SMP : Spécification et Modélisation de Programme

Classes et Objets

Myriam Servières

Ecole Centrale de Nantes
-
sur la base du cours de Guillaume Moreau

Plan

Plan

Plan

Qu’est-ce qu’un objet ?

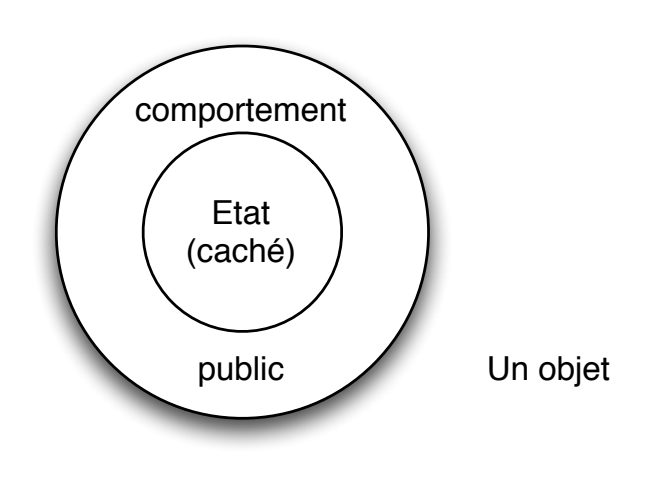
- **modélise** toute entité identifiable, concrète ou abstraite, manipulée par une application logicielle
  - une chose tangible : une ville, un étudiant, un bouton sur l’écran
  - une chose conceptuelle : une date, une réunion, un planning de réservation
- **réagit** à certains messages qu’on lui envoie de l’extérieur ; la réaction détermine le **comportement** de l’objet
- ne réagit pas toujours de la même façon à un même message ; sa réaction dépend de l’**état dans lequel il se trouve**

Un objet possède :

- une **identité unique** (pour distinguer un objet d'un autre)
- un **état interne** donné par la valeur d'un certain nombre de variables qu'on appelle des **attributs**
  - les attributs décrivent l'état de l'objet à un instant donné (cet homme mesure 1,75m, pèse 54 kg et s'appelle Charles)
  - les attributs sont typés et nommés (attribut *taille* de type réel)
- un **comportement** (capacités d'action de l'objet) donné par des fonctions qu'on appelle des **méthodes** (ou opérations)
  - les méthodes définissent ce que l'objet peut faire et comment il peut le faire

Objet = données + méthodes

- un objet est le regroupement de données (attributs) et des traitements associés (méthodes)
- Principe d'**encapsulation** : séparer les données des traitements



Plan

Principe d'encapsulation

- L'accès aux données (état) de l'objet ne peut être fait qu'au travers des méthodes
  - les données sont **privées** (cachées)
  - les méthodes **publiques** définissent l'**interface** de l'objet
- Intérêt : la modification des structures de données n'affecte pas les programmes qui utiliseront l'objet
  - masquage de l'implémentation : robustesse du code
  - facilité d'évolution du logiciel
  - pas de modification de l'interface

Plan

Déjà vus : les objets `string`

- Pas de vrai type chaîne de caractère en C
- Utilisation du type **`string`** de C++
- En réalité le type **`string`** est beaucoup plus puissant que l'utilisation faite en TP

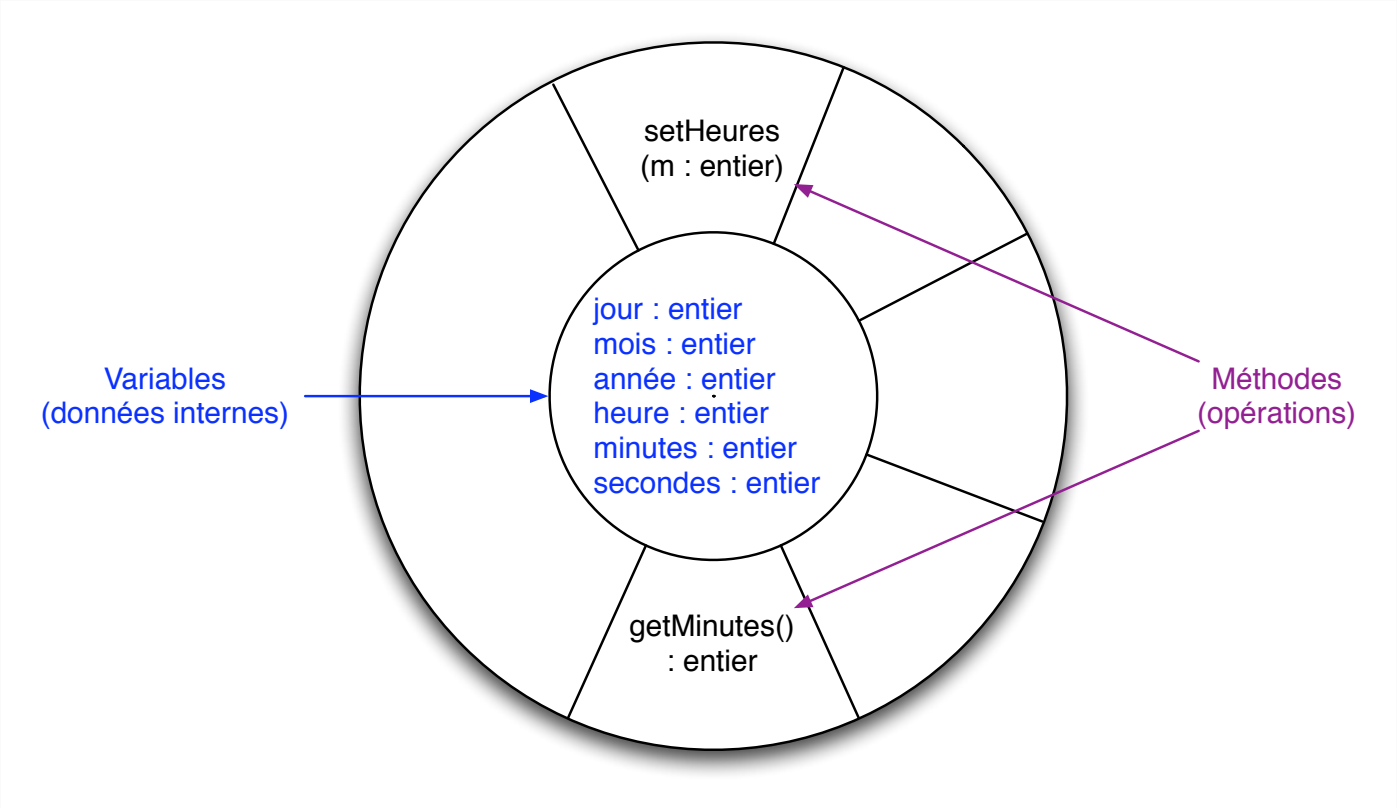
Utilisation

```
string maChaine("toto");
cout << maChaine << endl;
string maChaine2 = "titi";
cout << maChaine2 << endl;
string maChaine3 = maChaine+maChaine2;
cout << maChaine3 << endl;
```

Résultat

```
toto
titi
tototiti
```

Exemple : un objet pendule



classe Pendule (1/2)

```
1   /**
2      file : pendule.cxx
3
4      une classe de pendule toute simple en un seul fichier source. non recommandable...
5
6   */
7
8   #include <iostream>
9   using namespace std;
10
11  class pendule {
12  private:
13    int annee;
14    int mois;
15    int jour;
16    int heure;
17    int minute;
18    int seconde;
19
20  public:
21    // accesseurs en lecture
22    int getAnnee()    { return annee; }
23    int getMois()     { return mois; }
24    int getJour()     { return jour; }
25    int getHeure()    { return heure; }
26    int getMinute()   { return minute; }
27    int getSeconde()  { return seconde; }
28
```

classe Pendule (2/2)

```
29    // accesseurs en ecriture
30    void setAnnee(int annee)     { this->annee = annee; }
31    void setMois(int mois)       { this->mois = mois; }
32    void setJour(int jour)       { this->jour = jour; }
33    void setHeure(int heure)     { this->heure = heure; }
34    void setMinute(int minute)   { this->minute = minute; }
35    void setSeconde(int seconde) { this->seconde = seconde; }
36
37    // constructeur
38    pendule(int annee,int mois, int jour,int heure,int minute,int seconde) {
39      this->annee = annee;
40      this->mois = mois;
41      this->jour = jour;
42      this->heure = heure;
43      this->minute = minute;
44      this->seconde = seconde;
45    }
46
47    // affichage
48    void print() {
49      cout << jour << "/" << mois << "/" << annee << " " << heure << ":" << minute << ":" << se
50    }
51  };
52
53  int main() {
54    pendule p(2021,29,1,8,0,21);
55    p.print();
56  }
57
```

Plan

Communication par messages

Communication

Les objets **interagissent** et **communiquent** entre eux par l'envoi de **messages**

Communication par messages

Communication

Les objets **interagissent** et **communiquent** entre eux par l'envoi de **messages**

- Les méthodes publiques d'un objet correspondent aux messages que l'on peut lui envoyer

Communication par messages

Communication

Les objets **interagissent** et **communiquent** entre eux par l'envoi de **messages**

- Les méthodes publiques d'un objet correspondent aux messages que l'on peut lui envoyer
- Les messages sont caractérisés par :

Communication par messages

Communication

Les objets **interagissent** et **communiquent** entre eux par l'envoi de **messages**

- Les méthodes publiques d'un objet correspondent aux messages que l'on peut lui envoyer
- Les messages sont caractérisés par :
  - l'objet cible (récepteur) du message

Communication par messages

Communication

Les objets **interagissent** et **communiquent** entre eux par l'envoi de **messages**

- Les méthodes publiques d'un objet correspondent aux messages que l'on peut lui envoyer
- Les messages sont caractérisés par :
  - l'objet cible (récepteur) du message
  - le nom de la méthode à déclencher

Communication par messages

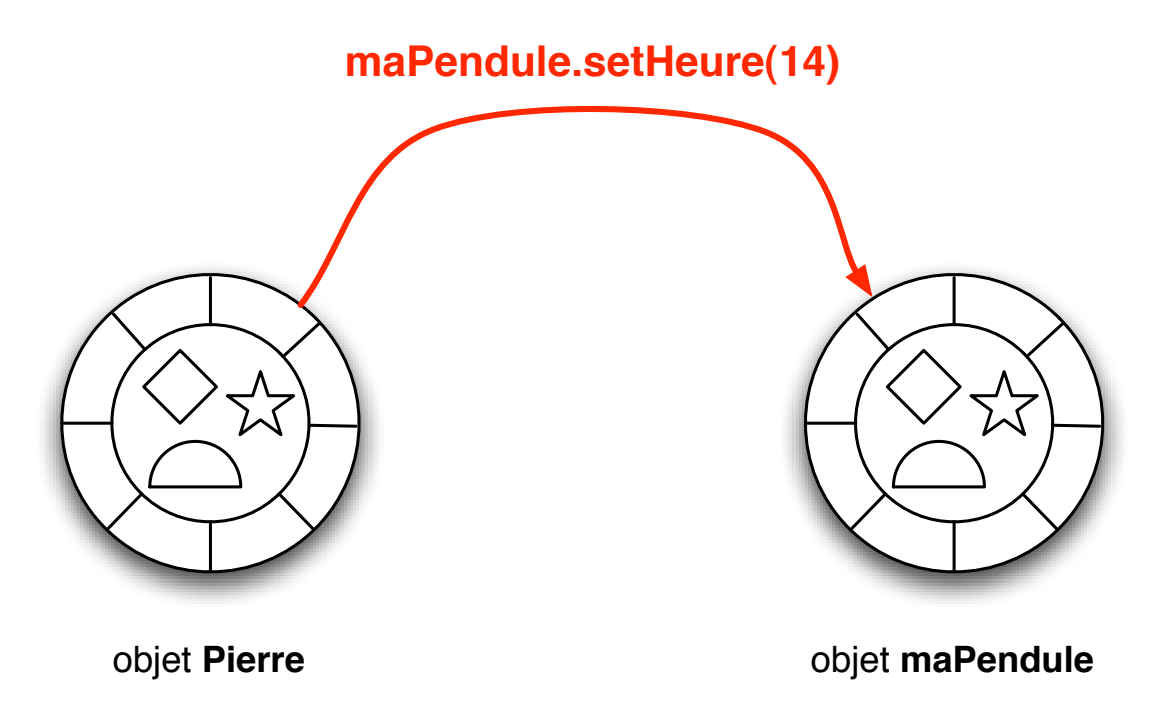
Communication

Les objets **interagissent** et **communiquent** entre eux par l'envoi de **messages**

- Les méthodes publiques d'un objet correspondent aux messages que l'on peut lui envoyer
- Les messages sont caractérisés par :
  - l'objet cible (récepteur) du message
  - le nom de la méthode à déclencher
  - les paramètres de cette méthode

Échange de messages

Les objets s'envoient des messages entre eux



Plan

Classes et instances

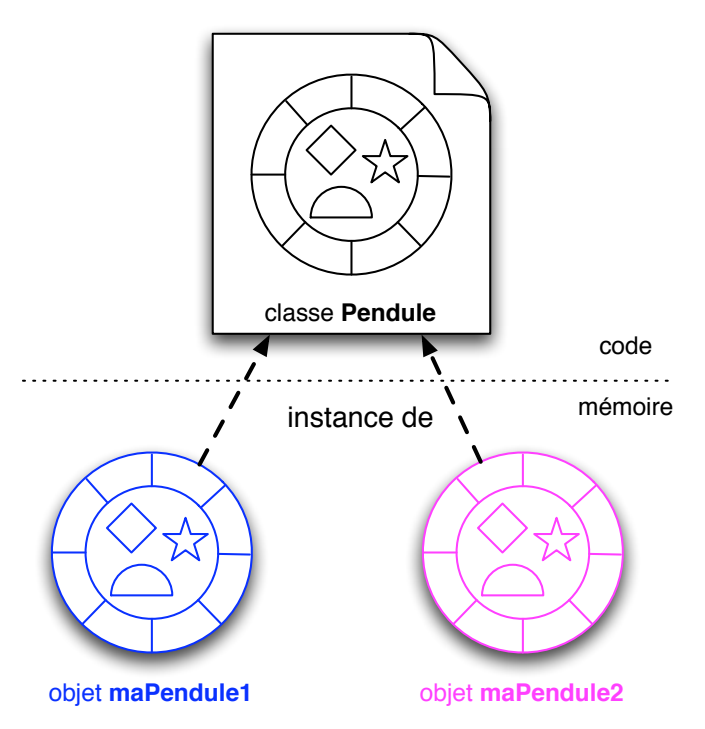
- Les objets (instances) sont créés (instanciés) à partir de «modèles» qu'on appelle des **classes**
- classe = schéma, moule, modèle d'objets décrivant :
  - partie privée : structure de données interne (attributs), corps des méthodes (algorithmes)
  - partie publique (interface) : noms et paramètres des méthodes
- la classe est un générateur d'objet : par **instanciation**, on peut fabriquer des objets (instances) respectant ce schéma/moule/modèle

Vue duale

- Une **classe** est un **modèle** de définition pour des objets
  - ayant même structure (même ensemble d'attributs)
  - ayant même comportement (mêmes opérations, méthodes)
  - ayant une sémantique commune
- Les **objets** sont des représentations **dynamiques** (instanciation) «vivantes» du modèle défini pour eux au travers de la classe
  - Une classe permet d'**instancier** (créer) plusieurs objets
  - Chaque objet est **instance** d'une (seule) classe

Classes et instances

La classe Pendule est le modèle commun des deux objets

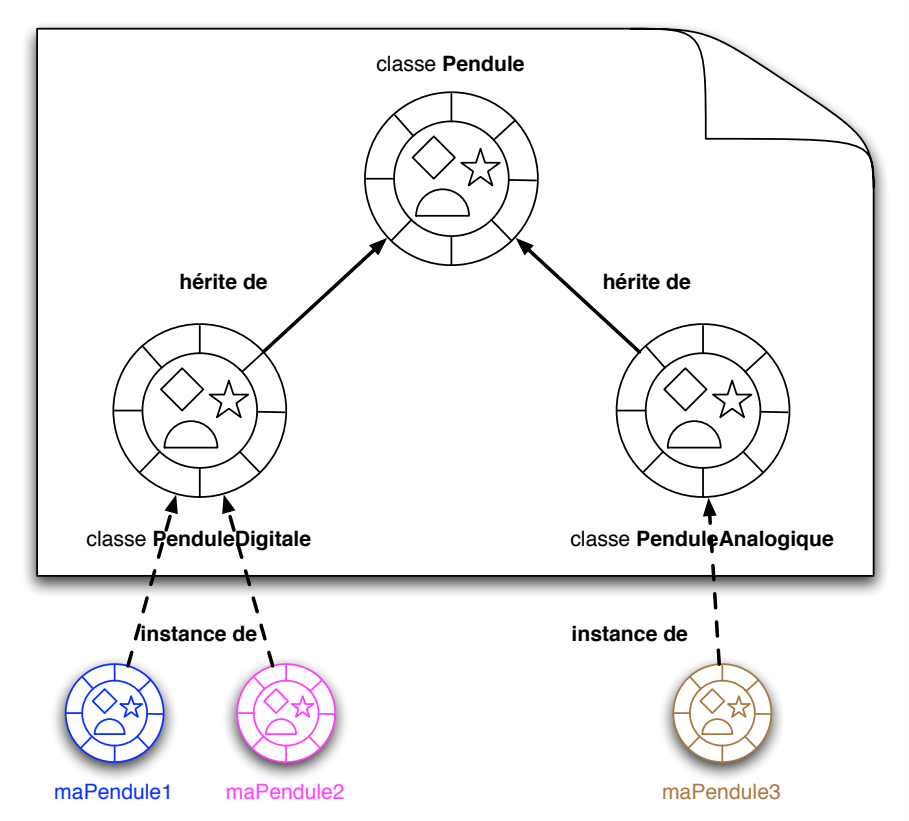


Plan

Hiérarchie de classes

- les classes peuvent être des raffinements/spécialisations de classes existantes
- Elles forment alors une **hiérarchie de classes** où chaque classe :
  - **hérite** des attributs et des méthodes de ses ancêtres (super-classe)
  - ajoute de nouveaux attributs et/ou de nouvelles méthodes
  - peut modifier ou redéfinir les méthodes héritées
- Intérêt de l'héritage :
  - réutilisation du code
  - n'ajouter que les spécialisations nécessaires

Hiérarchie de classes : exemple



Plan

Approche orientée-objet

Approche orientée-objet

- Rappel : approche procédurale
  - Définir les structures de données
  - Définir les traitements (analyse descendante)
  - Le programme principal enchaîne les traitements

Approche orientée-objet

- Rappel : approche procédurale
  - Définir les structures de données
  - Définir les traitements (analyse descendante)
  - Le programme principal enchaîne les traitements
- Approche orientée-objet
  - modéliser le monde de l'application avec des objets

Approche orientée-objet

Approche orientée-objet

- Identifier les classes

Approche orientée-objet

- Identifier les classes
- Pour chaque classe
  - Définir son interface publique (quoi)
  - Définir son implémentation (comment)

Approche orientée-objet

- Identifier les classes
- Pour chaque classe
  - Définir son interface publique (quoi)
  - Définir son implémentation (comment)
- Le programme principal
  - création (instanciation) d'objets en mémoire
  - lance l'exécution par envoi de messages aux objets créés
  - ces messages peuvent provoquer d'autres envois de messages et/ou la création d'autres objets

Pendule

pendule
- int année - int mois - int jour - int heure - int minute -int seconde

// Constructeurs
+ pendule( )
+ pendule (int annee,int mois, int jour,int heure,int minute,int seconde)
// Accesseur
+ int getAnnee()
+ int getMois()
+ int getJour()
+ int getHeure()
+ int getMinute()
+ int getSeconde()
// Mutateurs
+ void setAnnee(int annee)
+ void setMois(int mois)
+ void setJour(int jour)
+ void setHeure(int heure)
+ void setMinute(int minute)
+ void setSeconde(int seconde)
// Méthodes
+ void print()

Voiture

proposition ?

Plan

Plan

Déclaration de classe en C++

Definition

`class` *nom* {
... } ;

- Attention : le point-virgule à la fin est obligatoire !
- A l'intérieur de la classe :
  - Attributs
  - Méthodes (ou déclaration de méthodes)
  - Indicateurs de protection

Plan

Déclaration des attributs

Definition

type nom ;

- Le *type* peut être un type de base ou un type objet
- Si c'est un type objet, création d'une nouvelle instance

Exemple

```
int jour;
Pendule maPendule;
string maChaine;
int *pJour;
```

Plan

Déclaration des méthodes

- Déclaration et définition simultanées

Definition

typederetour nom (*listedeparamètrestypés*) {
```
// corps de la méthode
```
}

- Déclaration seule (classe dans un fichier .h)

Definition

typederetour nom (*listedeparamètrestypés*) ;

- Définition (dans le fichier .cxx)

Definition

typederetour `nomdeclasse::nom` (*listedeparamètrestypés*) {
```
// corps de la méthode
```
}

Plan

classe exemplemeth1 : un seul fichier

```
1  /**
2     file : exemplemeth1.cpp
3     classe simple avec methodes declarees a l'interieur de celle-ci
4  */
5
6  #include <iostream>
7  using namespace std;
8
9  class exemplemeth1 {
10 public:
11   void methode1(int a,int b) {
12     cout << "a = " << a << endl;
13     cout << "b = " << b << endl;
14   }
15
16   int somme(int a,int b) {
17     return a+b;
18   }
19 };
20
21 int main() {
22   int x,y;
23   cin >> x >> y;
24   exemplemeth1 t;
25   t.methode1(x,y);
26   cout << t.somme(x,y) << endl;
27   return 0;
28 }
```

classe exemplemeth2 : deux fichiers

```
1  /**
2     file : exemplemeth2.h
3
4     classe simple avec methodes declarees a l'exterieur de celle-ci
5
6  */
7
8  #include <iostream>
9  using namespace std;
10
11 #ifndef cpp_exemplemeth2_h
12 #define cpp_exemplemeth2_h
13
14 class exemplemeth2 {
15 public:
16   void methode1(int a,int b);
17   int somme(int a,int b);
18 };
19
20 #endif
```

classe exemplemeth2 : deux fichiers

```cpp
/**
   file : exemplemeth2.cpp

   classe simple avec methodes declarees a l'extérieur de celle-ci

*/

#include <iostream>
using namespace std;

#include "exemplemeth2.h"

void exemplemeth2::methode1(int a,int b) {
  cout << "a = " << a << endl;
  cout << "b = " << b << endl;
}

int exemplemeth2::somme(int a,int b) {
  return a+b;
}

int main() {
  int x,y;
  cin >> x >> y;
  exemplemeth2 t;
  t.methode1(x,y);
  cout << t.somme(x,y) << endl;
}
```

Exécution

Résultat d'exécution

```
(base) mservier@macbook-pro-de-myriam code % ./exemplemeth1
5
9
a = 5
b = 9
14
(base) mservier@macbook-pro-de-myriam code % ./exemplemeth2
5
9
a = 5
b = 9
14
```

1 ou 2 fichiers ?

- En réalité, la question est : déclaration seule ou déclaration+définition ?
- Réponse technique : on peut mixer les deux
- Réponse *bonnes pratiques*
  - Privilégier le découpage en plusieurs fichiers pour permettre le travail à plusieurs
  - Exception possible : les méthodes très très courtes et peu susceptibles d'être modifiées
  - Rappel : modifier un `.h` implique de recompiler tous les fichiers qui l'incluent (pas que directement)

Classe en deux parties : comment (bien) procéder ?

1. Construire le `.h`, le fichier d'entête avec attributs et déclarations de méthodes
2. Encadrer le code par des macros
   - S'assurer que le code ne sera inclus qu'une fois !

```
#ifndef cpp_exemplemeth2_h
#define cpp_exemplemeth2_h
...
#endif
```

3. ou `#pragma once` en début de chaque fichier .h
4. Inclure le `.h` dans le `.cpp`

```
#include "exemplemeth2.h"
```

5. Écrire les méthodes
   - Attention au nom d'une méthode à l'extérieur de la classe !

```
void exemplemeth2::methode1(int a, int b)
```

Plan

Classes en deux parties : problème classique

- Cas simple : une classe a besoin d'une autre
  1. Faire attention à l'ordre des `include`
  2. utiliser la *déclaration anticipé* (forward declaration)

Classes en deux parties : problème classique

- Cas simple : une classe a besoin d'une autre
  1. Faire attention à l'ordre des `include`
  2. utiliser la *déclaration anticipé* (forward declaration)
- Cas plus complexe où la déclaration anticipée est nécessaire
  - Deux classes *A* et *B*
    - *A* contenant un pointeur vers *B*
    - *B* contenant un pointeur vers *A*
  - Faire attention à l'ordre des `include` ne fonctionne plus

Cas problématique

Première classe

```
#ifndef TwoClassA_h
#define TwoClassA_h

class TwoClassA {
private:
    TwoClassB * ptrB;
public:
    void doSomething();
};


#endif /* TwoClassA_h */
```

Seconde classe

```
#ifndef TwoClassB_h
#define TwoClassB_h

class TwoclassB {
private:
    TwoclassA *ptrA;
};

#endif /* TwoClassB_h */
```

Comment inclure ces classes ?

- Si on inclut *A* avant *B*, erreur de compilation (et vice-versa) : problème d'inclusions cycliques
- Une solution
  - Utiliser la déclaration anticipée de *B* juste avant celle de *A*
    `class TwoClassB;`
  - Ainsi le compilateur sait que la classe B existe (sans avoir besoin de ses détails)
  - Fonctionne lorsqu'on n'utilise qu'un pointeur ou une référence

Déclaration de *A*

```
#ifndef TwoClassA_h
#define TwoClassA_h

// forward declaration: can use TwoClassB pointers and references only
class TwoClassB;

class TwoClassA {
private:
    TwoClassB * ptrB;
public:
    void doSomething();
};
```

Utilisation de la classe A

Au moment d'utiliser celle-ci (appel d'une fonction membre par exemple) il faudra bien avoir inclus son en-tête, mais ce sera fait dans le .cpp et non plus dans le .h, ce qui élimine le problème d'inclusion cyclique.

TwoClassA.cpp

```cpp
#include "TwoClassA.h"
// with only A declared, it is not possible to use B in a complete way

void TwoClassA::doSomething() {
    // do something
}

int main() {
    // do nothing
}
```

L'ordre des inclusions

- Quelques bons principes
  - Ne **jamais** inclure des .cpp
  - Ne pas renoncer à la modularité
  - N'inclure que ce qui est nécessaire (temps de compilation)
- Quelques questions à se poser
  - Inclure du plus général (libs standards) au plus particulier (ses propres fichiers) ? ou l'inverse ?
    - première approche : Large-Scale C++ Software Design, J. Lakos. Addison-Wesley
    - seconde approche : Google C++ Style Guide : `https://google.github.io/styleguide/cppguide.html`
  - Construire un fichier `includethemall.h` qui fait toutes les bonnes inclusions dans le bon ordre une fois pour toutes ?

Plan

Parties publiques et privées d'une classe : vue depuis C++

- On a vu que les attributs d'un objet avaient vocation à être privés, tandis que certaines méthodes ont vocation à être publiques (l'interface de l'objet ou la liste des messages qu'il peut recevoir)
- Définition des droits d'accès dans une classe
  - `public:` ce qui est déclaré après cette ligne est accessible depuis partout
  - `private:` ce qui est déclaré après cette ligne n'est accessible que depuis l'intérieur de l'objet

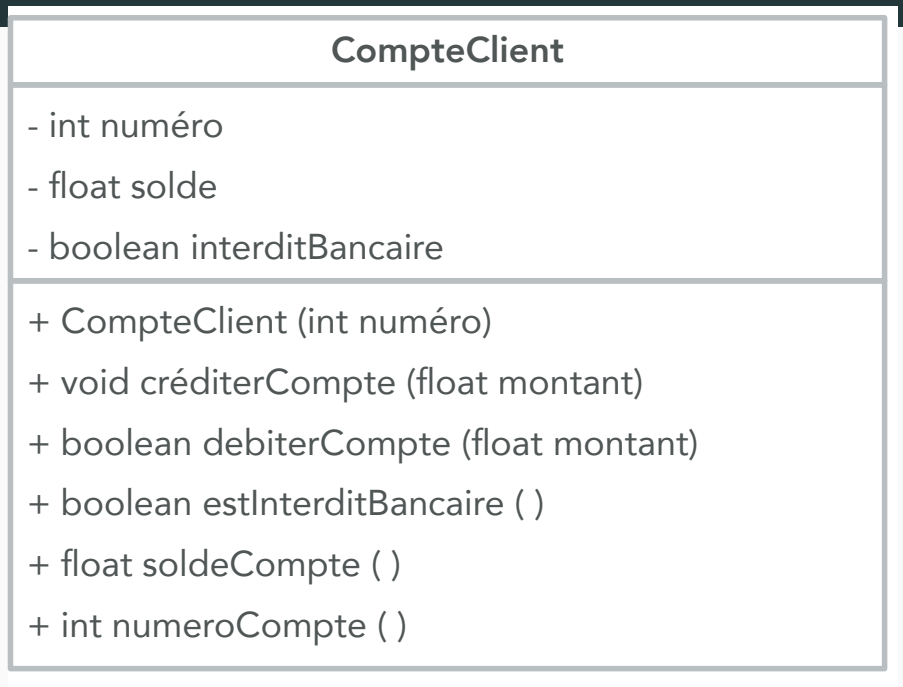
Exemple en Gestion bancaire : le compte client

- Que contient un compte (version simplifiée) ?
  - un numéro (c'est un entier)
  - un solde (c'est un réel)
  - une éventuelle interdiction bancaire (booléen)
- Que peut-on faire avec un compte bancaire ?
  - Connaître son solde
  - Déposer et retirer de l'argent
  - Savoir si un compte est interdit bancaire

Exemple en Gestion bancaire: le compte client

- Un enregistrement (`struct` en C) et des fonctions répondent-ils aux questions posées ?
- Pas complètement car:
  - il est interdit de modifier directement le solde
  - on ne doit pas modifier le numéro d'un compte client
  - seules certaines personnes sont habilitées à faire des opérations sur un compte
- d'où la protection des données proposées par les mécanismes objets

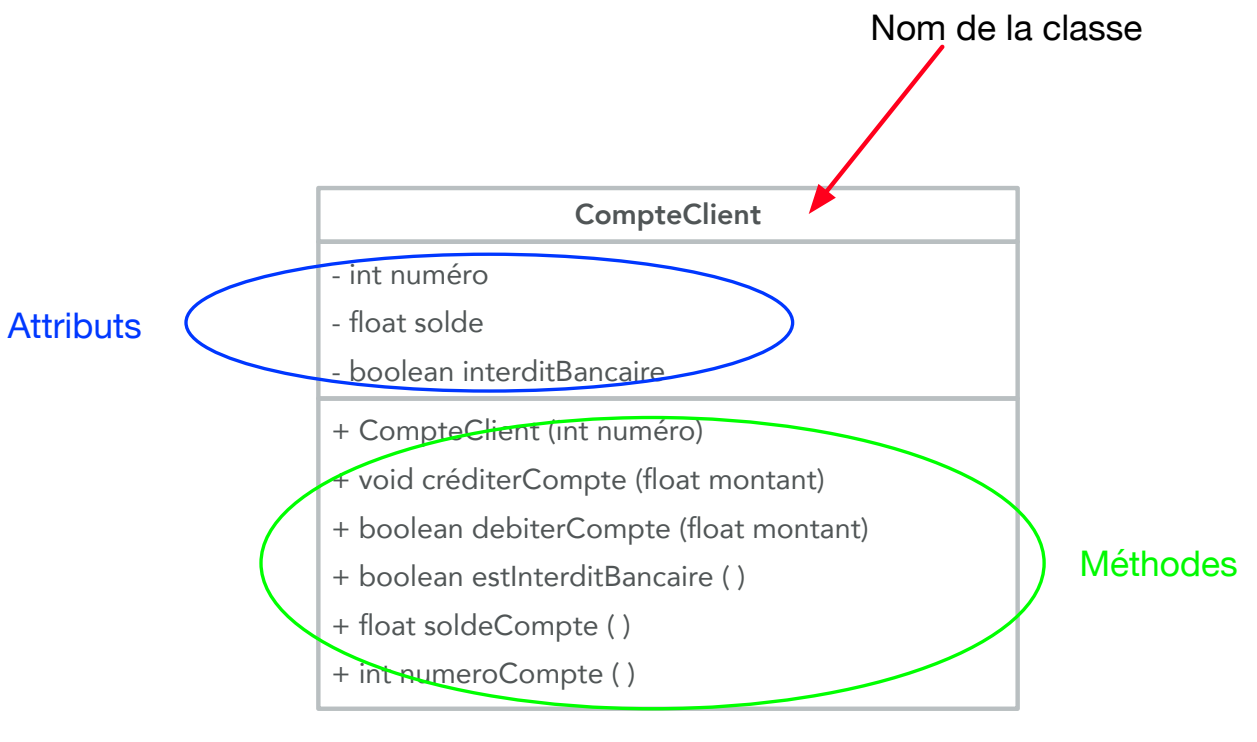
Exemple: La classe `CompteClient` dans une banque (1/2)



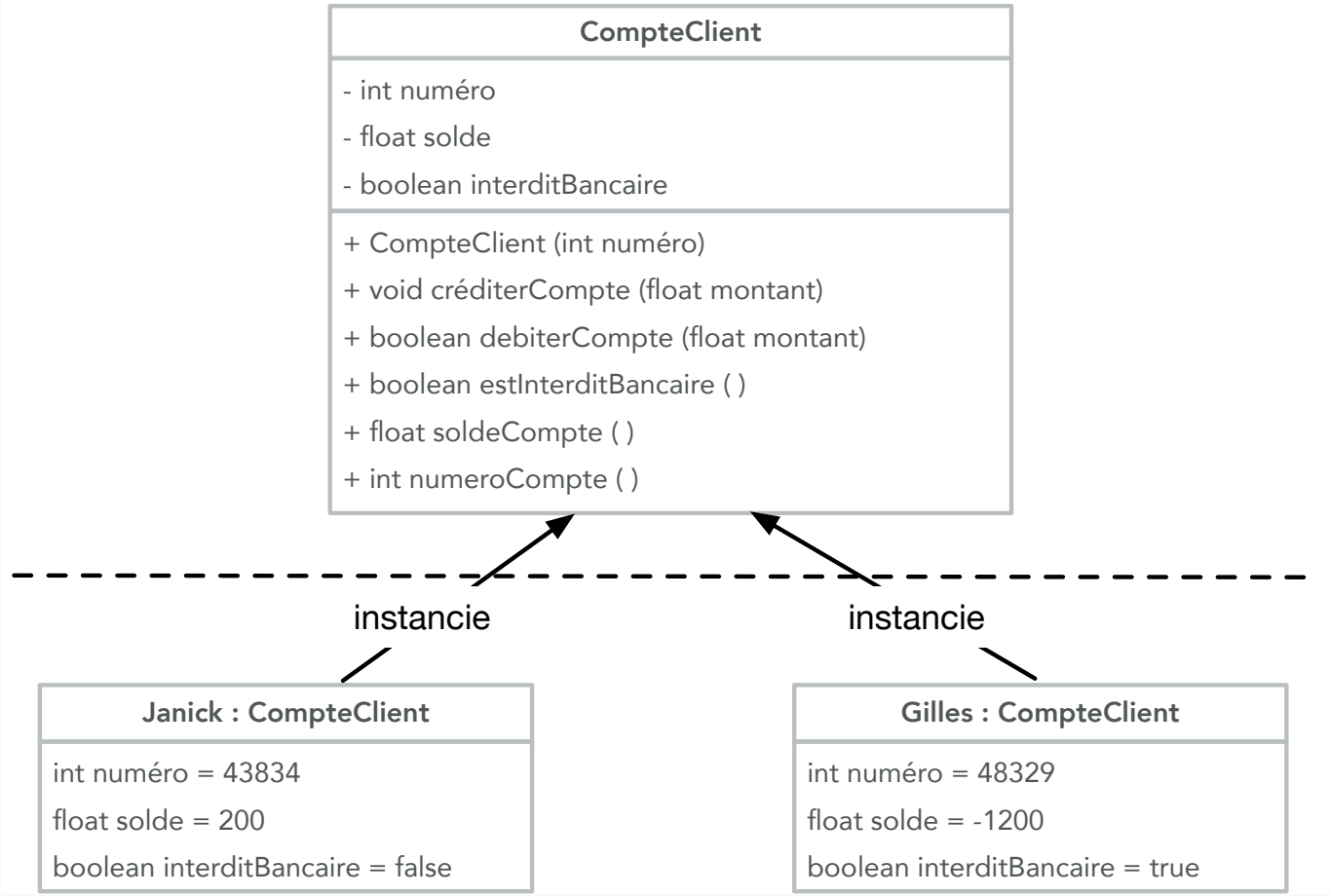
- *Encapsulation*: on a séparé les données des fonctions qui les manipulent, tout en conservant une seule entité, l'objet
- *Protection*: on a protégé le `solde`, dont la valeur ne peut être lue que par la méthode `soldeCompte( )`, et non modifiée

  sauf en ajoutant une méthode *void affecterCompte(float x)*.

Exemple: La classe `CompteClient` dans une banque (2/2)



Classes, instances...



Création d'un compte client

- Constat : on ne peut pas modifier le numéro d'un compte !
- La création d'un compte est une opération particulière qui doit :
  - définir de façon définitive le numéro du compte
  - initialiser le solde du compte (à 0 généralement)
  - mettre la variable *interditBancaire* à faux
- Les langages objet fournissent une méthode particulière : le *constructeur*
- Certains (dont C++) fournissent aussi un *destructeur*

Constructeurs et destructeurs

- Constructeur
  - Un constructeur est une fonction membre spéciale dont le nom est forcément celui de la classe
  - Un constructeur n'a pas de type
  - Une classe comporte généralement plusieurs constructeurs (en utilisant la surcharge, cf. 4)
  - En fonction du nombre et du type des arguments passés, le constructeur approprié sera choisi
  - Le constructeur est appelé **après** l'allocation de l'espace mémoire destiné à l'objet
  - Lors d'une construction par copie attention à la copie superficielle (simple affectation) ou profonde (re-création) des objets

Constructeurs et destructeurs

- Destructeur
  - Un destructeur est une fonction membre spéciale dont le nom est forcément un tilde (~) suivi du nom de la classe
  - Par définition, un destructeur n'a pas de type de retour ni d'argument
  - Le destructeur est appelé avant la libération de l'espace mémoire associé à l'objet

Constructeurs et destructeurs

- Si on ne définit pas de constructeur par copie, de destructeur et d'opérateur d'affection (voir 4), ces méthodes sont crées automatiquement
- Dans certains cas, un des champs de la classe est un pointeur vers une zone mémoire hors de la classe. Dans ce cas, les méthodes créées automatiquement ne fonctionnent pas correctement (fuite de mémoire, ou objets différents partageant la même zone mémoire).
- Il faut alors redéfinir à la main un constructeur par copie, un destructeur et un opérateur d'affectation.

compte.h

```
1   #ifndef compte_h
2   #define compte_h
3
4   class compte {
5   private:
6       bool ib;
7       int numero;
8
9   protected:
10      float solde;
11
12  public:
13      float getSolde();
14      int getNumero();
15
16      void crediter(float montant);
17      bool debiter(float montant);
18
19      // constructeurs
20      compte(int numero);
21
22      // destructeur
23      ~compte();
24  };
25
26  #endif /* compte_h */
```

compte.cpp

```cpp
#include "compte.h"

int compte::getNumero() {
    return numero;
}

float compte::getSolde() {
    return solde;
}

void compte::crediter(float montant) {
    solde += montant;
}

bool compte::debiter(float montant) {
    if (!ib && solde>=montant) {
        solde -= montant;
        return true;
    }
    return false;
}

compte::compte(int _numero) {
    numero = _numero;
    solde = 0.0;
    ib = false;
}

compte::~compte() {
    //nothing to do
}
```

Utilisation

```cpp
#include <iostream>
using namespace std;

#include "compte.h"

int main() {
    compte janick(65136);
    janick.crediter(2000.0);
    cout << "solde du compte janick : " << janick.getSolde() << endl;
    if (janick.debiter(1000.0)) {
        cout << "debit autorise" << endl;
    }
    else {
        cout << "operation interdite" << endl;
    }
    cout << "solde du compte janick : " << janick.getSolde() << endl;
    compte *thomas = new compte(45558);
    thomas->crediter(500);
    cout << "solde du compte thomas : " << thomas->getSolde() << endl;
    delete thomas;
}
```

Utilisation

```cpp
#include <iostream>
using namespace std;

#include "compte.h"

int main() {
    compte janick(65136);
    janick.crediter(2000.0);
    cout << "solde du compte janick : " << janick.getSolde() << endl;
    if (janick.debiter(1000.0)) {
        cout << "debit autorise" << endl;
    }
    else {
        cout << "operation interdite" << endl;
    }
    cout << "solde du compte janick : " << janick.getSolde() << endl;
    compte *thomas = new compte(45558);
    thomas->crediter(500);
    cout << "solde du compte thomas : " << thomas->getSolde() << endl;
    delete thomas;
}
```

Résultat

```
solde du compte janick : 2000
debit autorise
solde du compte janick : 1000
solde du compte thomas : 500
```

Constructeurs et destructeurs

- En C++, si une classe ne définit pas explicitement :
  - le constructeur de copie,
  - l'opérateur d'affectation par copie,
  - le destructeur,

  alors le compilateur génère automatiquement des versions par défaut.
- Ces trois méthodes sont fortement liées :
  - **Constructeur de copie**
  - **Opérateur d'affectation par copie**
  - **Destructeur**

  (on parle parfois de la *règle des trois*)
- Problème classique : si l'on définit explicitement l'un de ces éléments, le compilateur peut ne plus générer automatiquement les autres.

Constructeur de copie et opérateur d'affectation

- Exemple : une classe `Point2D` définit un constructeur de copie

```
class Point2D {
public:
    Point2D(Point2D const &p); // constructeur de copie
};
```

- Dans ce cas :
  - le compilateur ne génère plus automatiquement l'opérateur d'affectation par copie,
  - l'opérateur implicite devient déprécié,
  - ce qui peut produire un warning ou une erreur (avec `-Werror`).

Constructeur de copie et opérateur d'affectation

Solution explicite :

```
class Point2D {
public:
    Point2D() = default;

    // Constructeur de copie défini par l'utilisateur
    Point2D(Point2D const &p);

    // Opérateur d'affectation par copie explicitement demandé
    Point2D& operator=(Point2D const &other) = default;
};
```

- La syntaxe `= default` demande explicitement au compilateur :
  - de générer l'opérateur d'affectation standard,
  - avec le comportement par défaut (copie membre à membre).

Rappels : encapsulation

- Principe d'*encapsulation* : tous les attributs d'une classe doivent toujours être privés
- Cela signifie que que seules les méthodes internes y ont accès (et donc qu'elles font les modifications comme il faut)
- En clair : on a accès aux commandes de la voiture, pas directement à ce qu'il y a sous le capot
- Ce n'est pas qu'un concept, c'est mis en œuvre et forcé par le langage

Plan

Les méthodes constantes

- Une méthode constante est une méthode qui garantit ne pas altérer l'état d'un objet
- On parle également de méthode en lecture seule
- Déclaration :
  - Le prototype de la méthode est suivi du mot-clé `const`
  - La déclaration de la méthode (dans le .cpp) est suivi de `const`, juste avant l'accolade de début
- Utilité :
  - Savoir ce que fait réellement une méthode (utilisation sans risque)
  - Optimisations du compilateur

Les méthodes constantes

- Exemple : méthode `getSolde()` d'un compte bancaire

```
float getSolde() const;

float compte::getSolde() const {
    return solde;
}
```

Plan

Les fonctions amies

- En C++, l'unité de protection est la classe et non pas l'objet
- Donc une fonction membre d'une classe peut accéder à tous les membres privés de n'importe quel objet de sa classe
- En revanche, ces membres privés restent inaccessibles à n'importe quelle fonction membre d'une autre classe ou à n'importe quelle fonction indépendante
- La notion de "fonction amie", permet de déclarer dans une classe les fonctions que l'on autorise à accéder à ses membres privés avec le mot clé `friend`.

Les fonctions amies - exemple d'une fonction indépendante amie

```cpp
#include <iostream>
using namespace std;
class A {
    int i; // attribut de A, par défaut privé
    friend void exemple (A&);

    public:
     A(int e=100):i(e){}
};

void exemple (A& a) {
     // accès direct à i qui est membre données privées.
     cout << a.i << endl;
}

int main() {
    A y;
    exemple(y);
    return 0;
}
```

Les fonctions amies - exemple d'une fonction membre d'une autre classe amie

```
#include <iostream>
using namespace std;
// déclaration de la classe A, le
// compilateur sait que la classe A
// existe.
class A;
// définition de la classe B.
class B {
    public:
    /* La fonction exemple est uniquement
     déclarée. Comme argument, elle prend
     une référence à un objet de la classe
     A. On ne peut pas définir la fonction
     exemple à ce niveau, car si elle
     utilise un membre donné de la classe
     A, sachant qu'a n'a pas été encore
     définie, on aura des problèmes à la
     compilation.*/
     void exemple (A&);
};
// définition de la classe A
class A {
     int i;
     /* On connaît déjà le contenu de la classe
      B, exemple est une fonction membre de
      cette classe, elle est déclarée aussi, à
      ce niveau, comme amie de la classe A. Elle
      aura donc accès aux données privées de la
      classe A. */
     friend void B::exemple (A&);
     public:
      // constructeur qui initialise i à la
      // valeur de e, par défaut 100.
         A(int e=100):i(e) {}
};

// A & B étant définies, on peut définir
// la fonction exemple.
void B::exemple (A& a) {
     cout << a.i << endl;
}

int main() {
     A x;
     B y;
     y.exemple(x); // en sortie: 100
     return 0;
}
```

Classe amie d'une autre classe

```cpp
#include <iostream>
using namespace std;
// déclaration de la classe A.
class A;
// définition de la classe B.
class B {
public:
     void change (A&);
     void affiche(A&);
};
class A {
     int i;
/* La classe B est amie la classe A.
 De ce fait, les fonctions membres
 de la classe B c.-à-d. (change &
 affiche) sont amies de la classe A. */
friend class B;
public:
     A(int e=100):i(e) {} // constructeur.
};

/* définition de change, fonction membre
de la classe B.*/
void B::change(A& a) {
     a.i = 200;
}
/* définition de la fonction affiche,
 fonction membre de la classe B. */
void B::affiche(A& a) {
     cout << a.i << endl;
}
int main() {
     A x;
     B y;
     y.affiche(x); // en sortie: 100
     y.change(x);
     y.affiche(x); // en sortie: 200
     return 0;
}
```

Plan

Les membres données statiques

- Un membre donnée déclaré avec l'attribut `static` est partagé par tous les objets de la même classe
- Il existe même lorsque aucun objet de cette classe n'a été déclaré
- Un membre donnée statique doit être initialisé explicitement à l'extérieur de la classe (même si il est privé), en utilisant l'opérateur de résolution de portée (`::`) pour spécifier sa classe
- En général, son initialisation se fait dans la définition de la classe

Les membres données statiques - Exemple

```
class test{
    static int i; //Déclaration dans la classe
    ...
};
int test::i=3;
```

La variable `test::i` sera partagée par tous les objets instances de la classe `test`, et sa valeur initiale est 3.

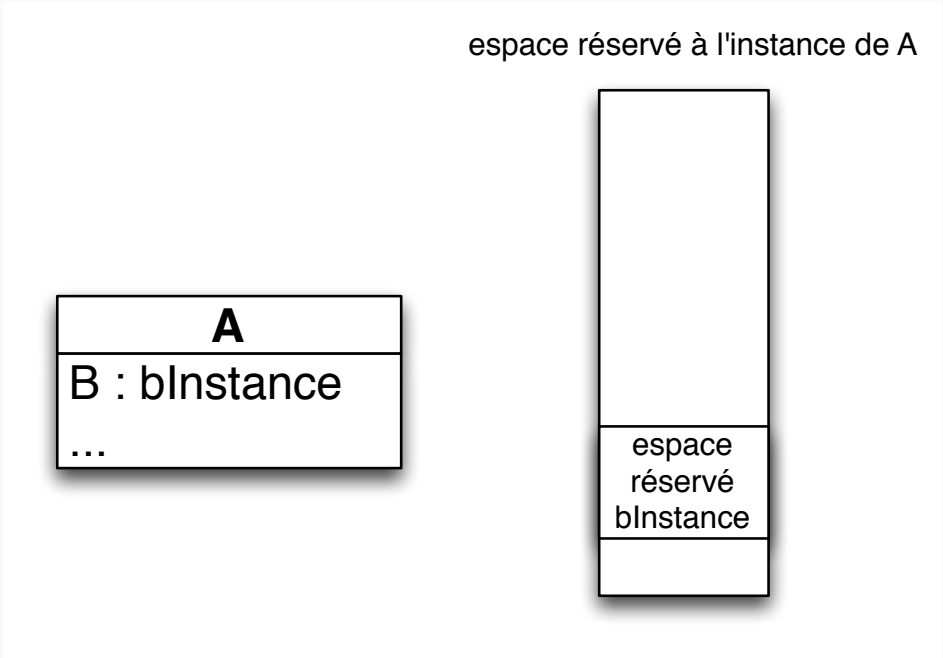
Plan

Gestion de la mémoire en C++ - les classes

- Que se passe-t-il lorsqu'une classe contient un élément d'un autre classe ?

```
class A {
        B instB;
};
```

- En pratique l'instanciation de A créée automatiquement une instance de B dans la zone mémoire réservée pour l'instance de A.



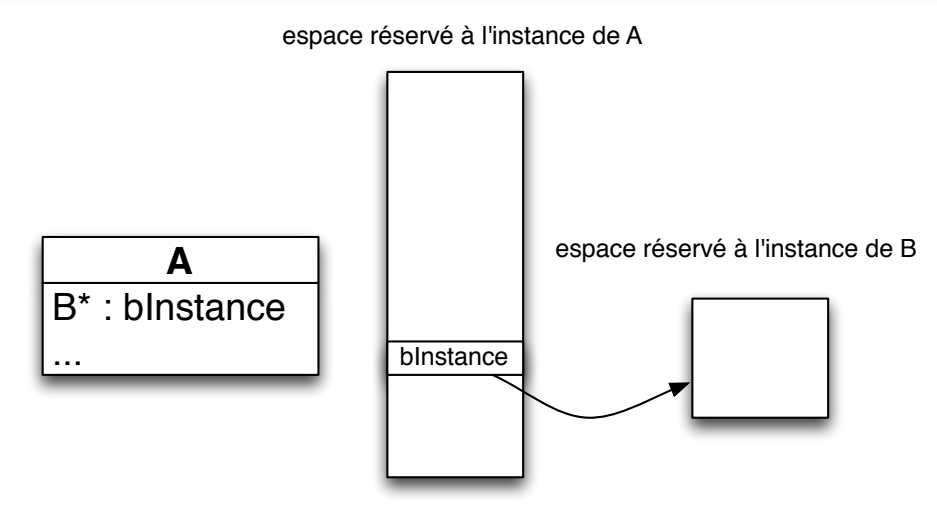
- la destruction de A entraine la destruction de B

Gestion de la mémoire en C++ - les classes

- Utilisation d'un pointeur

```
class A {
        B* instB;
};
```

- Les deux instances sont liées mais leur durée de vie est indépendante



Exemple de création/destruction de *instB*

```
A::A() {
        instB = new B();
}

A::~A() {
        delete instB;
}
```

Questions ?

- Pourquoi avoir écrit **exemple** au transparent précédent ?
- Cycle de vie d'un objet vs un autre
  - une question de génie de logiciel : B a-t-il une raison de survivre à A ?
    - non : bouton appartenant à une fenêtre
    - oui : un objet qui change de propriétaire (analogie jeu vidéo : un objet ramassé puis relâché)

Questions ?

- Ok, mais C++ ne sait pas se débarrasser d'un objet tout seul
  - Surtout si on utilise des pointeurs et des références
  - On **doit** avoir une stratégie de gestion de la mémoire
    - éviter l'utilisation de mémoire sans raison
    - éviter d'utiliser un objet préalablement libéré
  - Cas typique : dans un contexte de gestion dynamique de la mémoire, on a une fonction qui retourne un objet
    - qui se charge de l'allocation mémoire ? l'appelant, l'appelé ?
    - qui est responsable de la libération de la mémoire ?
    - attention aux créations d'objets temporaires

Exemple d'allocation dynamique d'un objet

```cpp
#include <iostream>
#include "point.h"
using namespace std;

int main() {
    Point *pointC; // je suis un pointeur sur un objet de type Point
    pointC = new Point(2,2); // j'alloue dynamiquement un objet de type Point
    pointC->afficher(); // Comme pointC est une adresse, je dois utiliser
    // l'opérateur -> pour accéder aux membres de cet objet
    pointC->setY(0); // je modifie la valeur de l'attribut _y de pointB
    (*pointC).afficher(); // cette écriture est possible :
    // je déréférence l'objet puis j'appelle sa méthode
    delete pointC; // ne pas oublier de libérérer la mémoire allouée pour cet objet
    return 0;

}
```

Les objets en mémoire

- Les objets **automatiques**
  - sont créés par une déclaration soit dans une fonction, soit au sein d'un bloc
  - sont créés au moment de la déclaration (peut apparaitre n'importe où dans un programme)
  - sont détruits lorsqu'on sort de la fonction ou du bloc
- Les objets **statiques**
  - sont créés par une déclaration située en dehors de toute fonction ou par une déclaration précédée de `static` (dans une fonction ou un bloc)
  - sont créés avant l'entrée dans la fonction `main` et détruits après la fin de son exécution

Les objets en mémoire

- Les objets **temporaires**
  - l'appel explicite, au sein d'un constructeur d'un objet provoque la création d'un objet temporaire (on n'a pas accès à son adresse)
  - pourra être automatiquement détruit dès qu'il ne sera plus utile
- Les objet **dynamiques**
  - sont créés par l'opérateur `new` avec le cas échéant les valeurs des arguments destinés à un constructeur
  - n'ont pas de durée de vie définie à priori, ils sont détruits par l'opérateur `delete`

Passage d'objets par valeur, par référence

Passage par valeur

```
1  void f(Complexe z){
2          z.set_real_imag(2, 3);
3          cout << "f : ";
4          z.affiche();
5  }
6  int main(){
7          Complexe p(1, 0.5);
8          f(p);
9          cout << "main : ";
10         p.affiche();
11         return 0;
12 }
```

Résultat

```
f : (2 + 3i)
main : (1 + 0.5i)
```

Passage par référence

```
1  void g(Complexe& z){
2          z.set_real_imag(2, 3);
3          cout << "g : ";
4          z.affiche();
5  }
6  int main(){
7          Complexe p(1, 0.5);
8          g(p);
9          cout << "main : ";
10         p.affiche();
11         return 0;
12 }
```

Résultat

```
g : (2 + 3i)
main : (2 + 3i)
```

Passage d'objets par valeur, par référence

Passage par valeur

```
1  void f(Complexe z){
2          cout << "Le complexe vaut ";
3          z.affiche();
4  }
5  int main(){
6          Complexe p(1, 0.5);
7          f(p);
8          return 0;
9  }
```

Résultat

L'objet est copié

Passage par référence constante

```
1  void g(const Complexe& z){
2          cout << "Le complexe vaut ";
3          z.affiche();
4  }
5  int main(){
6          Complexe p(1, 0.5);
7          g(p);
8          return 0;
9  }
```

Résultat

Une copie inutile est évitée

Copier un objet

La copie est la création d'un nouvel objet ayant la même valeur qu'un objet existant.

```
1   class Point
2   {
3           public:
4           ...
5           Point(const Point& other);
6           ...
7           private:
8           int m_x;
9           int m_y;
10  };
```

Copier un objet : le constructeur de copie

Exemple de définition d'un constructeur de copie :

```
1  #include "Point.h"
2  #include <iostream>
3  using namespace std;
4  Point::Point(const Point& other): m_x(other.m_x), m_y(other.m_y)
5  {
6      cout << "copie de Point" << endl;
7  }
```

Copier un objet : le constructeur de copie

Exemple de définition d'un constructeur de copie :

```
1  #include "Point.h"
2  #include <iostream>
3  using namespace std;
4  Point::Point(const Point& other): m_x(other.m_x), m_y(other.m_y)
5  {
6      cout << "copie de Point" << endl;
7  }
```

- En l'absence de constructeur de copie le compilateur en génère un automatiquement : il copie une à une les valeurs des membres.

Copier un objet : le constructeur de copie

Exemple de définition d'un constructeur de copie :

```
1  #include "Point.h"
2  #include <iostream>
3  using namespace std;
4  Point::Point(const Point& other): m_x(other.m_x), m_y(other.m_y)
5  {
6      cout << "copie de Point" << endl;
7  }
```

- En l'absence de constructeur de copie le compilateur en génère un automatiquement : il copie une à une les valeurs des membres.
- Lorsqu'une classe gère de la mémoire allouée dynamiquement, un constructeur de copie est donc indispensable.

Plan

Exercice - classe point

Réaliser un classe *point* permettant de manipuler les points d'un plan avec :

- un constructeur qui prend en argument les coordonnées réelles d'un point
- une fonction membre *deplace* qui effectue une translation définie par deux paramètres réels
- une fonction membre *affiche* qui affiche les coordonnées cartésiennes d'un point à l'écran
- deux accesseurs sur les attributs

La déclaration et les définition de la classe seront dans deux fichiers séparés et la classe sera testée à l'aide d'un programme principal.

Comment compter le nombre total d'instances de points créés ?

Plan

Plan

Surcharge et redéfinition en C++

- principe : des fonctions qui portent le même nom
- **surcharge** : plusieurs fonctions qui portent le même nom mais qui n'ont pas les mêmes paramètres (en nombre et/ou en type)
  - le compilateur décide de la fonction à appeler en fonction du type des paramètres
  - exemple : des fonctions *max()* à respectivement 2, 3 ou 4 paramètres, des fonctions *max()* s'appliquant à des types différents
  - Cas particulier : surcharge des opérateurs
- **redéfinition** : plusieurs fonctions qui portent le même nom et qui ont les mêmes paramètres
  - utilisé plus tard pour l'héritage

Plan

Surcharge de fonctions

- Exemple : des fonctions qui calculent le maximum de plusieurs entiers
- Ecriture “C” : une fonction *max2(a,b)* , une fonction *max3(a,b,c)*...
- Ecriture “C++” : toujours le même nombre de fonctions, mais elles portent toutes le même nom

```
1   #include<iostream>
2   using namespace std;
3
4   int max(int a,int b) {
5       if (a>b) {
6           return a;
7       }
8       else {
9           return b;
10      }
11  }
12
13  int max(int a,int b, int c) {
14      return max(a,max(b,c));
15  }
```

```
17  int max(int a,int b, int c,int d) {
18      return max(max(a,b),max(c,d));
19  }
20
21  int main() {
22      cout << "max(2,4) = " << max(2,4) << endl;
23      cout << "max(2,4,3) = " << max(2,4,3)
24              << endl;
25      cout << "max(6,4,5,7) = " << max(6,4,5,7)
26              << endl;
27  }
```

Surcharge de fonctions

Seule contrainte pour la surcharge : pas d'ambiguïté sur l'identité de la fonction à partir des arguments

Exemple non autorisé

```
double somme(int, int);
int somme(int, int);
somme(2,4); //Ambiguïté sur la fonction appelée
```

Surcharge de fonctions - arguments par défaut

Arguments par défaut ⇒ surcharge automatique de fonctions

Exemple :

```
double mafonction(int n, int m=2, double x=0.);
```

est équivalent aux **trois** déclarations :

```
double mafonction(int n);
double mafonction(int n, int m);
double mafonction(int n, int m, double x);
```

`mafonction(n)` correspond à `mafonction(n, 2, 0.)`

`mafonction(n,m)` correspond à `mafonction(n, m, 0.)`

Plan

Cas particulier : surcharge des opérateurs

- La plupart des opérateurs du C++ (par exemple, +, *, -, /, etc.) sont en fait des fonctions et peuvent être surchargées
  - Exemple : l'opérateur (binaire) d'addition est une fonction surchargée à deux paramètres de type `t` et qui retourne un type `t`
  - Elle pourrait s'écrire `int add(int a,int b)`
  - Inversement, une fonction de comparaison entre deux objets de type `C` s'écrit
    `bool egal(C const& a, C const&b)`
  - Autre possibilité, sous forme de méthode de la classe `C` :
    `bool egal(C const &a)`
  - On aimerait pouvoir l'écrire avec l'opérateur `==`

Exemple : comparaison de deux nombres complexes

- Déclaration des opérateurs (syntaxe forcée par la suite)

```
1   #ifndef complexe1_h
2   #define complexe1_h
3
4   class complexe1 {
5   private:
6       double re; //partie réelle
7       double im; //partie imaginaire
8
9   public:
10      complexe1();
11      complexe1(double _re, double _im);
12      // ... définition d'autres membres
13      bool egal(complexe1 const& c1) const;
14  };
15
16  // operateurs externes
17  bool egal(complexe1 const&a,complexe1 const&b);
18
19  #endif /* complexe1_h */
```

Exemple : comparaison de deux nombres complexes

- Définition

```
1   #include "complexe1.h"
2
3   complexe1::complexe1():re(0),im(0){}
4
5   complexe1::complexe1(double _re, double _im):re(_re),im(_im){}
6
7   bool complexe1::egal(complexe1 const& c1) const {
8       if (re == c1.re && im == c1.im) {
9           return true;
10      }else {
11          return false;
12      }
13  }
14
15  bool egal(complexe1 const&a,complexe1 const&b) {
16      return a.egal(b);
17  }
```

Utilisation des fonctions

- Code de test

```
int main() {
    complexe1 c1,c2(2,3),c3(c2);
    cout << "c1 == c2 (interne) " << c1.egal(c2) << endl;
    cout << "c1 == c2 (externe) " << egal(c1,c2) << endl;
    cout << "c3 == c2 (interne) " << c3.egal(c2) << endl;
    cout << "c3 == c2 (externe) " << egal(c3,c2) << endl;

}
```

Résultat

```
c1 == c2 (interne) 0
c1 == c2 (externe) 0
c3 == c2 (interne) 1
c3 == c2 (externe) 1
```

- Comment écrire `c1 == c2` ?
  - Utilisation d'un nouveau mot-clé : `operator`
  - La fonction associée à un opérateur `==` est `operator==()`

Surcharge de l'opérateur ==

- Déclaration

```
bool operator==(complexe1 const&a,complexe1 const&b);
```

- Définition

```
bool operator==(complexe1 const&a,complexe1 const&b) {
    return a.egal(b);
}
```

- Utilisation

```
    cout << "c1 == c2 (==) " << (c1 == c2) << endl;
    cout << "c3 == c2 (==) " << (c3 == c2) << endl;
```

Résultat

```
c1 == c2 (==) 0
c3 == c2 (==) 1
```

- A vous de faire de même avec l'opérateur `!=`

Surcharge de l'opérateur !=

- Déclaration

```
bool operator!=(complexe1 const&a,complexe1 const&b);
```

- Définition

```
bool operator!=(complexe1 const&a,complexe1 const&b) {
    return !a.egal(b);
}
```

- Utilisation

```
    cout << "c1 != c2 " << (c1 != c2) << endl;
    cout << "c3 != c2 " << (c3 != c2) << endl;
```

Résultat

```
c1 != c2 1
c3 != c2 0
```

- Remarques
  - Pourquoi avoir réutilisé le code existant ?
  - Fonction externe et encapsulation...

Le cas de l'addition

- Idée *naturelle*

```
complexe1 operator+(complexe1 const&c1,
                    complexe1 const&c2);
```

- Implémentation
  - encapsulation = pas d'accès direct aux variables membres, lourdeur syntaxique
  - passer par une fonction *amie* : rupture du principe d'encapsulation
  - passer par une méthode
    - Soit par délégation
    - Soit par analogie avec l'opérateur unaire : `a += 5`
    - Une approche véritablement objet

Surcharge de l'opérateur +=

- Déclaration

```
void operator+=(complexe1 const&c1);
```

- Définition

```
void complexe1::operator+=(complexe1 const&c1) {
    re += c1.re;
    im += c1.im;
}
```

- Utilisation

```
c2 += c3;
```

Résultat

```
2+3i + 2+3i = 4+6i
```

- Remarques
  - Pour bien faire, l'opérateur devrait retourner un `complexe1&`
  - Utilisation de `this`

Surcharge de l'opérateur +

- Déclaration

```
complexe1 operator+(complexe1 const& c1,
                    complexe1 const& c2);
```

- Définition

```
complexe1 operator+(complexe1 const& c1,
                    complexe1 const &c2) {
    complexe1 c3(c1);
    c3 += c2;
    return c3;
}
```

- Utilisation

```
    c = a+b;
    t = t1+t2+t3;
```

Résultat

```
3+4i + 3+2i = 6+6i
23+3i + 3+5i + 2+1i = 28+9i
```


- pour aller plus loin https://fr.wikibooks.org/

Autres opérateurs surchargeables

- opérateurs mathématiques
  - opérateurs binaires : +,-,*,/,%
  - opérateurs unaires : +=,-=,*=,/=,%=
- opérateurs logiques
  - bits à bits : ^, |, &, ~, «, » (et leur version unaire)
  - relationnels : ==, !=, <, >, <=, >=
  - logiques : !, &&, ||
- mais aussi
  - opérateur d'affectation =
  - opérateur d'accès tableau []
  - opérateur d'appel de fonction ()
  - opérateur de gestion des adresses : &, * et ->
- Certains sont plus dangereux que d'autres...

Autres opérateurs surchargeables (suite)

- Les opérateurs de gestion de mémoire (new, new[], delete, delete[])
- l'opérateur de séparation d'une liste , (utilité ??)
- les opérateurs de conversion
- Certains opérateurs ne **peuvent (quand même) pas** être surchargés
  - l'opérateur ternaire ? :
  - sélection d'un membre ., .*
  - opérateur de résolution de portée ::
  - `sizeof`
  - `typeid`
- De façon générale, la surcharge des opérateurs est à utiliser
  - avec parcimonie (on ne travaille pas tout seul)
  - pour l'extension des opérations communes à de nouveaux types
  - avec la même sémantique qu'avant la surcharge
    - typiquement ne pas transformer un + en - !

Cas particulier intéressant : surcharge de l'opérateur <<

- Pourquoi faire ?
  - afficher des infos relatives à un objet qu'on a défini
    - via un simple `cout` par exemple
  - sauver (et plus tard recharger) à partir d'un fichier
- Comment ?
  - surcharge simple et utilisation avec `cout`
  - `<<` est bien un opérateur :
    `ostream& operator<<(ostream&, type const&)`
- Redéfinition pour le type complexe

```
ostream& operator<<(ostream&s,complexe1 const&c) {
    s << c.getRe() << "+" << c.getIm() << "i";
    return s;
}
```

A retenir

Deux règles à garder en mémoire :

- On déclare au maximum les objets comme constants, pour éviter des erreurs dues à une modification d'un objet qui aurait dû rester inchangé
- On utilise au maximum des références, pour éviter des recopies d'objets qui pourraient être coûteuses (par exemple si les objets considérées manipulent des tableaux de grande taille).

Plan

Plan

Rappels : les classes

- Une classe représente une «famille» d'objets partageant les mêmes propriétés et méthodes
- Une classe sert à définir les propriétés des objets d'un type donné
    - décrit l'ensemble des données et des opérations sur ces données
    - elle sert de modèle pour la création d'objets (instances de la classe)

Réutilisation : introduction

- Comment utiliser une classe comme une brique de base pour concevoir d'autres objets ?
- En conception objet, on définit des associations (relations) entre objets pour définir la réutilisation entre classes
- UML (Unified Modeling Language) définit toute une terminologie des associations possibles entre classes
  - un objet fait appel à un autre $\longrightarrow$ délégation
  - un objet peut être créé à partir d'un autre objet $\longrightarrow$ héritage

Plan

Délégation

- Un objet **o1** membre de la classe *C1* utilise les services d'un objet **o2** instance de la classe *C2* (**o1** délègue une partie de son activité à **o2**)
- La classe *C1* utilise les services de la classe *C2*

- La classe cliente (*C1*) utilise les services de la classe serveuse

Schéma type

```
public class C1 {
private:
   C2 o2; // ou C2 *o2;
```

Délégation : exemple

- Exemple de la classe *Cercle*
  - rayon : un nombre réel
  - centre : deux réels ou bien un *Point*

```
public class Cercle {
private:
  Point centre; // ou Point *centre
  double rayon;

public:
  Cercle(Point _centre, double _rayon) {
    centre = _centre;
    rayon = _rayon;
  }
};
```

Délégation : exemple

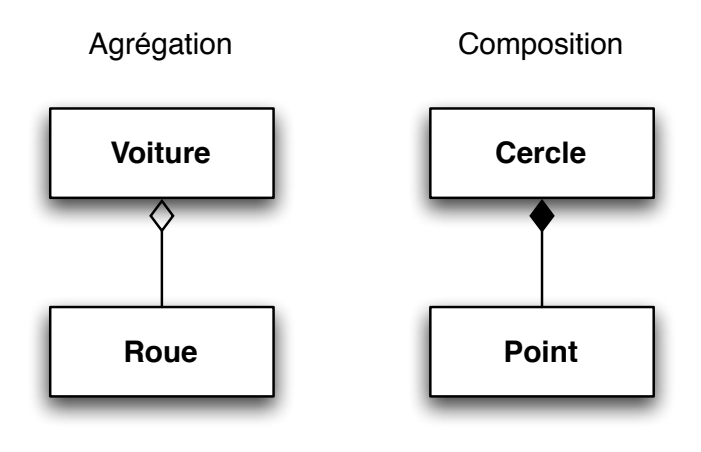
- l'association entre les classes *Cercle* et *Point* exprime le fait qu'un cercle **possède** (a un) centre
- Le point représentant le centre a une existence autonome (cycle de vie indépendant)
- Il peut être utilisé en dehors du cercle dont il est le centre !
  - si on translate le cercle, on translate le point et tous les objets qui utilisent ce point
  - Solution (en place) : effectuer une copie du point dans le constructeur

```
// cas Point centre
  centre = _centre; // dans les deux cas
// cas Point *centre
  centre = new Point(_centre);
```

Plan

Agrégation / Composition

- L'exemple précédent traduit deux nuances (sémantiques) de l'association **a un** entre la classe *Cercle* et la classe *Point*
- UML distingue deux types de sémantique en définissant deux types de relations



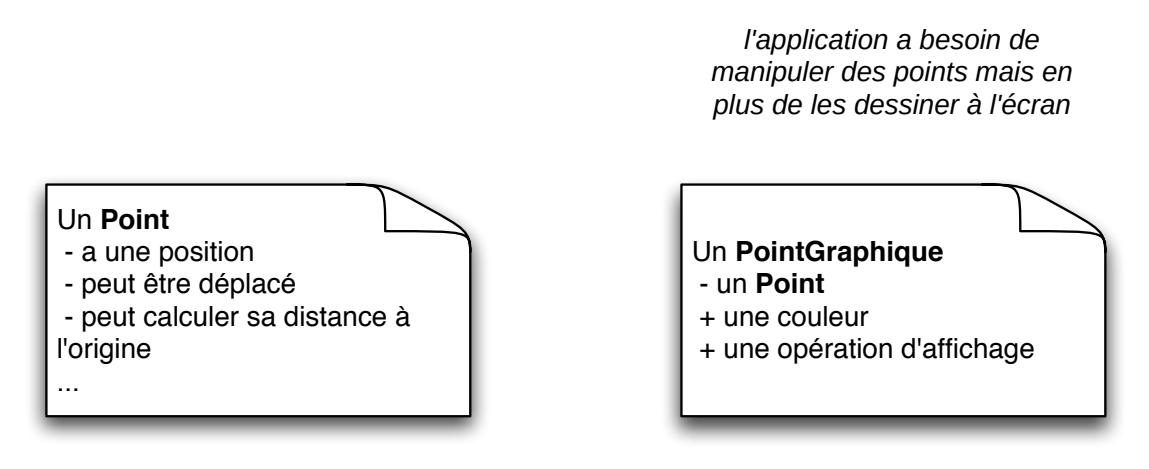
Agrégation / Composition

- **Agrégation** : l'élément agrégé *Roue* a une existence autonome en dehors de l'agrégat
- **Agrégation forte / composition** : à un même moment, une instance de composant *Point* ne peut être liée qu'à un seul agrégat *Cercle* et le composant a un cycle de vie dépendant de l'agrégat
  - Pas vrai dans tous les cas, mais voulu et forcé ici !

Plan

Héritage : exemple introductif

- Le problème :
  - une application a besoin de services dont une partie seulement est proposée par une classe déjà définie
  - ne pas réécrire le code



Pourquoi l'héritage plutôt que agrégation/composition ?

- par simplicité...
- ici, on réutilise les méthodes de Point dans PointGraphique
- éviter de rerouter tous les appels à *deplacer()* de PointGraphique vers Point
- exemple opposé : création d'une pile à partir d'une liste
  - on ne veut pas donner accès à toutes les méthodes de la liste

Héritage : syntaxe C++

- La classe *PointGraphique* hérite de (elle étend) la classe *Point*
  - elle possède les variables et méthodes définies dans la classe *Point* (*x* et *y*)
  - elle ajoute des attributs de couleur `r,g,b`
  - elle définit une nouvelle méthode `dessine()`

Déclaration

```
1  class PointGraphique : public Point {
2  private:
3      int r,g,b;
4
5  public:
6      PointGraphique();
7      PointGraphique(double _x,double _y,int _r,int _g, int _b);
8      PointGraphique(double _x,double _y);
9
10     void dessine();
11 };
```

Héritage : Ecriture des méthodes

- Seul le constructeur diffère !
- Il peut (doit) faire appel au constructeur de la classe *Point*
- Syntaxe `Pointgraphique::PointGraphique( ) : Point( )`

Définition des méthodes

```
1  #include "point.h"
2  #include "pointgraphique.h"
3
4  PointGraphique::PointGraphique() : Point() {
5      r = g = b = 255; // couleur par defaut = blanc
6  }
7
8  PointGraphique::PointGraphique(double _x,double _y,int _r,int _g, int _b)
9   : Point(_x,_y), r(_r), g(_g), b(_b) {
10     // plus rien a faire
11 }
12
13 PointGraphique::PointGraphique(double _x,double _y) : Point(_x,_y) {
14     r = g = b = 255; // couleur par defaut = blanc
15 }
```

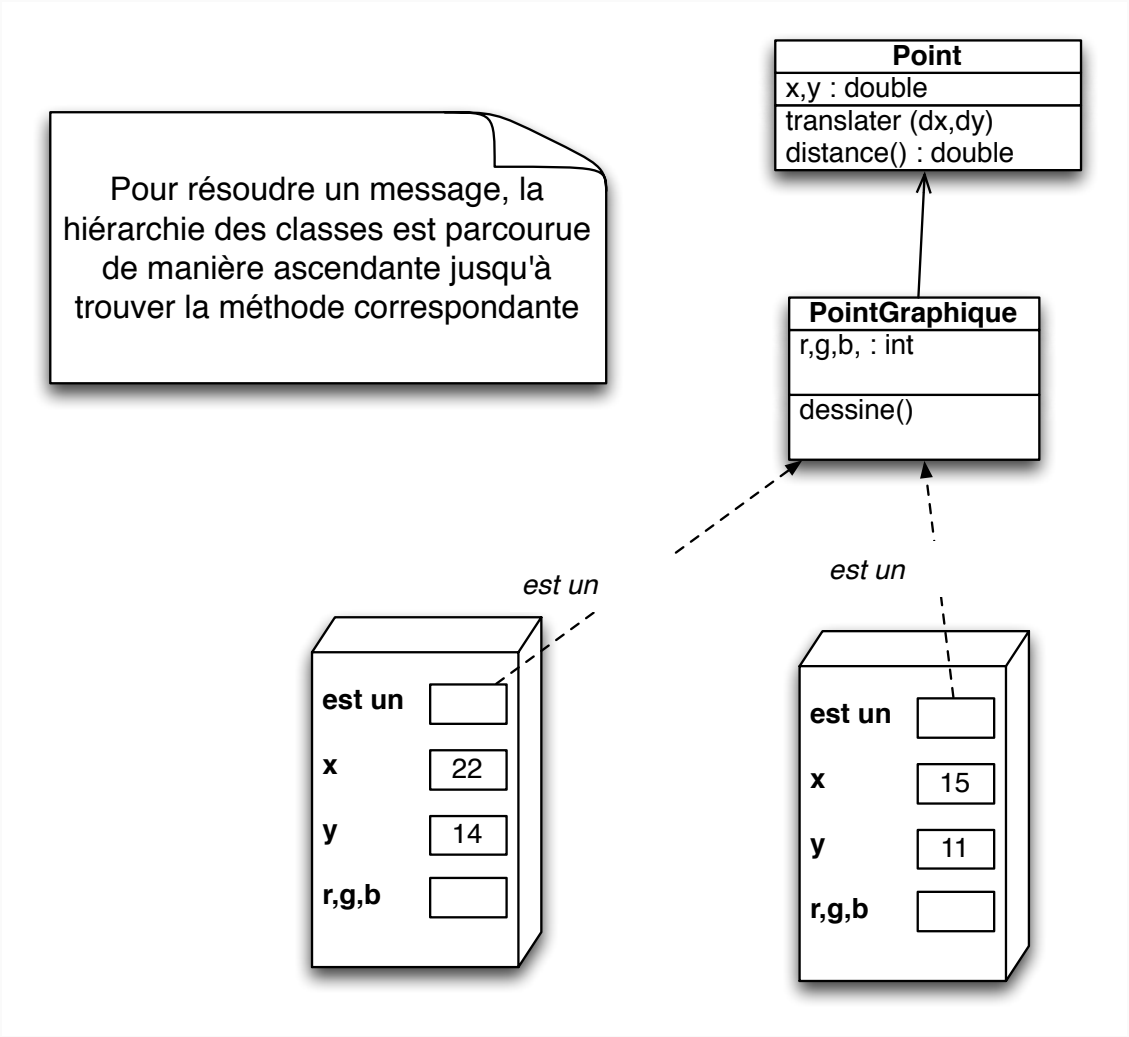
Utilisation des instances d'une classe héritée

- Un objet instance de *PointGraphique* possède les attributs définis dans *PointGraphique* ainsi que ceux définis dans *Point*
- Un objet instance de *PointGraphique* répond aux messages définis par les méthodes décrites dans *PointGraphique* et aussi à ceux définis dans les méthodes de *Point*

Utilisation

```
1  int main() {
2      PointGraphique pg;
3
4      pg.setX(5.5);
5      pg.setY(2.3);
6      pg.translater(2.0, 0.0);
7      pg.setCouleur(128, 128, 0);
8
9  }
```

Résolution statique des messages

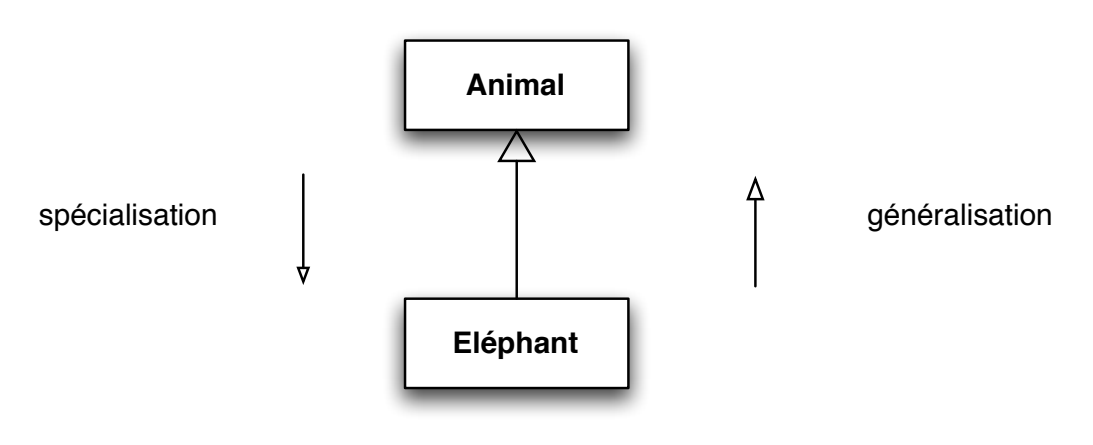


Plan

Terminologie de l'héritage

- L'**héritage** permet de reprendre les caractéristiques d'une classe *M* existante pour les étendre et définir ainsi une nouvelle classe *F* qui hérite de *M*
- les objets de *F* possèdent toutes les caractéristiques de *M* avec en plus celles définies dans *F*
  - *M* est la classe mère et *F* la classe fille
  - *F* hérite de *M*
  - *F* est une sous-classe de *M*
  - *M* est la super-classe de *F*

Généralisation / spécialisation



- la **généralisation** exprime la relation «est-un» entre une classe et sa super-classe
- la **spécialisation** exprime la relation de «particularisation» entre une classe et sa sous-classe

Généralisation / spécialisation

- Utilisation de la spécialisation : **réutilisation** par modification incrémentielle des descriptions existantes
- Utilisation de la généralisation : **abstraction** par factorisation des propriétés communes aux sous-classes
- il n'y a pas de limitation dans le nombre de niveaux de la hiérarchie d'héritage
- les méthodes et les attributs sont automatiquement héritées au travers de tous les niveaux

Plan

Redéfinition des méthodes

- une sous-classe peut **redéfinir** des méthodes dont elle hérite et fournir ainsi des implémentations spécialisées pour celles-ci
- lorsque la classe définit une méthode dont le nom, le type de retour et les arguments sont identiques à ceux d'une méthode dont on hérite
  - On dit aussi qu'une méthode **masque** celle de la classe mère
- Lorsqu'une méthode redéfinie par une classe est invoquée pour un objet de cette classe, c'est la nouvelle définition qui est invoquée

Redéfinition des méthodes : exemple (1/2)

- Déclaration

```
1   #ifndef exheritage_h
2   #define exheritage_h
3
4   class exheritageA {
5   public:
6       void hello();
7       void affiche();
8
9   };
10
11  class exheritageB : public exheritageA {
12  public:
13      void affiche();
14  };
15
16  #endif /* exheritage_h */
```

Redéfinition des méthodes : exemple (1/2)

- Définition

```
1  #include "exheritage.h"
2  #include <iostream>
3  using namespace std;
4
5  void exheritageA::hello() {
6      cout << "hello" << endl;
7  }
8
9  void exheritageA::affiche() {
10     cout << "je suis un objet de la classe exheritageA" << endl;
11 }
12
13 void exheritageB::affiche() {
14     cout << "je suis un objet de la classe exheritageB" << endl;
15 }
```

Redéfinition des méthodes : exemple (2/2)

```
1   #include "exheritage.h"
2   #include <iostream>
3   using namespace std;
4
5   int main() {
6       exheritageA a;
7       exheritageB b;
8
9       a.hello();
10      b.hello();
11
12      a.affiche();
13      b.affiche();
14  }
```

- produit l'affichage suivant

Redéfinition des méthodes : exemple (2/2)

```
1  #include "exheritage.h"
2  #include <iostream>
3  using namespace std;
4
5  int main() {
6      exheritageA a;
7      exheritageB b;
8
9      a.hello();
10     b.hello();
11
12     a.affiche();
13     b.affiche();
14 }
```

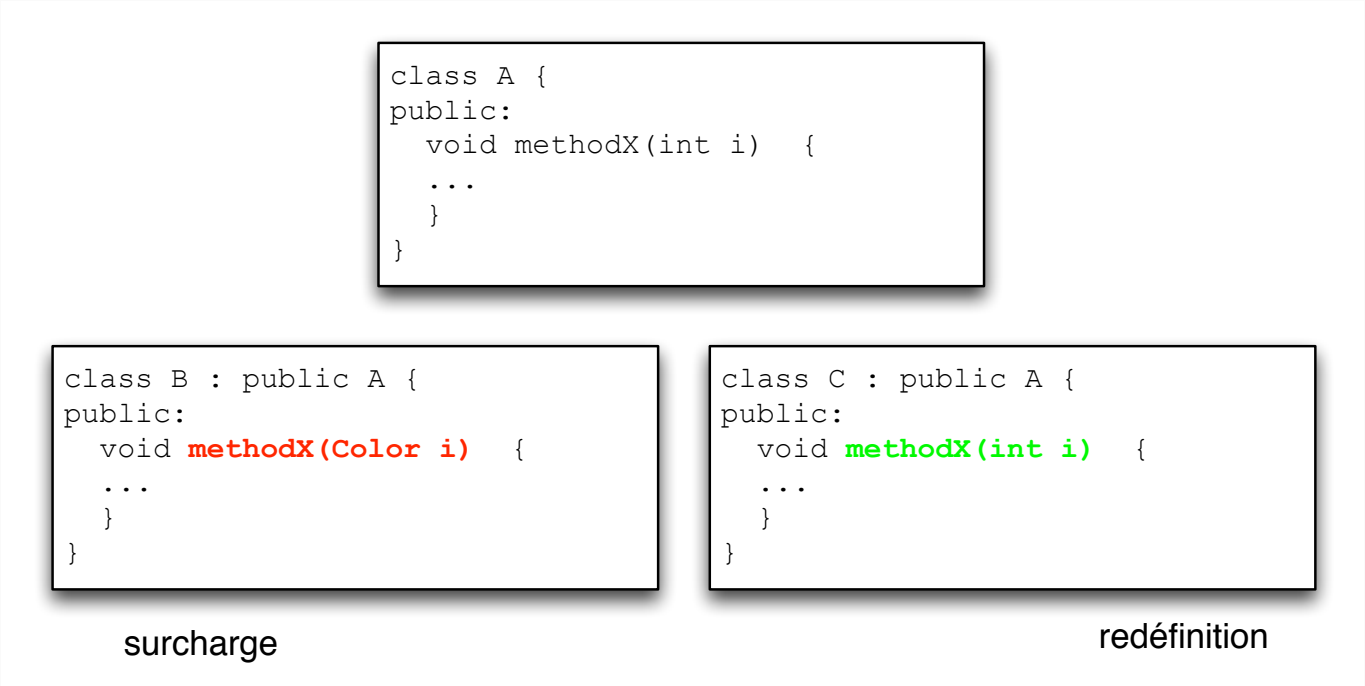
- produit l'affichage suivant

Résultat

```
hello
hello
je suis un objet de la classe exheritageA
je suis un objet de la classe exheritageB
```

Redéfinition des méthodes

- Ne pas confondre redéfinition et surcharge !



Limiter les risques (fonctions virtuelles et c+11) !

- Il arrive qu'on se trompe en redéfinissant une fonction … ce qui about à une surchage
- Exemple : fonction déclarée dans la classe mère

```
virtual void toto(int a, float b) const ;
```

- Erreur classique fonction *redéfinie* dans la classe fille

```
virtual void toto(int a, float b) ;
```

- Solution : le mot clé `override` dans la déclaration de la classe fille qui annonce qu'il s'agit d'une redéfinition
- Le compilateur va vérifier que c'est bien le cas

```
virtual void toto(int a, float b) const override;
```

Redéfinition avec réutilisation

- On peut réutiliser le code de la classe mère avec l'opérateur de résolution de portée `::`
- Exemple : Définition d'une classe *exheritageC* similaire à *exheritageB*

```
1  void exheritageC::affiche() {
2      exheritageA::affiche();
3      cout << "je suis un objet de la classe exheritageC" << endl;
4  }
5
6  // extrait du main()
7  int main() {
8  ...
9      exheritageC c;
10     c.affiche();
11 }
```

Redéfinition avec réutilisation

- On peut réutiliser le code de la classe mère avec l'opérateur de résolution de portée `::`
- Exemple : Définition d'une classe *exheritageC* similaire à *exheritageB*

```
1  void exheritageC::affiche() {
2      exheritageA::affiche();
3      cout << "je suis un objet de la classe exheritageC" << endl;
4  }
5
6  // extrait du main()
7  int main() {
8  ...
9      exheritageC c;
10     c.affiche();
11 }
```

Résultat

```
je suis un objet de la classe exheritageA
je suis un objet de la classe exheritageC
```

Plan

Protection

- Situation actuelle
  - **public** accessible depuis partout (sous réserve de disposer d'une instance de la classe)
  - **private** accessible uniquement depuis l'intérieur (i.e. les méthodes de la classe)
  - problème : une classe dérivée ne peut accéder aux membres privés de sa classe mère
- Solution
  - **protected** : accessible depuis l'intérieur de la classe et de toutes les classes dérivées
- Ce n'est pas tout
  - notion de fonctions **friend**
  - héritage via le mot-clé **public**... il existe d'autres formes d'héritage
  - classes internes

Plan

Exemple : compte bancaire avec autorisation de découvert

- Compte bancaire ”standard” : notion forcément incomplète
  - en pas uniquement en raison de nos simplifications
- On peut avoir différents types de comptes bancaires
- Exemple : certains comptes ont une autorisation de découvert, c’est-à-dire qu’on peut avoir un *solde légèrement négatif*, dans une limite fixée par la banque
- Nouvel attribut
  - le montant du découvert autorisé
- Méthodes
  - modification de la méthode *debiter()*
  - ajout d’un setter pour le montant du découvert autorisé
  - le reste est identique

Exemple : le code (1/2)

Déclaration de la classe

```
1   #ifndef comptead_h
2   #define comptead_h
3
4   class comptead : public compte {
5   protected:
6       float montant_ad;
7
8   public:
9       bool debiter(float montant);
10      void infos() const;
11      void setMontantAD(float m);
12      comptead(int numero);
13  };
14
15  #endif /* comptead_h */
```

Exemple : le code (2/2)

Définition des méthodes

```
1   #include "comptead.h"
2
3   void comptead::setMontantAD(float m) {
4       montant_ad = m;
5   }
6
7   bool comptead::debiter(float montant) {
8       if (!ib && solde+montant_ad>=montant) {
9           solde -= montant;
10          return true;
11      }
12      return false;
13  }
14
15  comptead::comptead(int n) : compte(n) {
16      montant_ad = 0.0;
17  }
18
19  void comptead::infos() const {
20      compte::infos();
21      cout << "montant de l'autorisation de decouvert : " << montant_ad << endl;
22  }
```

Plan

Plan

Rappels : les classes

- La réutilisation est un aspect important de l'héritage, mais pas forcément le plus important !
- Le deuxième point fondamental est la relation qui lie une classe à sa super-classe :
  - une classe *B* qui hérite de la classe *A* peut être vue comme un **sous-type** (sous-ensemble) du type défini par la classe *A*

Surclassement

- Tout objet instance de la classe *B* peut aussi être vu comme une instance de la classe *A*
- Cette relation est directement supportée par le langage C++
  - à une référence de type *A*, il est possible d'affecter une valeur qui est une référence vers un objet de type *B* (surclassement ou upcasting)
  - plus généralement, à une référence d'un type donné, il est possible d'affecter une valeur qui correspond à une référence dont le type effectif est n'importe quelle sous-classe du type de la référence
- Exemple : en supposant définie une classe *Etudiant* dérivant d'une classe *Personne*, on peut écrire

```
Personne *p = new Etudiant();
```

- Pourquoi utiliser des pointeurs ici ?

Le problème

- Déclaration d'une fonction d'affichage

```
void afficher(exheritageA a);
```

- Définition de la fonction

```
void afficher(exheritageA a) {
    a.affiche();
}
```

- Utilisation

```
    afficher(a);
    afficher(b);
    afficher(c);
```

Le problème

- Déclaration d'une fonction d'affichage
  ```
  void afficher(exheritageA a);
  ```
- Définition de la fonction
  ```
  void afficher(exheritageA a) {
      a.affiche();
  }
  ```
- Utilisation
  ```
      afficher(a);
      afficher(b);
      afficher(c);
  ```

Résultat

```
je suis un objet de la classe exheritageA
je suis un objet de la classe exheritageA
je suis un objet de la classe exheritageA
```

Le problème

- Déclaration d'une fonction d'affichage

```
void afficher(exheritageA a);
```

- Définition de la fonction

```
void afficher(exheritageA a) {
    a.affiche();
}
```

- Utilisation

```
    afficher(a);
    afficher(b);
    afficher(c);
```

Résultat

```
je suis un objet de la classe exheritageA
je suis un objet de la classe exheritageA
je suis un objet de la classe exheritageA
```

- Explication : passage par valeur $\Longrightarrow$ copie de l'objet

Le problème (suite)

- Passons par un pointeur !
- Déclaration de la nouvelle fonction

```
void afficherPtr(exheritageA *a);
```

- Définition de la fonction

```
void afficherPtr(exheritageA *a) {
    a->affiche();
}
```

- Utilisation

```
    afficherPtr(&a);
    afficherPtr(&b);
    afficherPtr(&c);
```

Le problème (suite)

- Passons par un pointeur !
- Déclaration de la nouvelle fonction

```
void afficherPtr(exheritageA *a);
```

- Définition de la fonction

```
void afficherPtr(exheritageA *a) {
    a->affiche();
}
```

- Utilisation

```
    afficherPtr(&a);
    afficherPtr(&b);
    afficherPtr(&c);
```

Résultat

```
je suis un objet de la classe exheritageA
je suis un objet de la classe exheritageA
je suis un objet de la classe exheritageA
```

Le problème (suite)

- la ligne `a->affiche();` appelle toujours affiche de la classe `exheritageA` (que `a` soit instance de `exheritageA`, `exheritageB` ou `exheritageC`)
- problème de typage statique
- nécessité d'un typage dynamique : *ie.* que les appels à une fonction ou à n'importe quelle de ses redéfinitions dans des classes dérivées soient "résolus" au moment de l'exécution, selon le type d'objet concerné
- notion de fonction *virtuelle* nécessaire pour avoir un typage dynamique en C++

Plan

Fonction virtuelle

- Syntaxe très simple
  - il suffit d'ajouter le préfixe *virtual* dans la **déclaration** de la classe mère
  - pas obligatoire dans les classes filles et les méthodes redéfinies (mais cela ne fait pas de mal à la compréhension du code)
  - NB : pour ceux qui connaissent java, toutes les fonctions sont virtuelles en java
- Signification
  - La résolution *statique* des liens est remplacée par une résolution *dynamique*
  - Si on utilise des fonctions virtuelles **et** des pointeurs/références vers un objet, alors la méthode appelée sera celle du type réel de l'objet
- Ca sera plus clair sur un exemple !

Exemple simpliste

- Reprise de l'exemple précédent en rendant *affiche()* virtuelle
- Seule modification : la déclaration de *affiche()* dans *exheritageA*

```
virtual void affiche();
```

- Utilisation inchangée

```
afficher(a);
afficher(b);
afficher(c);

afficherPtr(&a);
afficherPtr(&b);
afficherPtr(&c);
```

Exemple simpliste

- Reprise de l'exemple précédent en rendant *affiche()* virtuelle
- Seule modification : la déclaration de *affiche()* dans *exheritageA*

```
virtual void affiche();
```

- Utilisation inchangée

```
afficher(a);
afficher(b);
afficher(c);

afficherPtr(&a);
afficherPtr(&b);
afficherPtr(&c);
```

```
je suis un objet de la classe exheritageA
je suis un objet de la classe exheritageA
je suis un objet de la classe exheritageA
je suis un objet de la classe exheritageA
je suis un objet de la classe exheritageB
je suis un objet de la classe exheritageA
je suis un objet de la classe exheritageC
```

Surclassement bis

- Lorsqu'un objet est surclassé, il est vu comme un objet du type de la référence utilisée pour le désigner
- Ses fonctionnalités sont alors limitées à celle du type de la référence
- Résolution des messages :
  - Que se passe-t-il si on appelle une méthode (virtuelle) surchargée ?
  - c'est bien la méthode du type «réel» de la référence qui est appelée, telle qu'elle est définie au niveau de la classe

Exemple : limitation de fonctionnalités

- on ajoute à *exheritageC*

```
void methodeSupplementaire() {
  // ne fait rien
}
```

- dans le `main()` :

```
exheritageA *ptrC = &c;
ptrC->affiche();
ptrC->methodeSupplementaire();
```

Exécution

```
exheritage.cpp:24:11: error: no member named 'methodeSupplementaire' in 'exheritageA'
    ptrC->methodeSupplementaire();
    ~~~~  ^
1 error generated.
```

Liaison dynamique (uniquement méthodes *virtuelles*)

- Les messages sont résolus à l'exécution
  - La méthode exécutée est déterminée à l'exécution (run-time) et non pas à la compilation
  - à cet instant (et seulement à cet instant), le type exact de l'objet qui reçoit le message est connu
  - ce mécanisme est connu sous le nom de **liaison dynamique** ou dynamic binding ou encore late-binding voire run-time binding
- A la compilation, seules les vérifications statiques sont effectuées (signature)

Vérification statique insuffisante

- à la compilation il n'est pas possible de déterminer le type exact de l'objet qui reçoit le message
- la vérification statique garantit simplement dès la compilation que les messages pourront être résolus

La preuve !

```
exheritageA *ex[100];
for (int i=0 ; i<100 ; i++) {
    int z = rand() % 100;
    if (z > 50) {
        ex[i] = new exheritageB();

    }
    else {
        ex[i] = new exheritageC();
    }
    ex[i]->affiche();
}
```

Cas particulier : constructeurs et destructeurs

- Les constructeurs ne peuvent pas être virtuels :
  - à la construction, on sait quel objet on va construire !
  - il est également interdit de faire appel à une méthode virtuelle dans un constructeur
- Le destructeur peut être rendu virtuel
  - voire : le destructeur devrait être virtuel (lorsqu'on utilise le polymorphisme)
  - sinon appel à `delete` sur un pointeur surclassé génère un appel au mauvais destructeur !

Plan

Exemple : comptes rémunérés

- Il existe des comptes bancaires auxquels les banques apportent une rémunération
  - Par exemple, les livrets bancaires bénéficient d'une rémunération annuelle[1]
  - Mais il existe aussi des comptes courants rémunérés qui bénéficient d'intérêts pour lorsque le solde d'un compte dépasse un certain seuil
- Point commun : ils ont tous une fonction de calcul de rémunération du compte
- Souhait de l'agence : Pour chaque compte, appeler la fonction de rémunération du compte

[1] calculée tous les 15j, mais non considéré ici

Architecture de la solution

- Une classe mère commune à tous les comptes rémunérés (qui dérive elle-même de *compte*)
  - Elle comporte une méthode de calcul des intérêts qui ne fait rien
- Une classe *livret* qui hérite de la classe précédente et qui implémente sa propre version du calcul d'intérêts
- Une classe *compte courant rémunéré* qui fait de même
- Une agence bancaire contient une liste de comptes rémunérés (de tous types)
  - polymorphisme : pour chaque compte rémunéré, on peut appeler la méthode de calcul appropriée
  - C++ : nécessité d'utiliser des pointeurs dans la liste et des méthodes virtuelles

Exemple : déclaration des classes

- Classe *compteremun*

```
4   class compteremun : public compte{
5   public :
6       virtual void calc_interets();
7       compteremun(int n);
8   };
```

- Classes *livret* et *ccourantremun*

```
10  class livret : public compteremun{
11  protected:
12      float tauxannuel;
13  public:
14      virtual void calc_interets();
15      livret(int n, float t);
16  };
17
18  class ccourantremun : public compteremun{
19  protected:
20      float tauxquotidien;
21      float seuil;
22  public:
23      virtual void calc_interets();
24      ccourantremun(int n,float t,float s);
25  };
```

Exemple : définition des méthodes

```
4   void compteremun::calc_interets() {
5       // do noting
6   }
7
8   compteremun::compteremun(int n) : compte(n) {
9       // do nothing
10  }
11
12  void livret::calc_interets() {
13      solde += solde*(1.0+tauxannuel);
14  }
15
16  livret::livret(int n,float t) : compteremun(n) {
17      tauxannuel = t;
18  }
19
20  ccourantremun::ccourantremun(int n,float t,float s) : compteremun(n) {
21      tauxquotidien = t;
22      seuil = s;
23  }
24
25  void ccourantremun::calc_interets() {
26      if (solde > seuil) {
27          solde += (solde-seuil)*(1.0+tauxquotidien);
28      }
29  }
```

Exemple : calcul des intérêts

- Liste de comptes

```
8       list<compteremun*> agence;
9       agence.push_back(new livret(1,0.03));
10      agence.push_back(new ccourantremun(2,0.001,1500.0));
```

- Calcul des intérêts

```
17      for (compteremun* & c : agence) {
18              c->calc_interets();
19      }
```

- Problème : la méthode `compteremun::calc_interets()` qui ne fait rien, on aimerait
  - Forcer la redéfinition d'une méthode dans la classe fille et interdire l'instanciation de *compteremun*
- Solution : les classes et méthodes abstraites

Plan

Fonctions virtuelles pures

- Dans l'exemple précédent, on veut se dispenser de code pour la méthode `compteremun::calc_interets()`, forcer sa redéfinition dans les classes filles (sous-entendu : chaque classe fille doit **nécessairement** avoir une méthode `calc_interets()`)
  - Dans les langages à objet, on parle de méthode abstraite
- En C++, cela s'appelle une fonction virtuelle pure
- On ajoute juste `=0` derrière la déclaration de la fonction (et on supprime les définitions qui ne font rien)
- Exemple : `virtual void calc_interets()=0;`

Classes abstraites

- Une classe qui possède une méthode virtuelle pure ne peut pas avoir d'instance
  - Quel sens aurait un appel sur la méthode virtuelle pure ?
- On appelle une telle classe une classe abstraite
- En résumé
  1. Une méthode virtuelle *peut* être redéfinie dans une classe fille
  2. Une méthode virtuelle pure *doit* être définie dans une classe fille
  3. Une classe qui contient au moins une méthode virtuelle pure est une classe abstraite[2]

[2] il n'y a pas de mot clé associé en C++

Classe abstraite sans méthode en C++

- En C++, une classe est dite **abstraite** dès qu'elle contient au moins une **fonction virtuelle pure**.
- Cette fonction virtuelle pure peut être :
  - une méthode classique,
  - **ou un destructeur**.
- Il est donc possible de définir une classe abstraite **sans aucune méthode fonctionnelle**.

Exemple minimal :

```
class Plugin {
public:
    virtual ~Plugin() = 0; // destructeur virtuel pur
};

// le destructeur pur doit quand même être défini
Plugin::~Plugin() = default;
```

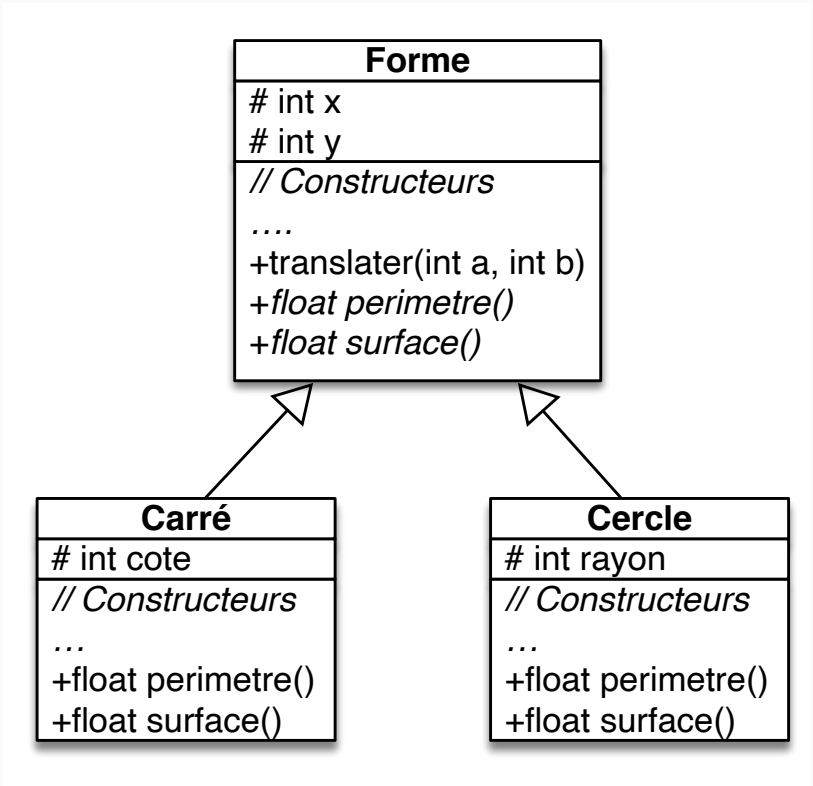
Utilisation des classes abstraites

- La classe ne peut pas être instanciée.
- Elle peut servir de type de base polymorphe.
- Les classes abstraites servent à définir des concepts incomplets qui devront être complétés dans les classes dérivées
- Elles permettent la factorisation du code, notamment pour la gestion des collections hétérogènes

Mini TP

1. On cherche à construire des classes correspondant à des formes géométriques définies a minima par un point d'ancrage qui doivent toutes êtres capables d'êtres translatées, de calculer leur aire et leur périmètre. Proposer une architecture à base de classe abstraite et au moins deux classes dérivées (cercle et carré par exemple) et les écrire en C++.
2. Écrire une classe *ListeDeForme* capable de prendre en compte une liste hétérogènes de formes géométriques.
3. Écrire une méthode ListeDeForme::AireTotale() qui calcule la somme des aires des formes contenues dans une liste

Architecture de la solution



Plan

Bibliographie

- B. Stroustrup (2013). The C++ programming Language, 4th edition. Addison Wesley.
- C. Delannoy (2000). Exercices en langage C++, 2nd édition. Eyrolles
- C. Hamadache, N. Kebaili (2020-2021). Initiation à la programmation orientée objet, Magistère 2ème année de Physique Fondamentale. Université Paris-Saclay - Faculté des Sciences d'Orsay.
- R. Roux (2020-2021). Cours de programmation en C++, master 2 ingénierie mathématique. Sorbonne Université.

Cours en ligne

- OpenClassrooms
  `http://fr.openclassrooms.com/informatique/cours/programmez-avec-le-langage-c`
- cplusplus.com
  `http://www.cplusplus.com/doc/tutorial/`