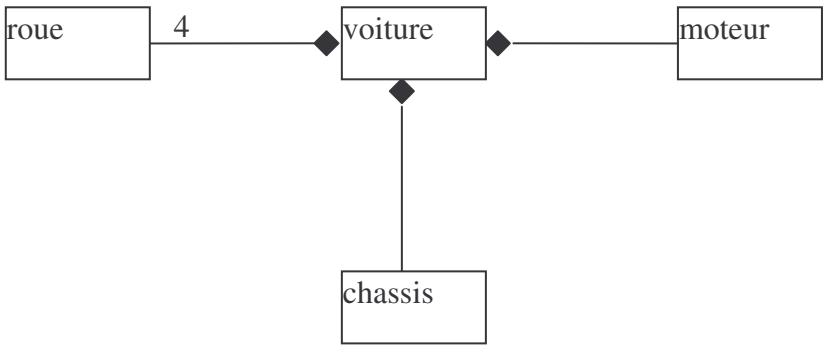
**Exemple**


La notion de composition est très proche de l'agrégation. La composition est une agrégation réalisée par valeur. Tout se passe comme si les classes composantes étaient des attributs de la classe composée.

**Exemple :**

Une voiture est composée d'un châssis, un moteur et quatre roues.

Elle se note :




**Exemple :**

```
class A {…}
class B
{
string nom; // string classe prédéfinie
…
}
class C
{
A a1, a2 ;
B* pb;
…
}
```

Un C est composé de deux A et un B

## 9-2 Objets membres d'une classe:

Il est tout à fait possible qu'une classe possède un membre donné, lui-même de type classe. Par exemple :

Ayant définit :

```
class point
{
private :
int x,y ;
public :
…
deplace(int,int)
void affiche() ;
}
```

Nous pouvons définir :

```
class cercle
{
point centre ;
int rayon ;
public :
void affrayon() ;
….
}
```

Si on déclare alors

cercle c ;

L'objet c possède un membre privé centre, de type point. L'objet c peut accéder classiquement à la méthode affrayon() par c.affrayon()

c ne pourra pas accéder à la méthode deplacer() du membre centre car centre est privé.

Si centre était public, on pourrait accéder aux méthodes de centre par c.centre.deplace(….) ou c.centre.affiche()

**Remarque :**

La relation entre les classes cercle et point est de type relation de possession, On peut dire qu'un cercle possède un centre (de type point)

## 9-3 Mise en œuvre des constructeurs et destructeurs:

Supposons cette fois que notre classe point ait été avec un constructeur :

```
class point

{

private :

int x,y ;

public :

point (int,int) ;

}
```

Nous ne pouvons plus définir la classe cercle précédente sans constructeur. En effet, si nous le faisons, son membre centre se verrait certes attribuer un emplacement, mais son constructeur ne pourrait être appelé

Il faut donc :

1- définir un constructeur pour cercle

2- spécifier les arguments à fournir au constructeur de point : ceux-ci doivent être choisis obligatoirement parmi ceux fournis à cercle.

La définition de cercle sera la suivante :

```
class cercle

{

point centre ;

int rayon ;

public :

cercle (int, int, int) ;

….

}

cercle ::cercle (int abs, int ord, int ray) :
centre(abs,ord)

{

rayon =ray ;

}
```

Les constructeurs seront appelé dans l'ordre suivant : point,cercle. S'il existe des destructeurs, ils seront appelé dans l'ordre inverse.

## Leçon 10 : Héritage simple

**Prérequis :**

Leçons précédentes, Programmation I et II

**Objectif du chapitre :**

Mise en œuvre de la réutilisation d'objets avec le troisième principe de la POO : L'héritage

**Durée :** 1 séances (1h30)

**Eléments du Contenu :**

Notion d'héritage

Propriétés de l'héritage

Cycle de vie d'une classe dérivée

## 10-1 Notion d'héritage

L'héritage est le second des trois principes fondamentaux du paradigme orienté objet. Il est chargé de traduire le principe naturel de Généralisation / spécialisation. En effet, on peut classer de façon hiérarchique des systèmes du monde réels.

Expliquons-nous d'abord sur le terme d'héritage. Il est basé sur l'idée qu'un objet spécialisé bénéficie ou hérite des caractéristiques de l'objet le plus général auquel il rajoute ses éléments propres.

En terme de concepts objets cela se traduit de la manière suivante :

Nous associons une classe au concept le plus général que nous l'appellerons **classe de base** ou **classe mère** ou **super-classe**.

Pour chaque concept spécialisé, on **dérive** une classe du concept de base. La nouvelle classe est dite **classe dérivée** ou **classe fille** ou **sous-classe**

L'héritage dénote une relation de généralisation / spécialisation, on peut traduire toute relation d'héritage par la phrase :

**« La classe dérivée est une version spécialisée de sa classe de base »**

On parle également de relation **est-un(e)** pour traduire le principe de généralisation / spécialisation.

## 10-2 Exemple : les objets graphiques

Considérons un ensemble d'objets graphiques. Chaque objet graphique peut être considéré relativement à un point de base que nous représenterons par ses coordonnées cartésiennes X et Y.

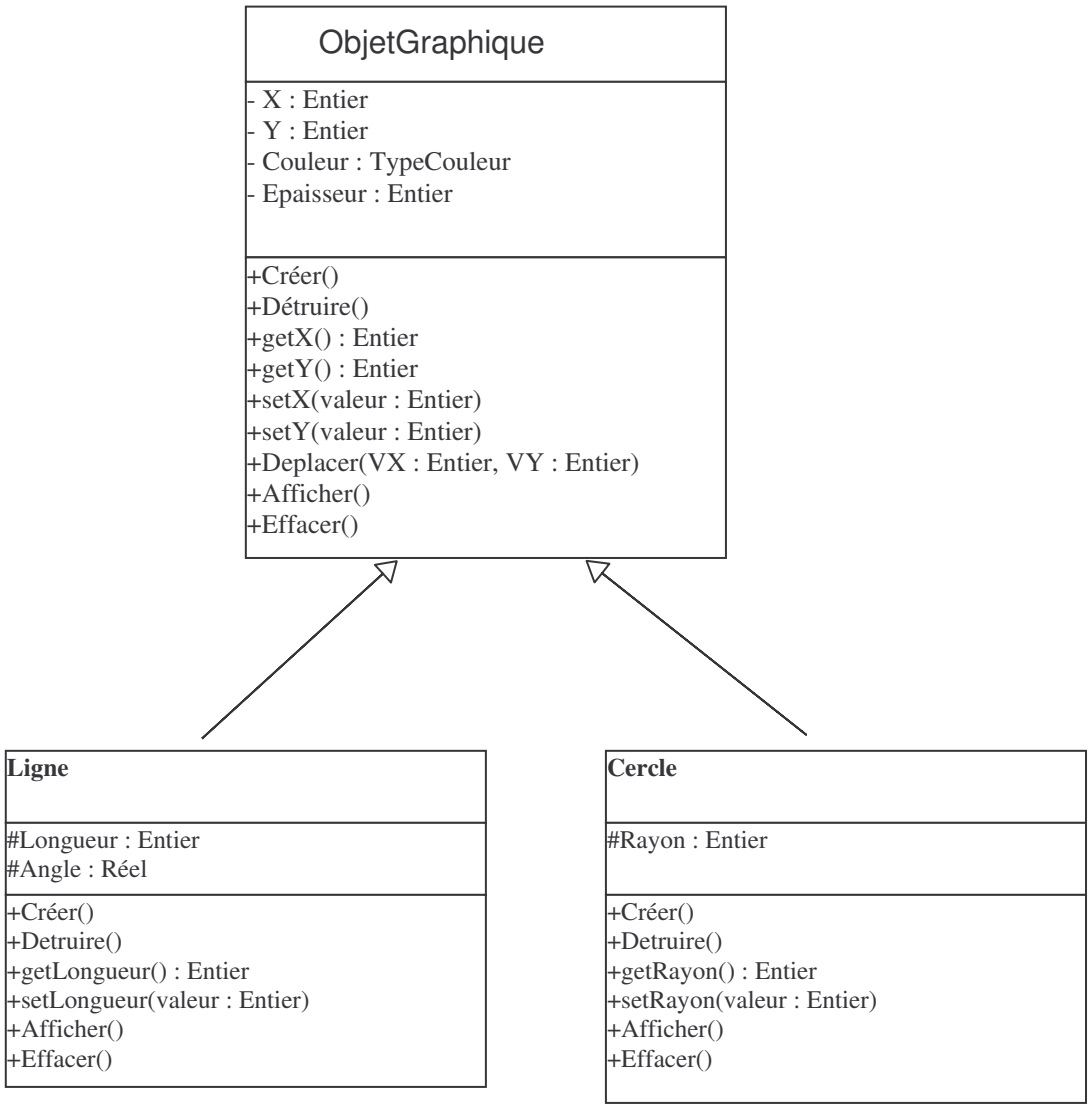
On lui associe également sa couleur ainsi que l'épaisseur du trait.

Hormis la création et la destruction d'objets, nous associons les méthodes suivantes à notre objet graphique :

- accès en lecture et en écriture des attributs
- affichage
- effacement
- déplacement d'un objet.

Rajoutons deux classes spécialisées : Ligne et Cercle. Chacune d'entre elles rajoute ses attributs propres : le rayon pour le cercle, la longueur et l'angle pour la ligne. Ainsi, les méthodes Ligne et Cercle disposent de leurs attributs propres qui traduisent leur spécialisation et des attributs de la classe de base qui sont hérités.

Les méthodes de la classe de base sont également héritées. Les classes Ligne et Cercle n'ont pas, par exemple, à fournir de code pour la méthode getX() chargée de renvoyer la valeur de l'attribut X. En revanche, elles sont libres de rajouter les méthodes qui leur sont propres, par exemple, des méthodes pour accéder aux attributs supplémentaires.




**Remarques :**

Cet exemple, aussi simple, soit-il illustre bien les avantages que l'on retire à utiliser de l'héritage :

Le code est de taille plus faible car l'on factorise au niveau des classes

Généraliser les comportements communs à toute une hiérarchie

Au niveau des classes dérivées, seul le code spécifique reste à écrire, il est donc inutile de réinventer la roue à chaque étape, et le développement est plus rapide

Modélisation d'une notion très naturelle permettant de créer des systèmes conceptuellement bien conçus.

**Exemple 2 :**

```
class A
{
public :
Void f();
};

class B : public A
{
public: void g();
};
…
A a ; B b;
a.f(); // legal
b.g(); //legal
b.f(); //legal
a.g(); //illegal : a n'a pas de g()
B est un A avec des "choses" en plus!
```

## 10-3 Propriétés issues de l'héritage :

- Si B hérite de A, Alors toutes les instances de B sont aussi des instances de A, et il est donc possible de faire :

  A a ; B b ; a=b ;

- Propriété conservée lorsqu'on utilise des pointeurs :

  A *pa ; B *pb = … ; pa = pb ;

  Car pointer sur B c'est avant tout pointer sur un A

  L'inverse n'est pas vrai

## 10-4 Droits d'accès aux membres :

Il y a 3 classes de membres : public, protected et private.

## 10-4-1 Cas de l'héritage public :

**Exemple**

```
class A
{
public :
int a;
void fa();
private:
int c;
protected :
int b;
};
class B : public A
{
public:
void fb();
```

```
};
```

**Remarques :**

- a est visible depuis n'importe où
- c n'est visible que dans A :: et donc pas dans B :: ni dans ::, c'est à dire que b est utilisable dans fa(), dans fb(), mais pas dans main().
- b est intermédiaire, visible dans A :: et dans B :: mais pas dans ::, c'est à dire que b est utilisable dans fa(), dans fb(), mais pas dans main().

## 10-5 Surcharge des membres :

```
Class A
{
public :
int val;
} ;
class B : public A
{
public:
double val;
} ;
…
A a ; B b ;
a.val est un int
b.val est un double
b.A ::val est un int
```

Si un membre (donnée ou fonction) de B a le même nom qu'un membre de A, Alors il le cache.

Il est possible d'utiliser A :: pour spécifier de quel membre on parle.

## 10-6 Construction d'une classe dérivée.

Comme une instance de B est avant tout une instance de A, dans le constructeur de B, on explicite la façon de créer A dans la liste d'initialisation.

```
Class A
{
int val1;
public:
A(int v1 =10) ;
};

class B : public A
{
int val2;
public:
B(int v1, int v2) ;
};
…
A::A(int v1) : val1(v1){}
B ::B(int v1, int v2) : A(v1),val2(v2)
{}
```

Le constructeur de A est utilisé avant de faire la partie {…}
Ce détail est souvent important !

## 10-7 Ordre de construction et de destruction

Les constructeurs sont appelés dans l'ordre logique de construction :
« pour faire un B, il faut déjà faire un A »
Les destructeurs dans l'ordre inverse

**Exemple :**

```
Class A
{
public:
A(){ cout<<"A::A()"<<endl;}
~A() {cout<<"A:: ~A()"<endl;}
} ;
class B : public A
{
public:
B(){ cout<<"B::B()"<<endl;}
~B() {cout<<"B:: ~B()"<endl;}
} ;
…
class C : public B
{
public:
C(int n){ cout<<"C::C() "<<n<<endl;}
~C() {cout<<"C:: ~C()"<endl;}
} ;
```

On obtient à l'exécution :

```
A::A()
B::B()
C::C() 10
C:: ~C()
B:: ~B()
A:: ~A()
```

## 10-8- Autres possibilités de dérivation :

Autre que la possibilité de dérivation publique il existe les dérivations privées et protégées

| Classe de base | | | dérivée Publique | | dérivée protégée | | dérivée privée | |
|---|---|---|---|---|---|---|---|---|
| initial | fma | user | nouveau | user | nouveau | user | nouveau | user |
| public | o | o | public | o | protégé | n | privé | n |
| protégé | o | n | protégé | n | protégé | n | privé | n |
| privé | o | n | privé | n | privé | n | privé | n |

fma : accès fonctions membres ou amies de la classe ?

user : accès utilisateur

initial : statut initial de la classe de base

nouveau : statut de la classe dérivée

# Leçon 11 : Polymorphisme

**Prérequis :**

Leçons précédentes, Programmation I et II

**Objectif du chapitre :**

Mise en œuvre de la réutilisation d'objets avec l'introduction du principe du polymorphisme

**Durée :** 1 séances (1h30)

**Eléments du Contenu :**

Notion de Fonction virtuelle

Le Polymorphisme

Destructeur virtuel

Notion de fonction virtuelle pure et classe Abstraite

## 11-1- Etude de Cas

Etudions l'exemple suivant :

```
class A
{
private :
…..
public :
….
void f()
….
} ;
void A ::f()
{
cout<< "A::f()" <<endl;
}
```

```
class B: public A
{
…
public:
void f();
…
};
void B::f()
{
cout<< "B::f()" <<endl;
}
```

```
main()
{
A a; B b; A *p;
P= &a;p->f(); // affiche A::f()
p=&b;p->f();  //  affiche  A::f()
aussi
}
```

**Remarques :**

Le choix de la fonction est conditionné par le type de **p** connu au moment de la **compilation**

Puisque p est un A* alors il utilise la méthode `A ::f()`

Il sera mieux que le deuxième appel écrit `B ::f()` ? Puisque p contient un B

→ Solution : Fonction virtuelle

## 11-2- Fonction virtuelle :

```
class A
{
private :
…..
public :
….
```

```
     virtual void f()
     ….
     } ;
     void A ::f()
     {
     cout<< "A::f()" <<endl;
     }
```

```
     class B: public A
     {
     …
     public:
     void f();
     …
     };
     void B::f()
     {
     cout<< "B::f()" <<endl;
     }
```

```
     main()
     {
     A a; B b; A *p;
     P= &a;p->f(); // affiche A::f()
     p=&b;p->f();  //  affiche  B::f()
aussi
     }
```

**Remarques :**

Le choix de la fonction est **maintenant** conditionné par le type exact de l'objet pointé par p connu au moment de **l'exécution** puisque p nous emmène sur un B* alors on utilise `B ::f()`

## 11-3 Polymorphisme :

Le C++ permet que des objets de types différents répondent différemment à un même appel de fonction : c'est le **polymorphisme.**

**En pratique** : Déclarer **virtual** la fonction concernée dans la classe la plus générale de la hiérarchie d'héritage. (Dans l'exemple précédent A)

Toutes les classes dérivées qui apportent une nouvelle version de f() utiliseront leur fonction à elles.

Il reste évidement possible d'appeler la fonction f() de A en spécifiant p->A ::f(), mais à ce moment là pourquoi avoir utilisé une fonction virtuelle !?

## 11-4 Destructeur virtuel :

Considérons l'exemple suivant :

```
     class A
     {
       int *p;
```

```
public :
  A() { p=new int[2] ;cout<<"A()"<<endl };
  ~A(){ delete [] p ; cout<<"~A()"<<endl };
} ;
class B : public A
{
  int *q;
public :
  B() { p=new int[20] ;cout<<"B()"<<endl };
  ~B(){ delete [] p ; cout<<"~B()"<<endl };
} ;

main()
{
  for (int I =0; I<4;I++)
  {A *pa = new B(); delete pa;}
}
```

Affiche ceci :

```
A() B() ~A()
A() B() ~A()
A() B() ~A()
A() B() ~A()
```

On doit faire appel au destructeur de B avant celui de A, puisque **pa** référence un ***B**

Dans ce cas on a un espace mémoire alloué non libéré : (fuite de mémoire)

→ Solution : destructeur virtuel

La classe A devient ainsi :

```
class A
{
  int *p;
public :
  A() { p=new int[2] ;cout<<"A()"<<endl };
  Virtual ~A(){ delete [] p ; cout<<"~A()"<<endl };
} ;
```

A l'exécution, on a ceci:

```
A() B()~B()~A()
A() B()~B()~A()
A() B()~B()~A()
A() B()~B()~A()
```

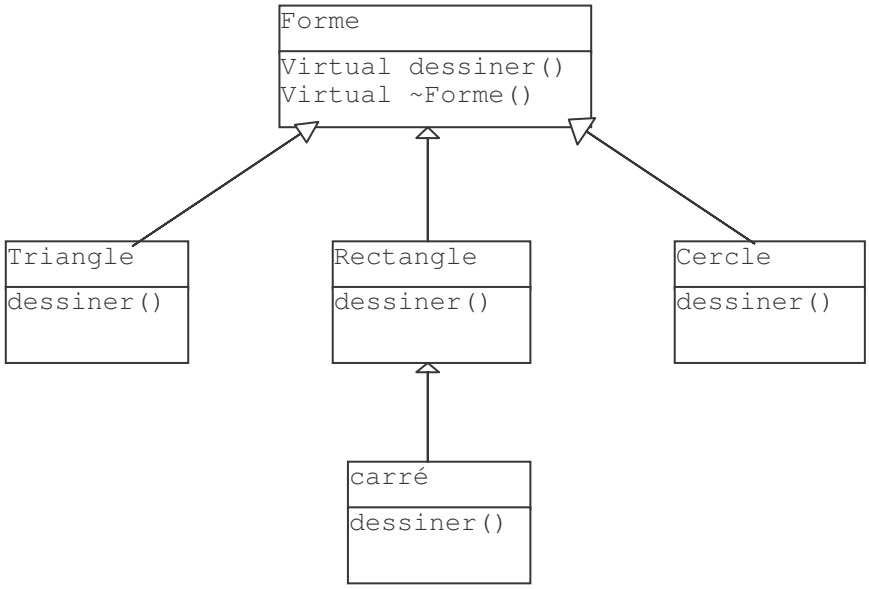
Le bon destructeur est appelé.

## 11-5 Fonction virtuelle pure – classe abstraite :

**Exemple :**

Considérons la hiérarchie suivante :

La fonction **dessiner()** est commune à toutes les classes de cette hiérarchie. Elle doit donc être attachée à la classe Forme, car toutes les autres classes sont dérivées de celle-ci.




Mais dans ce cas là, souvent on ne sais pas programmer ce que doit faire cette fonction dans le contexte de Forme : On veut juste s'assurer de sa présence dans toutes les classes filles, sans pouvoir la préciser dans la classe parente.

Voici comment déclarer Forme :

```
class Forme {
public :
virtual void dessiner() = 0;
//dessiner fait partie de forme
// mais on ne sait pas comment la programmer
} ;
class cercle : public Forme
{
private :
point centre ; double rayon ;
public :
void dessiner() { tracer_cercle(centre, rayon) ; }
} ;
```

**Remarques :**

- Cette déclaration, permet d'obliger les descendants à contenir une méthode dessiner().
- La fonction **`Forme ::dessiner()`** n'est pas définie ; elle est dite **virtuelle pure**
- La classe Forme ne peut pas être instanciée; elle est dite **abstraite**
- Une classe fille peut ne pas apporter de définition à **dessiner()** ; elle est alors aussi abstraite ( non instanciable) à son tour,etc.

- Une classe dans laquelle il n'y a plus une seule fonction virtuelle pure est dite **concrète** et devient **instanciable**.

Références Bibliographiques


- Programmer en langage C++

Claude Delannoy

Edition Eyrolles 2000

- Introduction à la conception par objet

Cours Philippe Dosch – IUT Informatique Nancy 2 – Année 2001

http://www.iuta.univ-nancy2.fr

- Cours de Programmation C++

E. Remy

IUT de Provence – Site d'Arles 2002-2003

- C++ La bible du Programmeur

Kris Jamsa, Lars Klander

Edition Eyrolles