Langages et Modèles à Objets

Mourad Oussalah

Professeur à l'Université de Nantes

Objectif du cours

- ➜ **Présenter l'approche objet et ses principaux domaines d'application**
- ➜ **Introduire les concepts**
- ➜ **Fixer la terminologie employée**

Impact des objets aujourd'hui

- Langages orientés objet pour faire de la programmation orientée objet
  - Java, C++, Smalltalk, etc.
- Méthodologies de conception orientées objet
  - UML, OMT, Booch, etc.
- Bases de données orientées objet
  - Versant, O2, ObjectStore, etc.
- Mais aussi ...
  - Générateurs d'interfaces graphiques basés sur des hiérarchies d'objets graphiques,
  - Environnements de programmation à base de composants,
  - Systèmes d'exploitation orientés objet,
  - CORBA, etc.

L'approche objet ... dans la pratique

- **Présentation des concepts objet indépendamment de tout langage de programmation, de toute méthode de conception.**
  - **Parce que ces techniques (concepts, mécanismes, etc.) sont communes à de nombreux langages, méthodes, etc.**
  - **Mais ...**
    - **Certains principes ne sont pas intégrés à tous les langages, à toutes les méthodes, etc.**
    - **Certains principes sont interprétés et / ou nommés différemment par certains langages ou méthodes.**

Historique et contexte

Les objets : une abstraction supplémentaire pour la programmation

de plus en plus d'abstraction

. programmation en langage machine

. programmation structurée

. programmation orientée objet

La programmation structurée (1)

La programmation structurée est basée sur la dichotomie :

- ➜ **Description des structures de données**
- ➜ **Codage des algorithmes de manipulation des structures de données**

Avantages :

- ➜ **Nette séparation entre les données et leur comportement**
- ➜ **Lisibilité et maintenabilité augmentées**

La programmation structurée (2)

Inconvénients :

- ➜ **Toutes les données sont mêlées**
- ➜ **Tous les comportements sont mêlés**
- ➜ **L'évolution des programmes (modification suite à une erreur de conception ou évolution des besoins) n'est pas facilitée**

Changement dans les données

Identification de tous les algorithmes utilisant ces données

Codage des modifications nécessaires dans les algorithmes

Modularité et encapsulation

L'idée innovante, issue de travaux sur les types abstraits de données, est de regrouper, dans un même module, les données et les traitements agissant sur ces données.

On parle alors d'encapsulation des données.

Ces modules constituent, en quelque sorte, des unités de programmation autonomes qu'il va s'agir de faire interagir entre elles pour constituer un programme.

Notion d'objet

La programmation orientée objet (1)

- ➜ **La programmation orientée objet a repris à son compte les principes de modularité et encapsulation.**
- ➜ **Les unités autonomes de programmation y sont désignées sous le terme générique d'objets.**

La notion d'objet est un regroupement de données et de procédures agissant sur ces données.

La programmation orientée objet (2)

➜ **La programmation orientée objet ne se résume pas à l'encapsulation des données et procédures. Elle est aussi caractérisée par l'utilisation d'un mécanisme (appelé mécanisme d'héritage) que nous présenterons plus loin.**

Remarque : Les langages orientés objet ne sont pas une solution "magique" à tous les problèmes de programmation.

Il est possible de réaliser un programme modulaire et respectant le principe d'encapsulation avec un langage structuré, comme il est possible de réaliser un programme mal conçu avec un langage orienté objet.

Les langages orientés objet fournissent simplement une abstraction de haut niveau qui permet, si le programmeur en tire parti, de réaliser plus facilement qu'avec les langages ne fournissant pas ce niveau d'abstraction, des programmes bien conçus.

De la programmation vers la conception (1)

- **L'encapsulation facilite**
  - **la mise au point des programmes (déboggage),**
  - **la maintenance des programmes,**
  - **l'évolution des programmes.**
- **L'objet est une unité "naturelle" de programmation puisqu'il regroupe les données et traitements capables d'assurer une fonctionnalité d'un programme.**

De la programmation à la conception (2)

- ➜ **Les objets sont plus qu'une unité de programmation : ils sont aussi une unité de conception.**
- ➜ **Pour concevoir une application, il est "plus facile" de la découper en termes d'objets.**
- ➜ **L'abstraction "objet" est proche du processus naturel de réflexion humaine.**

Les objets que nous percevons dans le monde réel sont des entités possédant des caractéristiques et réalisant des fonctionnalités.
On retrouve la dualité données / procédures représentant un même concept.

Exemple

➜ **Un objet `Pile`, par exemple, contiendrait :**

- ➢ **la structure de données permettant de sauvegarder les données empilées (par exemple une liste chaînée d'entiers),**
- ➢ **la valeur d'un pointeur de sommet de pile,**
- ➢ **des procédures permettant de réaliser les opérations usuelles**
  - **empiler un élément,**
  - **dépiler un élément,**
  - **savoir si la pile est vide.**

Cet objet contient, à lui seul, tout ce qui est nécessaire pour créer et utiliser des piles d'entiers. De plus, les objets utilisateurs de cette pile connaissent les trois fonctionnalités qui sont utilisables sur une pile.

Des données et des procédures

Visibilité

- **Dans un objet, les données et procédures peuvent être :**
  - **accessibles depuis d'autres objets,**
  - **accessibles seulement depuis les autres procédures de l'objet lui-même.**
- **On parle de visibilité des données et des procédures.**
- **En toute rigueur, il faudrait toujours que :**
  - **les données ne soient pas visibles,**
    - **Les données ne devraient pouvoir être lues ou modifiées que par l'intermédiaire de procédures prenant en charge les vérifications et effets de bord éventuels.**
  - **les procédures "utilitaires" ne soient pas visibles,**
  - **seules les fonctionnalités de l'objet, destinées à être utilisées par d'autres objets soient visibles.**

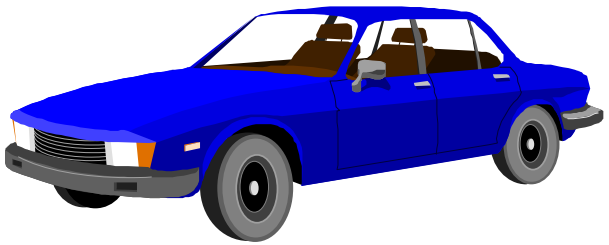
Interface et implémentation (1)

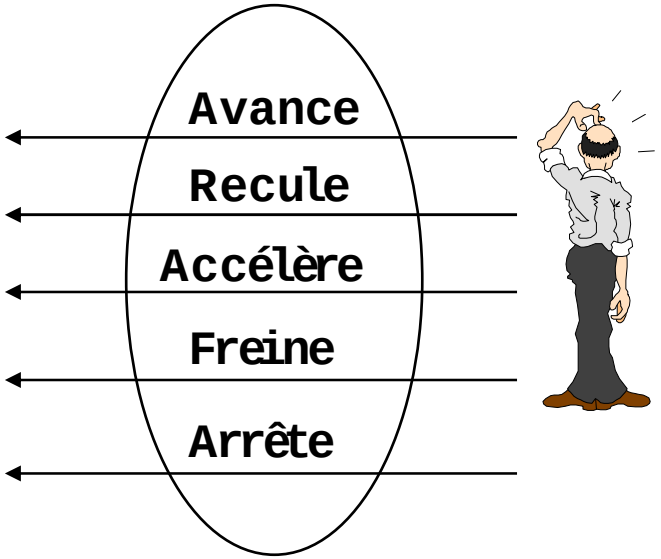
➜ **L'ensemble des procédures visibles d'un objet constitue son interface, c'est-à-dire les points d'entrée que peuvent avoir des objets extérieur pour agir sur lui.**

L'interface contient les fonctionnalités réalisables par un objet.

➜ **L'ensemble des données et des procédures invisibles d'un objet constitue son implémentation, c'est-à-dire UNE solution technique imaginée par le programmeur pour réaliser l'interface de l'objet.**

Interface et implémentation (2)





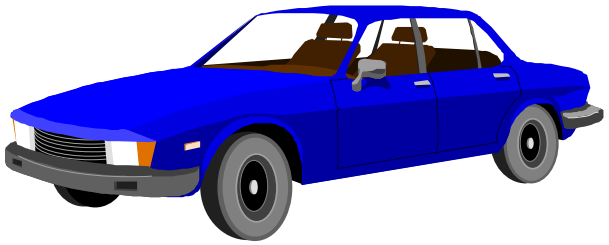
Peu importe à l'objet utilisateur quelle est la technologie utilisée pour réaliser ces fonctionnalités.

Interface de l'objet Voiture

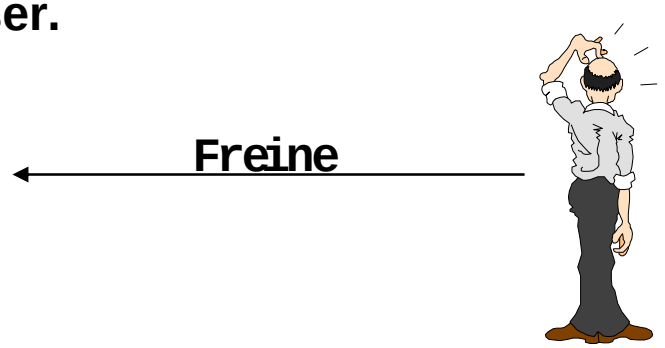
Interface et implémentation (3)

- La dichotomie interface / implémentation permet de faire évoluer plus facilement les programmes.
- Les programmes utilisant des objets respectant la dualité interface / implémentation sont, par exemple, indépendants de la façon dont sont stockées les données.

Pourvu que les fonctionnalités de l'interface ne changent pas, rien n'empêche de changer de solution technique (c'est-à-dire d'implémentation) pour les réaliser.



Freinage standard ou ABS



MO

Interface et implémentation (4)

Supposons que l'objet `L1` soit une liste d'entiers.

`L1 = ( 1 , 2 , 5 , 7 , 10 , 15 , 17 , 20 )`

Supposons aussi que l'on puisse récupérer la valeur de son 5ème élément en lui envoyant le message `Element(5)`. Ce message retourne l'entier `10`.

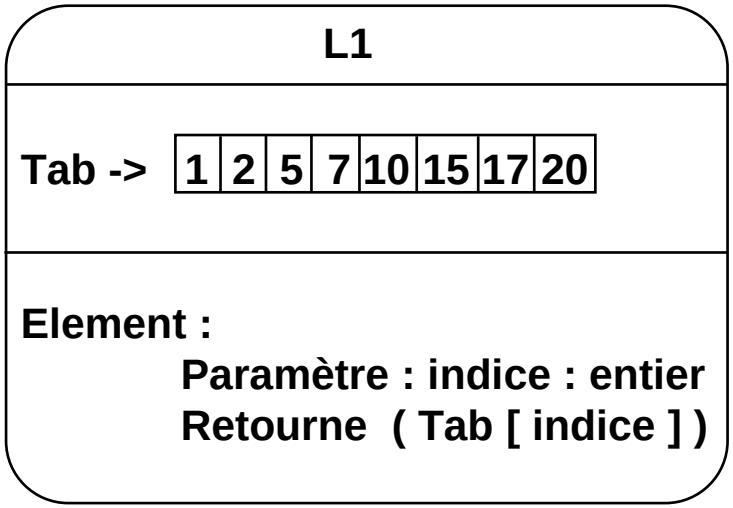
`Element` est donc un élément de l'interface de la liste.

Nous allons montrer, en proposant deux implémentations possibles de la liste, qu'à partir du moment où on utilise une fonctionnalité de l'interface, un changement dans l'implémentation de l'objet liste n'affecte pas les programmes utilisant cette liste.

Interface et implémentation

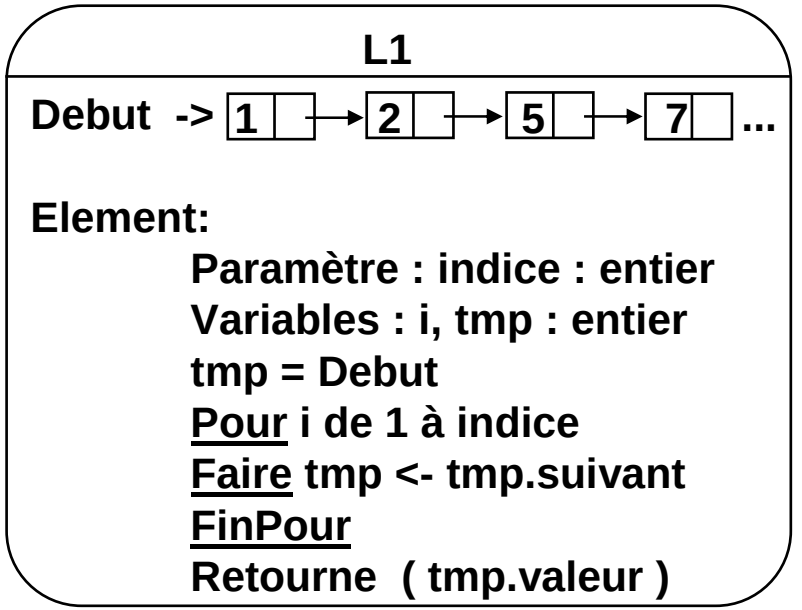
Première implémentation :

La liste `L1` est un objet dont la partie données contient un tableau d'entiers et dont la partie procédures contient la fonctionnalité `Element (x)`.



Deuxième implémentation :

La liste `L1` est un objet dont la partie données contient une liste chaînée d'entiers et dont la partie procédures contient la fonctionnalité `Element (x)`.



Classes

- **Les objets sont créés à partir d'une description générique des données (nom, type) et une définition des fonctionnalités qui agissent sur ces données (nom, signature et code) appelée classe.**

- **La notion de classe peut être considérée comme une extension de la notion de type utilisateur (exemple : structure en C).**

- **La classe est, en quelque sorte, un moule à partir duquel sont créés les objets.**

Instances et Instanciation

- **Le processus de création d'un objet à partir d'une classe s'appelle instanciation.**
- **L'instanciation est le processus qui consiste à donner une valeur à l'ensemble des données définies dans sa classe.**
- **Les objets sont aussi appelés instances (marginalement individus).**
- **Une instsance est un représentant physique d'une classe, c'est-à-dire une valuation concrète des données.**

Attributs

➜ **Les données d'une classe sont appelées attributs.**

➜ **La classe définit le nom et le type des attributs.**

➜ **Les instances contiennent une valeur pour chaque attribut.**

➜ **D'autres termes sont plus marginalement utilisés :**

- ➢ **données membres,**
- ➢ **propriétés.**

Méthode / Message

- **Les procédures d'une classe sont appelées méthodes.**
- **Les méthodes de l'interface d'une classe constituent le protocole de communication des instances de cette classe avec les autres instances de l'application.**
- **On dit que les instances communiquent par envois de messages. On parle d'instance émettrice (ou d'utilisateur) et d'instance réceptrice d'un message.**
- **A la réception d'un message, le langage orienté objet cherche si il y a une méthode qui permet de répondre à ce message (dans un certain ordre pour les cas où il y en aurait plusieurs).**

Exemple : une classe ...

- Une classe `Parallélépipède` possède les attributs réels :
  - `longueur`,
  - `largeur`,
  - `Profondeur`.
- et les méthodes :
  - `PlusGros` qui double la longueur, la largeur et la profondeur d'un parallélépipède,
  - `PlusPetit` qui divise par deux la longueur, la largeur et la profondeur d'un parallélépipède,
  - `PlusLarge` qui multiplie par trois la largeur d'un parallélépipède.

... et des instances

- Cette classe a, par exemple, deux instances :
  - `P1 : longueur = 5  largeur = 2  profondeur = 4`
  - `P2 : longueur = 6  largeur = 4  profondeur = 1`

- Les méthodes définies dans la classe Parallélépipède peuvent s'appliquer à `P1` et à `P2`.

- L 'application de la méthode `PlusPetit` sur `P1` modifie les données de `P1`. On obtient :
  - `P1 : longueur = 2,5  largeur = 1  profondeur = 2`

Héritage et mécanismes liés

Mécanisme d'héritage (1)

- **Afin d'éviter de définir plusieurs fois les mêmes informations (attributs et/ou méthodes), les langages orientés objet ont imaginé un mécanisme de "factorisation" de l'information contenue dans les classes : le mécanisme d'héritage.**
- **La notion d'héritage permet de définir une classe, la sous-classe, en en utilisant une autre, la superclasse.**

Mécanisme d'héritage (2)

- La sous-classe spécialise la superclasse : elle possède tous les attributs et toutes les méthodes de celle-ci ainsi que des attributs et des méthodes supplémentaires (qui lui sont propres).
- L'héritage évite au programmeur de spécifier une seconde fois les données et méthodes héritées. Notamment, on peut dire que l'interface de la sous-classe inclut l'interface de la superclasse.

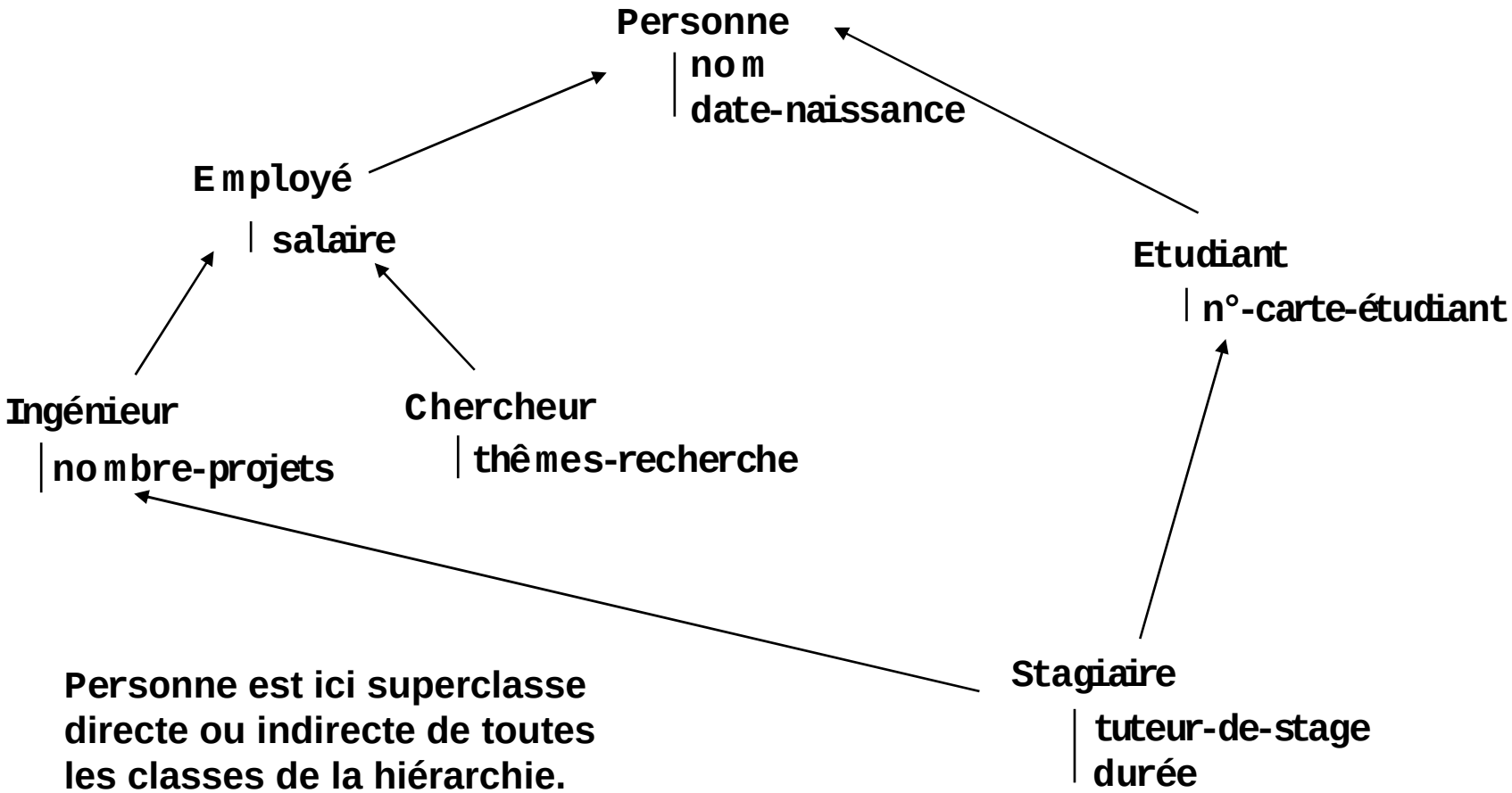
Synonymes (moins communs) pour superclasse et sous-classe :

- Classe de base et classe dérivée

Généralisation / spécialisation

- **Afin de mettre en œuvre le mécanisme d'héritage il faut que les liens entre sous-classes et superclasses soient concrétisés. On parle de relation de généralisation / spécialisation.**
- **Cette relation définit un ordre partiel sur les classes qui se traduit, d'un point de vue sémantique, par une hiérarchisation des concepts :**
  - **les concepts plus généraux sont "plus haut" dans la hiérarchie,**
  - **les concepts les plus spécifiques sont "plus bas" dans la hiérarchie.**
- **L'ajout d'attributs et de méthodes dans une sous-classe correspondent donc à une particularisation de la superclasse.**

Exemple :
Une hiérarchie de généralisation / spécialisation



Personne est ici superclasse directe ou indirecte de toutes les classes de la hiérarchie.

Héritage multiple

- Une même sous-classe peut posséder plusieurs superclasses directes.
- On parle alors d'héritage multiple (exemple : `Stagiaire` hérite de `Etudiant` et de `Ingénieur`).
- Avantages :
  - expressivité
- Inconvénients :
  - possibilité d'ambiguïtés

Redéfinition des méthodes / Masquage (1)

- **Dans une sous-classe, il est possible de définir une méthode de même nom et de même signature qu'une méthode existant déjà dans une de ses superclasses (directes ou indirectes).**
- **La méthode de la sous-classe redéfinit celle de la superclasse.**
- **La méthode de la sous-classe masque celle de la superclasse.**

Redéfinition des méthodes / Masquage (2)

- **Du point de vue d'une instance, la méthode qui sera effectivement exécutée lors d'un appel sera, par défaut, la méthode redéfinie (c'est-à-dire la "plus proche" de l'instance, la plus spécialisée donc a priori, la mieux adaptée à l'instance).**
- **Cependant, il existe généralement dans les langages orientés objet un moyen, lorsque c'est nécessaire, de déclencher la méthode de la superclasse, de façon explicite.**

Un des intérêts de la redéfinition des méthodes est d'économiser au programmeur la tâche de créer artificiellement des noms de fonctions distincts pour réaliser des opération similaires. Les programmes y gagnent bien sûr en lisibilité.

Exemple

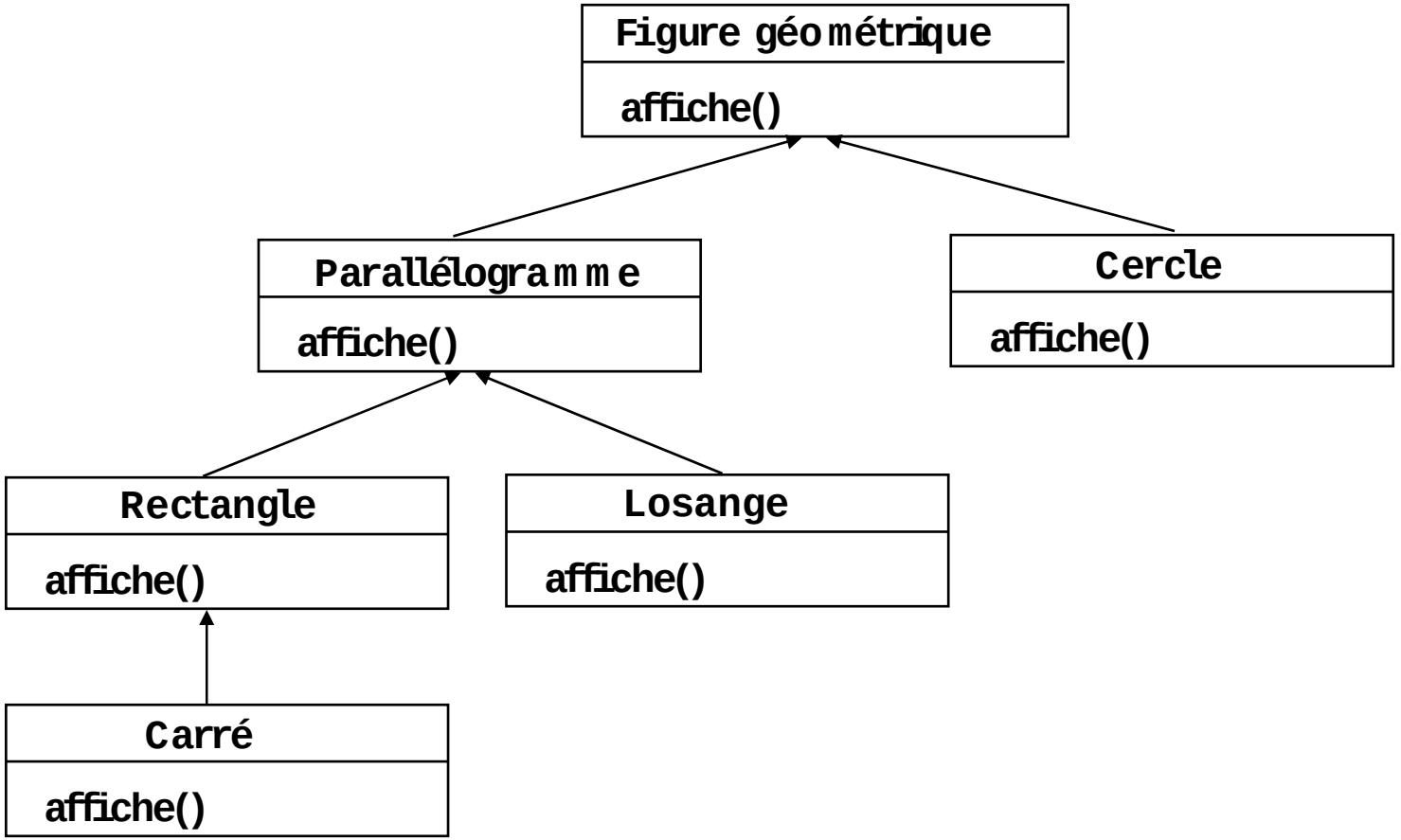


Figure géométrique
affiche()
Parallélogramme
affiche()
Cercle
affiche()
Rectangle
affiche()
Losange
affiche()
Carré
affiche()

Redéfinition ... des attributs

- En toute rigueur, la redéfinition devrait pouvoir s'appliquer sur les attributs.
- Dans la pratique, aucun langage orienté objet "grand public" n'offre cette capacité. En effet, pour des raisons de performance, l'accès aux attributs est y géré de manière statique.

"Downcasting" (typage sûr)

- **Lorsqu'on travaille avec des objets dynamiques (pointeurs ou références), il est possible de substituer, à un objet d'une classe, tout objet instance d'une sous-classe.**
  - **Cela se justifie car toutes les instances des sous-classe savent, au minimum, répondre aux mêmes messages (inclusion des interfaces).**
- **On parle de "downcasting".**
- **Ce mécanisme, combiné au mécanisme de "lookup" que l'on verra plus tard, est intéressant pour écrire du code réutilisable (exemple d'une interface graphique).**

Polymorphisme

Surdéfinition et Polymorphisme

- **Dans une même classe, plusieurs méthodes peuvent porter le même nom. Ceci était impensable dans des langages du type C.**
- **Ces méthodes doivent cependant se distinguer par leur signature (nombre et/ou type des paramètres).**
- **On dit qu'il y a surdéfinition de méthode.**
- **Cette qualité d'un langage de programmation s'appelle le polymorphisme.**

- **Le polymorphisme contribue à la lisibilité des programmes : il n'est plus la peine d' "inventer" des noms de fonctions lorsqu'elles ont des objectifs similaires.**

Invocation des méthodes (1)

- **L'envoi d'un message provoque le déclenchement d'un mécanisme de recherche (lookup) de la méthode qui va devoir être exécutée.**
  - **Cette recherche est fonction de la signature de la méthode, c'est-à-dire du type de l'objet récepteur, du nombre et du type des paramètres.**

Invocation des méthodes (2)

- **Cette recherche peut être réalisée avant l'exécution (à la compilation, pour les langages compilés). On parle alors de recherche statique.**
  - **Inconvénient : Dans ce cas, la recherche ne se fait pas sur le type réel des objets mais le type déclaré (cas des pointeurs).**
- **Cette recherche peut être réalisée à l'exécution, généralement dans les langages interprétés. On parle alors de recherche dynamique.**
  - **Avantage : La recherche peut se faire sur le type réel des objets.**
  - **Inconvénient : Plus lourd à gérer donc plus lent.**
- **Certains langages compilé permettent, dans une certaine mesure, des branchements dynamiques.**

Le langage C++

Introduction

MO

Contenu de cette partie

- ➜ Objectifs de cette partie
- ➜ Introduction
- ➜ Historique du langage C++
- ➜ C++, un langage orienté-objet ?

Objectif de cette partie du cours

- ➜ **Apprendre la programmation en langage C++.**
  - ➢ **introduction sur la programmation orientée objet afin de situer les solutions proposées par C++,**
  - ➢ **syntaxe et particularités du langage.**
- ➜ **Pré-requis pour ce cours**
  - ➢ **Notions et pratique du langage C.**

Introduction

- **C++ : un langage de programmation compilé,**
  - **conçu dans les années 1980,**
  - **par Bjarne Stroustrup de AT&T Bell Laboratory.**
- **C++, comme son nom l'indique, est construit à partir de la syntaxe du langage C.**
  - **L'objectif de B. Stroustrup dans la conception du langage C++ était d'introduire le concept de classe, issu d'autres langages, dans le langage C qui était un langage très utilisé.**
  - **Il souhaitait de plus, rester aussi compatible que possible avec le C classique.**

Historique

- **Le C++ a connu, depuis, un succès considérable et quelques évolutions (en particulier l'héritage multiple, la gestion des exceptions, les patrons ou " templates ", etc.).**
- **Il a fait l'objet d'une normalisation ANSI et ISO, ce qui permet d'augmenter la portabilité des programmes (c'est-à-dire la compilation du même code C++ avec des compilateurs différents et sur des architectures différentes).**

C++, un langage orienté-objet ?

- **C++ résulte de l'introduction de concepts objet (notions de classe, héritage, encapsulation, polymorphisme, etc…) dans un langage de programmation structurée.**
- **C++ n'est pas un langage orienté-objet à part entière car il n'est pas conçu pour appliquer exclusivement les concepts de la programmation orientée-objet : étant compatible avec C, il est toujours possible de faire, en C++, uniquement de la programmation structurée.**
- **L'avantage de cette philosophie est de permettre la réutilisation de code C. L'inconvénient est qu'il est possible de ne pas utiliser les concepts de C++ qui y ont été ajoutés pour améliorer la façon de programmer.**

De C à C++

MO

Contenu de cette partie

- C++, un C ++
- Nouveautés syntaxiques
- Les commentaires de fin de ligne
- Les déclarations de variables
- Notions sur les entrées / sorties conversationnelles

C++, un C ++

- C++ a repris toute la syntaxe du langage C
  - blocs, instructions, notion d'identificateur,
  - structures de contrôle,
  - désignation des types de base,
  - etc.
- C++ y a ajouté quelques éléments syntaxiques parmi lesquels
  - ceux liés aux objets, que nous verrons plus tard dans le cours,
  - les autres, qui ne sont pas liés aux objets, et qui apportent des fonctionnalités supplémentaires au langage.

Nouveautés syntaxiques

- **Voici une liste des apports syntaxiques majeurs de C++, hormis les concepts objets :**
  - **le commentaire de fin de ligne,**
  - **déclarations de variables "plus souples",**
  - **la notion de référence,**
  - **les arguments par défaut pour les fonctions,**
  - **la surdéfinition de fonctions,**
    - **deux fonctions "à la C" peuvent avoir le même nom si leur signature diffère.**
  - **les opérateurs `new` et `delete`,**
    - **créés pour les objets, ces opérateurs sont aussi valables pour les types de base**
  - **le spécificateur `inline`,**
  - **les entrées / sorties conversationnelles.**

Commentaires de fin de ligne

- // est une nouvelle forme de commentaires.
- // indique que la fin de la ligne (ce qui est entre les symboles // et la fin de la ligne) est un commentaire.
- Le commentaire /*...*/ du C est toujours utilisable en C++.

- Exemple :

```
int cptr;      // Commentaires sur ce compteur
```

Déclarations de variables

- Il n'est plus obligatoire de les regrouper en début de bloc. Leur portée est limitée à la partie comprise entre leur déclaration et la fin du bloc courant.
- Exemple :

```
main ()
{
cout << "Début de ce programme principal qui serait incorrect
   en C.\n";
int n=5;
for (int i=0; i<5; i++) cout << "i = " << i << "\n";
}
```

Les entrées / sorties conversationnelles

- C++ possède de nouvelles possibilités d'entrées / sorties par rapport à C. Elles sont plus simples à utiliser (pas besoin de formater).
- Il est toujours possible d'utiliser les routines C comme `printf` et `scanf`.
- C++ utilise la notion de flot. `cout` est le flot de sortie standard, `cin` le flot d'entrée standard. L'opérateur << permet d'écrire dans un flot. L'opérateur >> permet de lire dans un flot.
- L'utilisation des E/S spécifiques au C++ nécessite l'inclusion du fichier en-tête `iostream.h`

Ecriture sur la sortie standard cout (1)

```
#include<iostream.h>
#include<string.h>
main ()
{
    int n = 25;
    char c = 'a';
    char *ch;
    ch = new char[20];
   ch = strcpy(ch, "bonjour");
    cout << "valeur de n : " << n << "\n";
    cout << "valeur de c : " << c << "\n";
    cout << "chaîne ch : " << ch << "\n";
    cout << "adresse de ch : " << (void*)ch << "\n";
}
```

Ecriture sur la sortie standard cout (2)

➜ L'exécution de ce programme principal donne :

```
valeur de n : 25
valeur de c : a
chaîne ch : bonjour
adresse de ch : 0x19700b4
```

Lecture sur l'entrée standard cin

- Le flux d'entrée standard s'utilise de la même façon.
- Exemple d'utilisation de cin

```
#include"iostream.h"
main ()
{
  int n;
  char[10] ch;
  cout << "Entrez au clavier un entier et un mot"
       << "\n";
  cin >> n >> ch;
}
```

- Si l'utilisateur tape `10 bonjour` :
  - n aura comme valeur `10`
  - ch aura comme valeur `"bonjour"`

Classes et objets en C++

MO

Contenu de cette partie

- ➜ Déclaration d'une classe
- ➜ Création d'un objet et accès à ses membres
- ➜ Affectation d'objets
- ➜ Notions de constructeur et de destructeur
- ➜ Membres données statiques
- ➜ La classe comme composant logiciel

Terminologie

- ➜ **Les méthodes d'une classe sont appelées fonctions membres.**
- ➜ **Les attributs d'une classe sont appelées données membres.**

Déclaration d'une classe

```
class <nom_classe>
    {
    private :
        // données membres et fonctions membres privées
    public :
        // données membres et fonctions membres publiques
    };
```

Spécificateurs d'accès

- **Les mots clé private et public sont des spécificateurs d'accès.**
- **Un membre d'une classe (donnée membre ou fonction membre) peut être :**
  - **soit privé c'est-à-dire que son nom ne peut être utilisé que par les fonctions membres de la classe dans laquelle il est déclaré,**
  - **soit public, c'est-à-dire que son nom peut être utilisé par n'importe quelle fonction (y compris, par exemple, par une fonction membre d'une classe autre que celle dans laquelle il est déclaré).**

Remarques sur les spécificateurs d'accès

- Par défaut, si aucun spécificateur d'accès n'est précisé, les membres d'une classe sont privés.
- Les spécificateurs d'accès peuvent apparaître plusieurs fois et dans n'importe quel ordre dans la déclaration d'une classe :

```
class X
        {
        public :
        …
        private :
        …
        public :
        …
        };
```

Spécificateurs d'accès et encapsulation

- **C++ permet de violer le principe d'encapsulation.**
- **Pour respecter l'encapsulation :**
  - **les données membres doivent être privées. Il doit être impossible d'y accéder directement depuis des fonctions extérieures à la classe.**
  - **les fonctions membres doivent être divisées en deux catégories :**
    - **les fonctions d'interface avec l'extérieur de la classe (ses fonctionnalités), publiques,**
    - **les fonctions utilitaires, qui ne servent qu'à programmer de façon modulaire les fonctions d'interface, privées.**
  - **Terminologie : les fonctions d'interface qui ne servent qu'à accéder aux données en lecture ou en écriture sont appelées accesseurs et modifieurs.**
    - **elles sont souvent nommées get...() ou set...()**

Exemple de déclaration d'une classe

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

déclaration des données membres

déclaration des fonctions membres, c'est-à-dire, énoncé de leur signature

Définition d'une fonction membre

- Dans l'exemple ci-dessus, nous avons simplement déclaré les trois fonctions membre de la classe Point. Il faut par ailleurs les définir.
- La définition d'une fonction se fait de la façon suivante :

```
<type_retour> <nom_classe>::<nom_fonction>(<arguments>)
{
   // corps de la fonction
}
```

Exemples de définitions de fonctions membre

➜ **Les fonctions membre de la classe Point se définissent donc de la façon suivante.**

```
void Point::initialise (int abs, int ord)
{
 x=abs;
   y=ord;
}

void Point::deplace (int dx, int dy)
{
   x = x + dx;
   y = y + dy;
}

void Point::affiche ()
{
   cout << "Je suis en " << x << " " << y << "\n";
}
```

Déclaration d'un objet objet du type d'une classe

➜ Si on désire déclarer un objet de nom <nom_objet> comme étant du type <nom_classe>, on procède comme pour une déclaration de variable C :

<nom_classe> <nom_objet>;

➜ Exemple :

```
Point p1;
```

Déclaration d'un tableau d'objets ou d'un pointeur sur un objet

➜ Les tableaux d'objets et les pointeurs sur un objet se déclarent comme en C.

➜ Exemples :

```
Point tab[10]; // tableau de 10 objets de type Point
Point *ptr; // pointeur sur un objet de type Point
```

Accès aux membres (données et fonctions) d'un objet

- La syntaxe est la même que pour l'accès aux champs d'une structure C.
  - Utilisation des opérateurs . et →

```
<nom_objet>.<nom_donnée>;
<nom_objet>.<nom_fonction> (<arguments>);
<nom_pointeur_objet>→<nom_donnée>;
<nom_pointeur_objet>→<nom_fonction> (<arguments>);
```

Exemple

```
void main ()
{
    Point a, b;
    a.initialise (5,2);
    a.affiche ();
    a.deplace (-2,4);
    a.affiche ();
    b.initialise (1,-1);
    b.affiche ();
}
```

➜ **L'affichage à l'exécution de ce programme est :**

```
Je suis en 5 2
Je suis en 3 6
Je suis en 1 –1
```

Affectation d'objets (1)

➜ Si on a déclaré :

```
class Point
{
        int x;
        public :
        int y;
        …
};

Point a, b ;
```

➜ l'instruction `b = a;` provoque la recopie des valeurs des membres x et y de a dans les membres correspondants de b.

➜ Cela reviendrait à écrire :

```
b.x = a.x ;
b.y = a.y ;
```

➢ Attention : Comme x est un membre privé, la première affectation doit être écrite au sein d'une fonction membre de la classe `Point` pour être valide.

Affectation d'objets (2)

- **Remarque sur l'opérateur = : L'utilisation de l'opérateur = semble naturelle. Son rôle est cependant évident seulement dans des cas simples.**
- **Nous verrons des cas plus compliqués. En particulier lorsqu'un objet comporte des pointeurs sur des emplacements dynamiques, la recopie avec l'opérateur = ne concernera cette partie dynamique que si on a réalisé la surdéfinition de l'opérateur.**
  - **problème de la copie superficielle par rapport à la copie profonde.**

Notion de constructeur (1)

- Lorsque nous avons défini la classe `Point`, nous avons prévu une fonction membre pour initialiser les données membres d'un objet de classe `Point`.
- Les données membres non initialisées et, pire encore, l'oubli d'allocation dynamique de mémoire peuvent causer des erreurs de programmation. C++ prévoit un mécanisme pour ce faire : le constructeur.
- Un constructeur est une fonction membre qui sert à initialiser un objet et qui est appelée automatiquement à chaque création d'objet.

Notion de constructeur (2)

- **Si le programmeur définit pas de constructeur pour une classe, il en existe un par défaut, sans arguments, qui alloue seulement la place mémoire pour les données membres.**
  - **Attention : Si une donnée membre est un pointeur, seule la place du pointeur est réservée, pas celle de l'objet pointé !**
- **Lorsqu'il faut allouer dynamiquement de la mémoire pour un objet, le programmeur utilise le constructeur pour réaliser ces allocations.**

Notion de destructeur

- D'une manière similaire, un objet pourra posséder un destructeur, fonction membre appelée automatiquement à la destruction d'un objet.
- Le but de l'utilisation du destructeur est de libérer la mémoire qui a été allouée dynamiquement à la création de l'objet.

Conventions pour les constructeurs et les destructeurs

- Par convention, le constructeur porte le même nom que la classe et le destructeur le nom de la classe précédé de ~.
- De plus, les constructeurs et destructeurs n'ont pas de type de retour.
- Remarque : Les constructeurs et destructeurs sont généralement publics. S'ils étaient privés, il ne serait pas possible d'instancier la classe.

Exemple de constructeur (1)

➜ Remplaçons la fonction membre initialise par un constructeur.

```
class Point
{
        int x;
        int y;
        public :
                Point (int, int);  // constructeur
                void deplace (int, int);
                void affiche ();
};

Point::Point (int abs, int ord)
{
        x = abs;
y = ord;
}
```

Exemple de constructeur (2)

```
void main ()
{
    Point a (5,2); //remplace Point a; a.initialise(5,2);
    a.affiche ();
    a.deplace (-2, 4);
    a.affiche ();
}
```

➜ **Remarquer le passage des paramètres du constructeur à la création d'un objet statique.**

Cas d'un constructeur sans argument

➜ **Dans le cas d'un constructeur sans argument, il faut omettre les parenthèses.**

```
class Point
{
    Point ()
    {
    …
    }
};

main ()
{
    Point a;              // OK
    Point a ();   // erreur !
}
```

Exemple de destructeur (1)

```
class Point
{
    int x ;
    int y ;
    public :
        Point (int, int);
        void deplace (int, int);
        void affiche ();
        ~Point ();
} ;

Point::~Point ()
{
    cout << "Le destructeur de Point \n";
}
```

Exemple de destructeur (2)

```
main ()
{
    Point a (5,2);
    a.affiche ()
}
```

➜ **Exécution**

```
Je suis en 5 2
Le destructeur de Point
```

Variables de classe

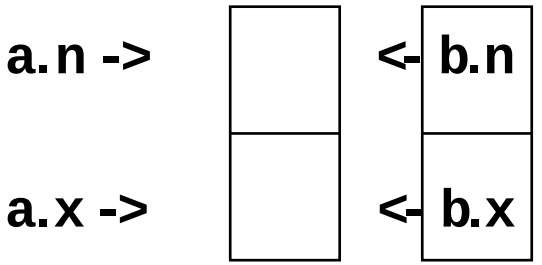
- **Il peut être nécessaire de définir des données qui caractérisent une classe.**
- **Leur valeur est alors unique pour tous les objets du type de cette classe.**
- **Ces données sont des variables de classe. C++ les appelle les membres données statiques.**
- **De façon à ne pas avoir à propager les modifications de valeur qui seraient faites sur une instance ainsi que par économie de mémoire ces données ne sont stockées qu'une seule fois, pour l'ensemble des objets du type d'une classe.**

Stockage des données "classiques"

- A chaque donnée d'un objet, correspond une place mémoire qui permet de stocker sa valeur.
- Exemple :

```
class ex1
{
    int n;
    float x;
}

ex1 a, b;
```

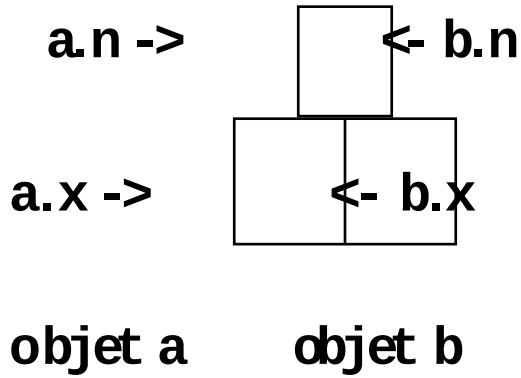


Stockage des données membres statiques

➜ **Exemple :**

```
class ex2
{
    static int n;
    float x;
}

ex2 a, b;
```



➜ **Grâce au mot réservé static, n a la même valeur pour tous les objets de la classe ex2. Pour tous ces objets, n n'est stocké qu'une seule fois.**

Initialisation des membres données statiques

- Elle ne peut ni avoir lieu dans le constructeur, ni lors de la déclaration.
- Un membre donnée statique doit être initialisé explicitement, en dehors de toute méthode, par une instruction telle que :

  `<type_donnée> <nom_classe>::<nom_donnée = <valeur>;`
- Exemple : `int ex2::n = 5;`
- La modification de la valeur d'un membre donnée statique, peut se faire, comme pour une donnée classique, dans une méthode.
  - Les spécificateurs d'accès s'appliquent comme pour les données classiques.

La classe comme composant logiciel

- **Pour que la classe puisse être réutilisée en tant que composant logiciel par d'autres applications, le programmeur de la classe fournit :**
  - **1 module objet résultant de la compilation de la définition de la classe,**
  - **1 fichier en-tête contenant uniquement la déclaration de la classe.**

Exemple de la classe Point

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

fichier Point.H

```
#include <iostream.h>
#include "Point.H"
Point::Point (int abs, ind ord)
{
   x = abs;
   y = ord;
}
void Point::deplace (int dx, int dy)
{
   x = x + dx;
   y = y + dy;
}
void Point::affiche ()
{
   cout << "Je suis en " << x
        << " " << y << "\n";
}
```

fichier Point.C à compiler pour
obtenir le module objet de la classe Point

Utilisation d'une classe dans un programme

➜ **Pour utiliser la classe `Point` dans un programme, il faut :**

- ➢ **inclure le fichier en-tête**

```
#include "Point.H"
```

- ➢ **faire l'édition de liens du module objet de la classe `Point` et du programme utilisateur.**

Protection contre les inclusions multiples

- Afin d'éviter les inclusions multiples, les en-têtes doivent être protégées par des instructions de compilation conditionnelle.
- Exemple :

```
#ifndef POINT_H
#define POINT_H
    // déclaration de la classe Point
#endif
```

- Remarque concernant les membres donnée statiques : Les initialiser de préférence dans le fichier contenant la définition de la classe.

Les propriétés des fonctions membres

Contenu de cette partie

- ➜ Surdéfinition de fonctions membres
- ➜ Arguments par défaut
- ➜ Modes de passage de paramètres à une fonction membre
- ➜ Autoréférence
- ➜ Fonctions membres statiques
- ➜ Fonctions membres constantes

Surdéfinition des fonctions membres

- En C++, il est possible de définir plusieurs fonctions membres d'une classe possédant le même nom.
- On parle de surdéfinition des fonctions membres.
- La sélection de la bonne fonction se fait selon le nombre ou le type des arguments.

Exemple (1)

```
class Point
{
    int x, y;
    public :
        //...
        void affiche ();
        void affiche (char*);
};

void Point::affiche ()
{
    cout << "Je suis en : "<< x <<" " << y << '\n";
}
```

Exemple (2)

```
void Point::affiche (char* message)
{
    cout << message;
    affiche ( );
}

main ()
{
    Point a;
    a.affiche ();
    a.affiche ("Bonjour");
}
```

Arguments par défaut (1)

- **Il est possible de donner des valeurs par défaut à des arguments d'une fonction membre.**
  - **Si l'argument n'est pas précisé lors de l'appel de la fonction, il prend la valeur par défaut.**
  - **Si l'argument est précisé lors de l'appel de la fonction cela se passe de façon habituelle.**
- **Restriction : Les arguments qui peuvent prendre une valeur par défaut doivent être les derniers dans la liste des paramètres.**

Arguments par défaut (2)

```
class Point
{
  int x, y;
    public :
      //...
       void affiche (char* = "");
}
void Point::affiche (char * message)
{
  cout << message <<" Je suis en " << x << " " << y << "\n";
}
main ()
{
  Point a (2, 5);
    a.affiche (); // message vaut "", sa valeur par défaut
    a.affiche ("Bonjour");
}
```

Fonctions membres en ligne (1)

- Pour des fonctions très courtes, il est possible d'accroître la rapidité d'exécution d'un programme en définissant ces fonctions "en ligne".
- Lors de l'appel de telles fonctions, le compilateur ne réalise pas un branchement à une procédure, il recopie le code de la fonction à la place de l'appel.
- Il existe deux techniques pour définir ce type de fonctions :
  - définir la fonction "en ligne" dans la déclaration même de la classe,
  - utiliser le mot clé `inline`.
- Les fonctions membres en ligne sont souvent utilisées pour :
  - les accesseurs et modifieurs,
  - les constructeurs et destructeurs.

Fonctions membres en ligne (2)

➜ **Exemple 1 : définition "en ligne"**

```
class Point
{
  int x, y;
  public :
    Point () {x = 0 ;  y = 0 ;}  // en ligne
    void affiche (char * = " ");
};
```

Fonctions membres en ligne (3)

➜ **Exemple 2 : utilisation du mot clé inline**

```
class Point
{
  int x,y;
   public :
     Point ();
     void affiche (char* = " ");
};

inline Point::Point () {x = 0; y = 0;}

// ...
```

Modes de transmission d'objets en argument

- En C++, il existe 3 modes de transmission d'objets en argument à une fonction :
  - transmission par valeur,
  - transmission par adresse,
  - transmission par référence.

Transmission par valeur (1)

➜ Ecrivons une fonction membre de la classe Point qui indique si deux points coïncident.

```
class Point
{
  int x, y;
   public :
     //...
     int coïncide (Point);
};

int Point::coïncide (Point pt)
{
 if ((pt.x == x) && (pt.y == y)) return 1; //points qui coïncident
 else return 0;
}
```

Transmission par valeur (2)

➜ **Utilisation :**

```
main ()
{
int egaux;
Point a (3, 5), b(4, 9);
egaux = a.coïncide (b);
}
```

➜ **Ce mode de passage est peu recommandé. En effet, une copie locale du paramètre est réalisée lors de l'appel. Il peut y avoir des problèmes si certaines données de l'objet sont des pointeurs (copie superficielle).**

Transmission par adresse (1)

```
class Point
{
  //...
   public:
     int coïncide (Point *);
};

int Point::coïncide (Point *ptr)
{
   if (( ptr -> x == x) && (ptr -> y == y )) return 1;
   else return 0;
}
```

Transmission par adresse (2)

➜ Utilisation :

```
main ()
{
  int egaux;
   Point a(4, 5), b(4,5);
   egaux = a.coïncide (&b); // plus difficile à
                                    // utiliser
}
```

Transmission par référence (1)

- En C++ il existe un moyen de transmission de paramètres qui n'existait pas en C : la transmission par référence.
- La transmission par référence est une transmission par adresse prise en charge par le compilateur.

Transmission par référence (2)

```
class Point
{
  int x, y;
  // ...
     public :
        int coïncide (Point &);
};

int Point::coïncide (Point & pt)
{
  if ( (pt.x == x) && (pt.y == y) ) return 1;
    else return 0;
}
```

Transmission par référence (3)

➜ **Utilisation :**

```
main ()
{
   Point a (4, 9), b(3, 7);
  int egaux;
     egaux = a.coïncide (b);
}
```

➜ **Ce mode de transmission de paramètres est à la fois rigoureux et simple à utiliser. A privilégier.**

Autoréférence

- **Une fonction membre d'une classe peut avoir besoin de posséder une référence sur l'objet ayant appelé la fonction.**
  - **Exemple : pour ajouter l'objet à une liste de pointeurs**
- **this désigne l'adresse de l'objet qui a appelé la méthode dans laquelle il est utilisé.**

Exemple d'utilisation de l'autoréférence

```
void Point::affiche ()
{
  cout << "Adresse : " << this
       << "Coordonnées : " << x << " " << y << "\n";
}
```

Fonctions membres statiques (1)

- **Nous avons déjà vu que les membres données statiques étaient des données propres à une classe, identiques pour tous les objets de cette classe.**
- **De la même façon, il existe des fonctions membres d'une classe ayant un rôle totalement indépendant d'un quelconque objet. Ce serait par exemple le cas de fonctions agissant uniquement sur des données statiques.**
- **Ces fonctions sont appelées fonctions membres statiques ou méthodes de classe.**

Fonctions membres statiques (2)

```
class Exemple
{
  static int ctr;
  public :
    Exemple ();
    ~Exemple ();
    static void compte ();
};

int Exemple::ctr = 0; //initialisation du membre statique
```

Fonctions membres statiques (3)

```
Exemple::Exemple ()
{
  cout << "Construction. Il y a maintenant " << ++ctr
       << "objets \n";
}
Exemple::~Exemple ()
{
  cout << "Destruction. Il y a maintenant " << --ctr
       << " objets \n";
}
void Exemple::compte ()
{
  cout  << "Compte affiche qu'il y a " << ctr
      << " objets \n";
}
```

Fonctions membres statiques (4)

```
void fct()
{
  Exemple x,y;
}

main ()
{
  Exemple::compte();
  Exemple a;
  Exemple b;
  fct();
}
```

Fonctions membres constantes (1)

- Le mot clé const (en C) peut désigner une variable dont on souhaite que la valeur n'évolue pas.
- Exemple :

```
const int n = 20;
n = 12; // erreur ...
```

- En C++, l'emploi de const se généralise aux fonctions membres d'une classe.

Fonctions membres constantes (2)

```
class Point
{
    int x,y;
    public :
      //...
      void affiche () const; // fonction utilisable sur un
            // point constant
      void deplace (...) ;  // fonction pas utilisable sur
                                    // un point constant
};

main()
{
    Point a ;     // Point "classique"
    const Point c;  // Point constant

  a.affiche ();   // OK
    a.deplace (...); // OK
    c.affiche ( );  //OK
    c.deplace (...); // erreur !!!
}
```

Construction, destruction, recopie et initialisation des objets

Contenu de cette partie

- ➜ Objets automatiques et statiques
- ➜ Objets dynamiques
- ➜ Constructeur de recopie
- ➜ Objets membres

Objets automatiques et statiques (1)

- **On appelle objets automatiques les objets créés par une déclaration :**
  - **dans une fonction**
    L'objet est créé lors de la rencontre de sa déclaration. Rappelons qu'en C++, les déclarations peuvent, contrairement à C, être situées après d'autres instructions exécutables.
  - **dans un bloc**
    En C++, comme en C, un bloc est délimité par une { et une }. L'objet est créé lors de la rencontre de sa déclaration et détruit lors de la sortie du bloc.
- **On appelle objets statiques les objets créés par une déclaration :**
  - **en dehors de toute fonction,**
  - **dans une fonction mais précédé du mot clé static.**

Objets automatiques et statiques (2)

- **Dans les deux cas, le constructeur est appelé juste après la création de l'objet, le destructeur juste avant sa destruction.**
- **Rappel : Il peut exister plusieurs constructeurs. Le constructeur exécuté est celui qui la les arguments qui correspondent avec l'appel (en nombre et en type).**

Exemple : Création et destruction d'objets automatiques (1)

```
#include <iostream.h>
class Point
{
  int x, y;
    public :
      Point (int abs, int ord)              // constructeur en ligne
      {
      x = abs ; y = ord ;
        cout << "++ construction du point : " << x << " "
          << y << "\n";
    }
      ~Point ()                  // destructeur en ligne
      {
      cout << "-- destruction du point : " << x << " "
           << y << "\ n";
      }
};
```

Exemple : Création et destruction d'objets automatiques (2)

```
Point a (1,1);                          // objet statique

main ()
{
  cout << " *** debut du main *** \n";
    Point b (10, 10);                   // objet automatique
    int i;                  // exemple de déclaration dans un bloc
    for (i = 1; i<= 3; i++)
    {
     cout << "** boucle" << i << " **\n";
       Point (i, 2*i);                  // objet automatique
    }
    cout << "*** fin du main *** \n" ;
}
```

Exemple : Création et destruction d'objets automatiques (3)

➜ **Execution**

```
++ construction du point : 11
*** début du main ***
++ construction du point : 10 10
** boucle 1 **
++ construction d'un point : 1 2
-- destruction du point : 1 2
** boucle 2 **
++ construction du point : 2 4
-- destruction du point : 2 4
** boucle 3 **
++ construction du point : 3 6
-- destruction du point : 3 6
*** fin du main ***
-- destruction du point : 10 10
-- destruction du point : 1 1
```

Objets dynamiques

- En C, l'allocation dynamique de mémoire faisait appel à des fonctions de la bibliothèque standard, dont `malloc` et `free`.
- C++ introduit deux nouveaux opérateurs pour la gestion de la mémoire : `new` et `delete`.
- Ces opérateurs sont adaptés à la gestion dynamique d'objets mais fonctionnent aussi pour la gestion dynamique des types classiques (entiers, flottants, caractères,…).
- On appelle objet dynamique un objet qui est créé dynamiquement.

Exemple de création d'objets dynamiques (1)

- Par exemple, si on a défini une classe `Point`, on peut déclarer un pointeur sur un point :

```
Point* adr ;
```

- Ensuite on peut créer dynamiquement un point et affecter son adresse à adr :

```
adr = new Point; // sans parenthèses
```

- On peut accéder aux méthodes de la classe `Point` de la façon suivante :

```
adr -> initialise (1,3);
adr -> affiche ();
```

Exemple de création d'objets dynamiques (2)

➜ De façon équivalente, on aurait pu écrire :

```
(*adr).initialise (1,3);
(*adr).affiche ();
```

➜ La suppression de l'objet se fera par :

```
delete adr;  // attention à la syntaxe
```

Fonctionnement du new

➜ **Après allocation dynamique de la place mémoire, l'opérateur new appelle un constructeur de l'objet.**

➜ **Exemple :**

```
Point* p1 = new Point (2, 5); // lance le constructeur a deux
                              // arguments
Point* p2 = new Point; // lance le constructeur sans argument
          // syntaxe equivalente : new Point ();
```

Fonctionnement du delete

➜ **L'opérateur delete appelle le destructeur juste avant de libérer la place mémoire.**

Initialisation d'objets avec recopie

- **Lorsque la valeur d'un objet doit être transmise en argument à une fonction, il y a création dans un emplacement local à la fonction d'un objet copie de l'argument effectif.**
- **Lorsqu'un objet est renvoyé par valeur comme résultat d'une fonction, il y a création, dans un emplacement local à la fonction appelante d'un objet copie du résultat.**
- **Il en va de même lorsqu'un objet est initialisé avec un objet du même type.**
- **Dans ces 3 cas, on dit qu'il y a initialisation avec recopie.**

Constructeur de recopie

- **Lorsqu'il y a initialisation d'objet avec recopie, C++ déclenche l'exécution d'un constructeur particulier dit constructeur de recopie.**
- **Si le programmeur ne fournit pas de constructeur de recopie, le compilateur en crée un par défaut qui fait la copie des données membres.**

Exemple d'utilisation du constructeur de recopie par défaut (1)

➜ Utilisation du constructeur de recopie par défaut lors de la transmission par valeur d'un objet argument.

```
class Vecteur
{
  int nbelem;
    double *adr;
    public :
       Vecteur (int);
       ~Vecteur ();
};

Vecteur::Vecteur (int n)
{
  adr = new double [nbelem = n];  // tableau de n double
    cout << "Constructeur Usuel - adr objet : " << this
        << "- adr vecteur : " << adr << "\n";
}
```

Exemple d'utilisation du constructeur de recopie par défaut (2)

```
Vecteur::~Vecteur ()
{
  cout << "Destructeur - adr objet : " << this
       << "- adr vecteur : " << adr << "\n";
  delete adr;
}

void fonction (Vecteur b)
{
  cout << "Appel de fonction\n";
}

main ()
{
  Vecteur a (5);
    fonction (a);
}
```

Exemple d'utilisation du constructeur de recopie par défaut (3)

➜ ExÉcution :

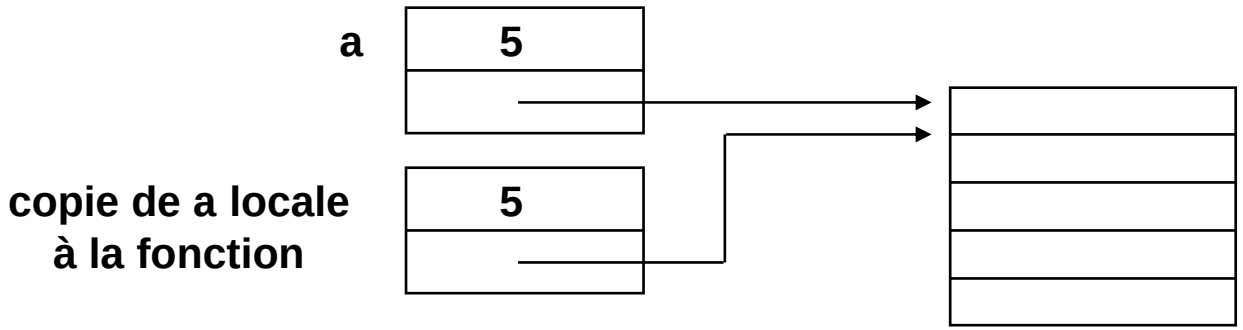
```
Constructeur usuel - adr objet: 0x40ez0ffa - adr vecteur: Ox429400004
Appel de fonction
Destructeur - adr objet: 0x40ez0ff4 - adr vecteur: 0x42940004
Destructeur - adr objet: 0x40ez0ffa - adr vecteur : 0x42940004
```

➜ Explication :

➢ L'exÉcution de fonction(a) a créé un nouvel objet dans lequel ont été recopiées les valeurs de `nbelem` et adr.

Exemple d'utilisation du constructeur de recopie par défaut (4)

➜ Au niveau mémoire, on a :



➜ A la fin de l'exécution de la fonction, il y a appel du destructeur de Vecteur pour la copie de a. Cela libère l'emplacement pointé par adr.

➜ A la fin de l'exécution du programme principal, il y a appel du destructeur de Vecteur pour a, ce qui libère le même emplacement : erreur d'exécution.

Définition d'un constructeur par recopie

- Le programmeur peut écrire un constructeur de recopie autre que celui qui existe par défaut.
- C'est un constructeur dont le seul argument est une référence vers un objet de même type.
- Exemple de signature d'un constructeur de recopie :

```
Point(Point&);      ou      Point(const Point&);
```

- C'est à ce constructeur de prendre en charge la copie superficielle et la copie profonde.

Exemple : Constructeur de recopie pour la classe Vecteur

```
Vecteur::Vecteur (const Vecteur &v) // rajouter la signature dans
                                    // la declaration de la classe
{
  adr = new double [nbelem = v.nbelem];
    int i;
    for (i = 0; i < nbelem; i++) adr[i] = v.adr[i];
    cout << "Constructeur recopie - adr objet: " << this
        << " - adr vecteur: " << adr << "\n";
}
```

➜ **Exécution du même programme principal :**

```
Constructeur usuel - adr objet: ox44ecoffa -adr vecteur: Ox469e0004
Constructeur recopie - adr objet: ox44ecoff4 -adr vecteur: Ox46a10004
Appel de fonction
Destructeur - adr objet: ox44ecoff4 - adr vecteur: Ox46a10004
Destructeur - adr objet: ox469e0004 - adr vecteur : Ox469e0004
```

Constructeur de recopie : bilan

- **Chaque objet possède son propre emplacement mémoire : aucun risque d'erreur à l'exécution.**
- **Bilan :**
  - **coder le constructeur par recopie afin de réaliser une copie profonde lors des initialisations d'objets avec recopie,**

    ou

  - **éviter absolument les situations donnant lieu à des initialisations d'objets avec recopie :**
    - **transmission d'objets par valeur,**
    - **retour d'objets par valeur,**
    - **initialisation d'objets avec recopie d'un objet du même type.**

Cascade des constructeurs dans le cas d'objets membre (1)

- Un objet peut avoir une donnée membre qui soit elle-même du type d'une classe.
- On appelle ce type de donnée un objet membre.
- Exemple :

```
class Point
{
  int x, y;
    public :
     Point ();
};
```

```
class Cercle
{
   Point centre;
  int rayon;
   public :
     Cercle ();
};
```

- A la création d'un cercle, il y aura aussi création d'un point : le constructeur de Cercle appelle automatiquement le constructeur de Point.

Cascade des constructeurs dans le cas d'objets membre (2)

- La situation est un peu plus compliquée dans le cas où on souhaite déclencher, pour l'objet membre, un constructeur avec paramètres.
- Exemple :

```
class Point
{
  int x, y;
  public :
     Point (int, int);
 };
```

```
class Cercle
{
   Point centre;
   int rayon;
   public :
      Cercle (int, int, int);
};
```

Cascade des constructeurs dans le cas d'objets membre (3)

- Le constructeur de la classe possédant un objet membre doit être écrit de façon à transmettre les bons paramètres au constructeur de l'objet membre.
- Dans notre exemple, le constructeur de `Cercle` doit être écrit de façon à permettre le passage des arguments au constructeur de son centre, qui est un objet de type `Point`.
- La définition du constructeur du cercle s'écrit de la façon suivante :

```
Cercle::Cercle (int abs, int ord, int ray) : centre (abs, ord)
  {
  // ...
}
```

nom de la donnée membre
qui est du type `Point`

Cas d'objets comportant plusieurs objets membre

➜ **Dans le cas d'objets comportant plusieurs objets membre, il est possible de déclencher tous les constructeurs avec paramètres.**

➢ **séparer les différentes listes d'arguments par des ,**

➜ **Exemple :**

```
class C
{
   A a1;
   B b;
   A a2;
   public:
     C (int, int, double, int, int);
};

C::C (int n, int p, double x, int q, int r) : a1 (p), b (x, q) , a2 (r)
{
  // ...
}
```

Surdéfinition d'opérateurs

Contenu de cette partie

- ➜ **Surdéfinition d'opérateurs**
- ➜ **Forme canonique d'une classe**

Surdefinition d'opérateurs

- `+ - * / = [] new delete ++` sont des exemples d'opéra- teurs (unaires ou binaires).
- `a+b` peut être une addition d'entiers, une addition de réels, etc.
- En C++, il est possible de définir ces opérateurs pour n'importe quelle classe : on parle de surdéfinition d'opérateur.
- Pour surdéfinir un opérateur (par ex : +) il faut définir une fonction dont le nom commence par operator suivi de l'opérateur (exemple : `operator+`).
- Pour surdéfinir l'opérateur + de façon à additionner des points on écrira une fonction membre de la classe `Point` de signature : `Point operator+ (Point)`

Exemple (1)

```
class Point
{
  int x, y;
    public :
       Point (int abs = 0, int ord = 0)  {...}
       Point operator+ (Point);
       void affiche ()
       { cout << "coordonnées: " << x << "" << y << "\n"; }
};

Point Point::operator + (Point a)
{
   Point p;
     p.x = x + a.x;          // on ajoute les coordonnées
     p.y = y + a.y;
     return p;
}
```

Exemple (2)

```
main ( )
{
    Point a (1,2);
   a.affiche ();
   Point b (2,5);
   b.affiche ( );
   Point c;
    c = a + b;
   c.affiche ();
   c = a + b + c;
   c.affiche ();
}
```

➜ Exécution :

```
coordonnées : 1 2
coordonnées : 2 5
coordonnées : 3 7
coordonnées : 6 14
```

➜ c = a + b;

est interprété comme

C = a.operator + (b);

Limitations à la surdéfinition d'opérateurs

- Utiliser des opérateurs existants déjà pour des types simples
  - impossible d'inventer de nouveaux symboles.
- Respecter l'arité
  - impossible de surdéfinir l'opérateur = comme opérateur unaire ou ++ comme opérateur binaire.
- Les priorités et l'associativité ne sont pas modifiables
  - * toujours prioritaire sur +
- Tous les opérateurs peuvent être surdéfinis (sauf .).
- Autres exemples :
  - () -> (cast) % << >> <= < >= > == ! = & || && += etc…

Forme canonique d'une classe (1)

- **Pour que la copie d'objets d'une classe fonctionne correctement (dans le cas où des données membre sont des pointeurs), il est conseillé de définir, au minimum, les fonctions membre suivantes :**
  - **constructeur**
  - **destructeur**
  - **constructeur de recopie**
  - **opérateur d'affectation (=)**
- **C'est ce qu'on appelle la forme canonique d'une classe.**

Forme canonique d'une classe (2)

➜ Si `T` est le nom de la classe, sa forme canonique est :

```
class T
{
 public :
     T(...);
     T(const T&);  // const protège l'argument
       // contre les modifications
     ~T();
     T& operator = (const T&);
...
};
```

La technique de l'héritage

Contenu de cette partie

- ➜ Mise en œuvre de l'héritage
- ➜ Redéfinition des fonctions membres
- ➜ Constructeurs, destructeurs et héritage
- ➜ Spécificateur d'accès protégé
- ➜ Héritage multiple

Intérêt et mise en œuvre

- Nous avons déjà évoqué, lorsque nous avons parlé de concepts objet, l'utilité de l'héritage pour réutiliser des données et des fonctions membres d'une classe.
- Pour indiquer qu'une classe <classe2> hérite d'une classe existante <classe1>, la déclaration de <classe2> doit être :

```
class <classe2> : public <classe1>
{
  ...
}
```

- On dit alors que <classe1> est la classe de base (superclasse) et que <classe2> est la classe dérivée (sous-classe).

Exemple (1)

```
class Point
{
  int x, y;
   public :
  void initialise (int, int);
  void deplace (int, int);
    void affiche ();
};
```

- On suppose que les fonctions initialise, deplace et affiche sont définies par ailleurs.
- Supposons que nous souhaitions définir une classe permettant de représenter des points de couleur.
- Pour ce faire nous devons prévoir de définir ses coordonnées et sa couleur, des méthodes pour l'initialiser, le déplacer, l'afficher, le colorier.

Exemple (2)

- Les données et fonctions membre définies pour `Point` conviennent et constituent un sous-ensemble de ce qu'il faut prévoir pour un point de couleur.
- Il n'est pas utile de tout redéfinir. Il est souhaitable d'utiliser l'héritage pour réutiliser la classe `Point` :

```
# include <string.h>
# include "Point.h"
class Pointcol : public Point
{
 char couleur [20];
   public :
     void colore (char *);
};
void Pointcol::colore (char *co)
{
 couleur = strcpy (couleur, co);
}
```

les membres publics de la classe de base seront des membres publics de la classe dérivée.

Exemple (3)

➜ **Utilisation :**

```
main ()
{
 Pointcol p;
   p.initialise (10, 20);
   p.colore ("bleu");
   p.affiche ();
   p.deplace (3,5);
   p.affiche ();
}
```

➜ **p est un point coloré. Il a accès à toutes les fonctions membres publiques de la classe Point. Il possède, comme les points, une abscisse, une ordonnée. En plus, il possède une couleur.**

Utilisation, dans une classe dérivée, des membres de la classe de base (1)

- Règle concernant les spécifications d'accès : Une classe dérivée n'a pas accès aux membres privés de sa classe de base.
- Si nous voulons, pour la classe `Pointcol`, réaliser une fonction membre d'affichage, nous ne pourrons pas écrire, car x et y sont privés :

```
void Pointcol::affichecol ( )
{
  cout << "Je suis en : " << x << " " << y << "et de couleur " << couleur <<
    "\n";
}
```

Utilisation, dans une classe dérivée, des membres de la classe de base (2)

➜ Nous définirons donc affichecol de la façon suivante :

```
void Pointcol::affichecol ()
{
 affiche ();
    cout << "et de couleur" << couleur << "\n";
}
```

➜ Remarque : Nous avons utilisé affiche "comme si" elle était directement définie dans la classe Pointcol.

Redéfinition des fonctions membre (1)

- Dans l'exemple ci-dessus un point coloré possède deux méthodes d'affichage (affiche et affichecol).
- En C++, il est possible de définir une fonction membre dans un classe dérivée qui possède le même nom et la même signature qu'une fonction membre de sa classe de base.
- On parle alors de redéfinition.
- Cette possibilité est exploitée pour définir des fonctions membres de but similaire mais plus adaptées aux classes dérivées.

Redéfinition des fonctions membre (1)

- Attention : Il devient dans ce cas moins naturel d'accéder à la fonction membre de même nom qui se trouve dans la classe de base. Il faut maintenant utiliser l'opérateur de résolution de portée ::
- Dans une fonction membre de la classe dérivée ayant redéfini <fonction>, l'appel à la fonction de la classe de base s'écrit :

```
<classe_base>::<fonction> (<arguments>);
```

- Si on dispose d'un objet du type de la classe dérivée, il est possible d'appeler sur cet objet, la fonction de la classe de base avec :

```
<objet_deriv>.<classe_base>::<fonction> (<arguments>);
```

- Dans ces deux cas <fonction>(<arguments>), aurait exécuté la fonction redéfinie dans la classe dérivée.

Exemple de redéfinition d'une fonction membre (1)

```
class Pointcol : public Point
{
  char couleur [20];
  public :
    void colore (char *); // defintion identique a celle
          // donnee precedemment
    void affiche ();
};
void Pointcol::affiche ()
{
  Point::affiche ();
  cout << "et de couleur" << couleur << "\n";
}
main ()
{
   Pointcol p;
   p.initialise (10, 20);
   p.affiche ( );
   p.Point::affiche ();
}
```

Exemple de redéfinition d'une fonction membre (2)

➜ **Exécution**

```
Je suis en : 10 20      affiche redéfinie dans Pointcol
Et de couleur bleu
Je suis en : 10 20      affiche de Point
```

➜ **Attention :**

➢ **Pour qu'il y ait effectivement redéfinition, il faut non seulement que la fonction membre ait le même nom mais aussi la même signature. Dans le cas contraire, on peut tout à fait définir une fonction membre de même nom ayant d'autres paramètres mais on ne parlera pas de redéfinition.**

Appel des constructeurs et des destructeurs
Ordre d'appel

- **Le constructeur de la classe de base est appelé automatiquement par C++ avant l'appel de celui de la classe dérivée lors de la création d'un objet de la classe dérivée.**
  - **"en descendant"**
- **Lors de la destruction d'un objet d'une classe dérivée, le destructeur de la classe dérivé est appelé et cet appel est automatiquement suivi de l'appel du destructeur de la classe de base.**
  - **"en remontant"**

Transmission de paramètres entre constructeurs (1)

- Si les constructeurs ont des paramètres, C++ fournit une façon de définir la transmission de paramètres entre les constructeurs des classes dérivées et de base.
- Exemple :

```
class Point
{
  ...
  public :
    Point (int, int);
    ...
};
```

```
class Pointcol : public Point
{
   ...
  public :
        Pointcol (int, int, int);
        ...
};
```

```
Pointcol::Pointcol (int abs, int ord, char*co) : Point(abs, ord)
{
  ...
}
```

Transmission de paramètres entre constructeurs (2)

➜ L'appel `Pointcol a (10, 15, "bleu");` déclenche :

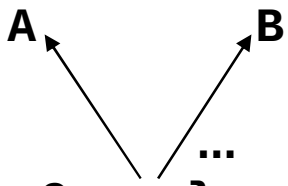
- ➢ l'appel de `Point` avec les arguments `10` et `15`
- ➢ puis l'appel de `Pointcol` avec les arguments `10`, `15` et `"bleu"`.

Contrôle des accès : le spécificateur d'accès protégé

- L'héritage nous entraîne à introduire un troisième spécificateur d'accès.
- Le mot clé est protected.
- Les membres protégés d'une classe ne sont accessibles qu'aux fonctions membres de cette classe et aux fonctions membres des éventuelles classes dérivées de cette classe.

Héritage multiple

- Il est possible, en C++, de déclarer une classe dérivée de plusieurs classes de base.
- On parle alors d'héritage multiple.
- Exemple :




```
class C : public A, public B
{
...
};
```

- La transmission de paramètres est possible entre constructeurs grâce à la syntaxe suivante :

```
C (...) : A (...), B (...)
{
  ...
}
```

Héritage multiple (2)

- Si `A` et `B` définissent une donnée membre de même nom `x`, `C` hérite des deux membres de même nom `x`.
- Au sein des fonctions membres de `C`, on les distingue grâce à l'opérateur de résolution de portée `::`

  `A::x` ou `B::x`

Fonctions virtuelles et typage dynamique

Contenu de cette partie

- ➜ **Problèmes liés au typage statique**
- ➜ **Typage dynamique et fonctions virtuelles**
- ➜ **Fonctions virtuelles pures et classes abstraites**

Problèmes liés au typage statique (1)

- En C++, un pointeur sur un type <classe1> d'objet peut recevoir l'adresse de n'importe quel objet dont le type est dérivé de <classe1>.
- Toutefois, l'appel d'une méthode pour l'objet pointé appelle la méthode qui correspond au type du pointeur et non pas à celui de l'objet pointé.
- Cela s'explique par le fait que le typage est déterminé par le compilateur et celui-ci ne peut pas savoir que l'objet pointé n'est pas du type du pointeur (typage statique).
- Pour pallier cet inconvénient C++ introduit des possibilités de typage dynamique (à l ’exécution) et la notion de fonction virtuelle.

Problèmes liés au typage statique (2)

```
class Point                    class Pointcol : public Point
{                  {
  ...                          ...
  void affiche ();             void affiche ();
};                                     };

main ()
{
  Point* a = new Point;
  Point* b = new Pointcol;
  a -> affiche ();          // appel de affiche de Point
  b -> affiche ();          // appel de affiche de Point
}
```

Fonction virtuelle

➜ La notion de fonction virtuelle permet d'apporter une solution.

```
class Point                          class Pointcol : public Point
{                       {
  ...                                ...
  virtual void affiche ();               void affiche ();
};                                         };

main ()
{
  Point* a = new Point;
  Point* b = new Pointcol;
  a -> affiche ();          // appel de affiche de Point
  b -> affiche ();          // appel de affiche de Pointcol
}
```

Deuxième exemple (1)

```
class Point
{
  int x, y;
    public :
       Point (int abs=0, int ord=0) { x=abs; y=ord; };
       void identifie () { cout << "Je suis un point \n"; };
       void affiche ()
       {
       identifie ();
         cout << "mes coordonnées sont" << x << "." << y << "\n";
    }
};
```

Deuxième exemple (2)

```
class Pointcol : public Point
{
  char couleur [20];
    public :
      Pointcol (int abs=0, int ord=0, char co[20] = "bleu") :
        Point (abs, ord)
      {
      couleur = co;
    }
      void identifie ()
      {
      cout << " je suis un point coloré de couleur "
          << couleur << "\n";
    }
};
```

Deuxième exemple (3)

```
main ()
{
    Pointcol pc (8, 6, "rouge");
    pc.affiche ();
}
```

➜ **Exécution**

```
je suis un point
mes coordonnées sont 8 6
```

➜ **Explication :**

- ➢ **affiche n'étant pas redéfinie dans `Pointcol`, l'instruction `pc.affiche ();` déclenche la méthode de la classe `Point`, où l'appel à la fonction `identifie` a déjà été compilé. La fonction membre affiche ne peut pas faire appel à la fonction `identifie` de sa classe dérivée.**

➜ **Il faudrait qu'`identifie` soit virtuelle dans la classe `Point`.**

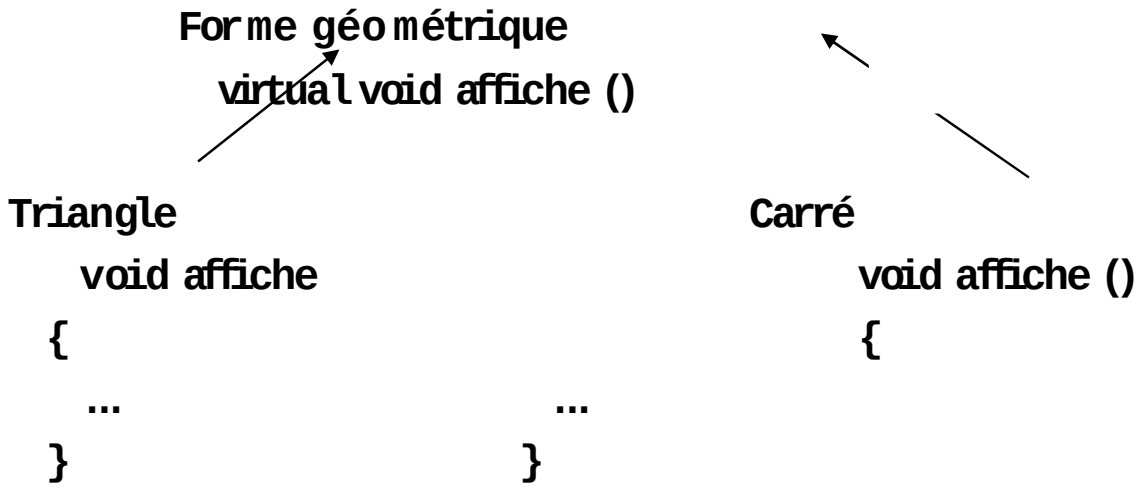
Virtualité

➜ **Remarques :**

- ➢ **un constructeur ne peut être virtuel.**
- ➢ **un destructeur peut être virtuel.**
- ➢ **une fonction virtuelle peut être redéfinie ou pas. La redéfinition peut être une fonction virtuelle ou pas.**

Fonctions virtuelles pures et classes abstraites (1)

- Il peut parfois être nécessaire de déclarer une fonction comme étant virtuelle sans pour autant avoir besoin de la définir.
- Exemple :



Fonctions virtuelles pures et classes abstraites (2)

- La déclaration de la fonction membre affiche dans la classe `Forme géométrique` comme fonction virtuelle sert, par exemple, à déclencher la fonction d'affichage du carré ou du triangle dans le cas où on disposerait d'une liste de pointeurs sur des formes géométriques.
- Par contre, le programmeur peut penser que cela n'a aucun sens de définir de fonction d'affichage dans forme géométrique.

Fonctions virtuelles pures et classes abstraites (3)

- En C++, il est possible de créer des fonctions virtuelles pures, c'est-à-dire des fonctions pour lesquelles on ne fournit pas le code.
- On dit aussi de ces fonctions qu'elles sont de valeur nulle.
- Exemple :

```
virtual void affiche () = 0; //dans la déclaration
                                    // de FormeGeometrique
```

- Cette définition de la fonction indique que celle-ci sera redéfinie dans les sous-classes de forme géométrique.
- Elle a une autre implication : il devient impossible d'instancier la classe possédant une ou plusieurs fonction virtuelle pure. On ne peut instancier que ses sous-classes.
- C'est une classe abstraite.

Patrons de classes

Contenu de cette partie

- ➜ Définition d'un patrons de classes
- ➜ Utilisation d'un patron de classes
- ➜ Patrons de classes comme composants logiciels
- ➜ Paramètres de type
- ➜ Paramètres expression
- ➜ Spécialisation d'un patron

Patrons de classes

- C++ permet de définir des classes génériques, qui peuvent être ensuite déclinées pour différents types.
- Exemple d'une classe générique liste qui se décline en :
  - liste d'entiers
  - liste de réels
  - liste de réels double précision
  - liste de points
  - ...
- Ces classes génériques sont appelées patrons de classes (templates).
- La définition de patrons permet de ne pas avoir à écrire, pour chaque type, une classe différente ayant un contenu très proche (fonction membre et données membre), à un type près.

Exemple de création et d'utilisation d'un patron de classes

- Supposons qu'on veuille que la classe `Point` puisse avoir une abscisse et une ordonnée soit entières, soit réelles, soit réelles double précision, etc.
- Il faut pour cela créer un patron :

```
template <class T> class Point
{
T x;
  T y;
  public :
    Point (T abs = 0, T ord = 0);
    void afiche ();
};
```

- Le mot clé `template` indique que c'est une classe patron. Le mot clé `class` dans `<class T>` indique que `T` est un argument de type (mais ce n'est pas forcément une classe).

Définitions des fonctions membres d'un patron de classes

➜ Dans le cas de fonction en ligne, il suffit d'écrire :

```
Point (T abs = 0, T ord = 0) { x = abs; y = ord;}
```

➜ Dans le cas de fonctions définies à l'extérieur de la déclaration de la classe, il faut écrire :

```
template <class T> void Point<T>::affiche ()
{
 cout << " coordonnées : " << x << " "
    << y << "\n" ;
}
```

Utilisation d'un patron de classes

```
Point<int> pt1;
Point<double> pt2;
```

➜ Pour passer deux paramètres aux constructeurs, nous écrirons :

```
Point<int> pt1 (3,5);
Point<double> pt2 (3.5, 4.3);
```

Patrons de classes comme composants logiciels

- **Les instructions de définition des fonctions membres d'un patron de classes sont utilisées par le compilateur pour "fabriquer", à chaque fois que nécessaire (c'est-à-dire pour chaque utilisation du patron avec un type différent), les instructions appropriées à chaque utilisation.**
- **On dit alors que les instructions constituant ces fonctions membres constituent la déclaration de la fonction (usuellement le code constitue la définition de la fonction).**
- **Conséquemment, il n'est pas possible de livrer à un utilisateur une classe patron toute compilée (déclaration + module objet). Il faut lui fournir le code de toutes les fonctions membres.**

Paramètres de type (1)

- On peut avoir un nombre quelconque de paramètres de type.
- Exemple avec trois paramètres de type :

```
template <class T, class U, class V> class essai
{
T x;      // une donnée membre x de type T
U t[5];  // un tableau t de 5 éléments de type U
...
V fm1(int, U); // fonction membre à 2 arguments
                  // de types int et U retournant
                  // un résultat de type V.
};
```

Paramètres de type (2)

➜ **Utilisation**

```
essai <int, float, int> e1;
essai <int, int*, double> e2;
essai <char *, int, Point> e3;  // s'il existe une
                                // classe Point
essai <char, Point<int>, float> e4;  // s'il existe un
                                     // patron de classes
                                                        // Point
```

Paramètres expression

- Pour définir une classe tableau qui permette de manipuler des tableaux d'objets de type quelconque et de longueur quelconque, il faudrait deux paramètres à notre patron de tableau :
  - le type des objets (paramètre de type)
  - la taille du tableau (paramètre expression)
- Exemple :

```
template < class T, int n > class Tableau
{
T tab [n];
  ...
};
```

- Utilisation : `Tableau<int, 4> t1;`
  `Tableau<Point, 10> t2;`

Spécialisation d'un patron de classes

➜ **Il est possible :**

- ➢ **de spécialiser, en se basant sur 1 ou plusieurs paramètres de type, une ou plusieurs fonctions membres, sans modifier la définition de la classe elle-même.**
- ➢ **de spécialiser la classe même (en se basant sur un ou plusieurs paramètres de type) en fournissant une nouvelle définition.**

Exemple de spécialisation d'un patron (1)

➜ **Exemple 1 : Spécialisation de la fonction d'affichage dans le cas d'un `Point<char>`**

```
template <class T> class Point
{
  T x;
  T y;
  public :
    Point ( T abs = 0 ; T ord = 0 ) {x = abs; y = ord;}
    void affiche ();
};

template <class T> void Point<T>::affiche ()
{
  cout << "coordonnées : " << x << " " << y << "\n";
}
```

Exemple de spécialisation d'un patron (2)

```
void Point<char>::affiche ()
{
  cout << "coordonnées : " << (int) x << " " << (int) y
     << "\n";
}

main ( )
{
  Point<int> pt1 (3, 5);
  pt1.affiche ();
  Point<char> pt2 ('d','y');
  pt2.affiche ();
  Point<double> pt3 (3.5, 4.3);
  pt3.affiche ();
}
```

Exemple de spécialisation d'un patron (3)

➜ Exemple 2 :

```
class Point<char>
{
  // nouvelle definition dans le cas d'un
  // parametre de type char
}
```

Fonctions amies

Contenu de cette partie

- ➜ Intérêt de la notion d'amitié
- ➜ Situations d'amitié
- ➜ Exemples de mise en œuvre

Intérêt de la notion d'amitié (1)

- Dans certaines circonstances, une fonction membre d'une classe peut avoir besoin d'accéder aux données membre privées d'une autre classe.
- Exemple :
  - classe `Vecteur`,
  - classe `Matrice`,
  - produit d'une `Matrice` par un `Vecteur`.

Intérêt de la notion d'amitié (2)

- **Solutions (?)**
  - **1 - Ecrire le produit comme fonction membre d'une des deux classes et rendre les données membre de l'autre classe publiques.**
    - **pas satisfaisant**
  - **2 - Ecrire une fonction indépendante des deux classes.**
    - **comment accéder aux données ?**
  - **3 - Ecrire dans l'une des deux classes (ou les deux) des fonctions publiques permettant d'accéder à leurs données privées (accesseurs) .**
    - **problème du temps d'exécution.**
- **Une autre solution consiste à utiliser les fonctions amies.**
- **Attention :**
  - **Ne pas utiliser trop fréquemment car une fonction amie viole l'encapsulation. Privilégier si possible les solutions de type (par exemple avec des accesseurs en ligne).**

Situations d'amitié

➜ **Il existe quatre situations d'amitié :**

- ➢ **(1) fonction indépendante amie d'une classe,**
- ➢ **(2) fonction membre d'une classe amie d'une autre classe,**
- ➢ **(3) fonction amie de plusieurs classes,**
- ➢ **(4) toutes les fonctions membres d'une classe amies d'une autre classe (notion de classe amie).**

Exemple 1 : Fonction indépendante, amie d'une classe

➜ Réécrivons la fonction coincide, non plus comme fonction membre de la classe Point, mais comme fonction indépendante, amie de la classe Point.

```
class Point
{
  ...
  public :
    friend int coincide (Point, Point);
  ...
};
int coincide (Point a, Point b)  // fonction "a la C"
{
    ...
}
```

Exemple 1 : Fonction indépendante, amie d'une classe (2)

- A noter :
  - mot clé `friend` dans la déclaration de la classe
  - `coincide` n'est fonction membre d'aucune classe.
- La fonction `coincide` accède aux données membres privées de la classe `Point` car elle est une fonction amie de cette classe.

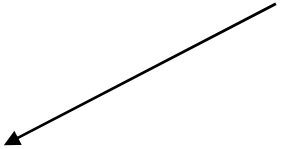
Exemple 2 : Fonction membre d'une classe, amie d'une autre classe.

```
class A
{
  ...
  public :
    friend int B::fct (char, A);
  ...
};

class B
{
  ...
  int fct (char, A);
  ...
};

int B::fct (char c, A a)
{
  ...
}
```

La fonction fct de B a accès aux membres privés de A.



(Petite) Bibliographie

(Petite) Bibliographie

- **Livres de cours et d'application du cours sur C++**
  - **Claude Delannoy - Programmer en langage C++ - 4ème édition revue et augmentée - Eyrolles 1998**
  - **Claude Delannoy - Exercices en langage C++ - Programmation orientée objet - 2ème édition revue et augmentée - Eyrolles 1996**
- **Livre de référence sur C++**
  (plus difficile d'abord mais complet - grammaire de C++)
  - **Bjarne Stroustrup - Le langage C++ - 2ème édition Int'l Thomson publishing - Traduction française - 1996**

(Petite) Bibliographie

- **Livres plus généraux sur les objets**
  - **Les langages à objets - Langages de classes, langages de frames, langages d'acteurs - G. Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre - InterEditions - Collection iia - 1989.**
  - **Les objets - M. Bouzeghoub, G. Gardarin, P. Valduriez - Eyrolles - 1997.**
  - **Ingénierie objet - Concepts et techniques - C. Oussalah et alii - InterEditions - 1997.**