Chapitre 3
Programmation orientée objet et générique

Cours « *Programmation Orientée Objet* »
Semestre 7 (2018)

Julien Yves ROLLAND
julien.rolland@univ-fcomte.fr

Laboratoire de Mathématiques de Besançon
Université Bourgogne Franche-Comté

Buts et objectifs de ce chapitre.

1. Identifier les différentes relations possibles entre les classes
2. Formaliser le principe d'agrégation et de composition
3. Présenter les 3 bases de la programmation orientée objet :
   - Encapsulation
   - Héritage
   - Polymorphisme
4. Présenter les "template" et la programmation générique

Structure du chapitre.

1. Encapsulation
2. Agrégation, composition
3. Héritage
4. Polymorphisme

Définitions

L'encapsulation des données en POO permet de limiter l'accès des données grâce à des fonctions spécifiques. Ce principe permet d'imposer des invariants au sein d'une classe et de contrôler les arguments permettant la modifications des données.
Ex : taille d'une matrice, l'ouverture et communication avec une base de données, existence d'un fichier de sortie...

Encapsulation

Principe d'abstraction des données contraignant leur accès. Les données sont masquées dans l'implémentation de la classe et ne sont manipulées qu'au travers de fonctions spécifiquement définies dans l'interface, limitant les actions possibles.

Invariant de classe

Propriété existant au sein d'une classe et devant être contrainte lors de la manipulation des données de la classe.

Mots-clés C++ relatifs à l'encapsulation

public

Définit l'ensemble des données et des fonctions librement accessibles à l'extérieur d'une classe.

private

Définit l'ensemble des données et des fonctions réservées à l'usage interne d'une classe.

protected

Définit l'ensemble des données et des fonctions réservées à l'usage d'une classe et de ses classes dérivées (cf. héritage).

Fonction friend

Fonction non membre d'une classe mais ayant les droits équivalents à private.

Principes d'utilisation de l'encapsulation

Il faut s'assurer de la cohérence d'une classe lors de son utilisation :

- Les données définissant l'état d'une classe doivent faire partie de la partie private
- Les fonctions public doivent permettre la modification des données sous contrainte des invariants de la classe
- Les fonctions membres const sont employées quand l'état n'est pas modifié et que les invariants n'ont pas d'impact.

En pratique

- Les données sont presque toujours rangées en private.
- Des fonctions get_xxx() sont définies (assesseurs) pour lire les données.
- Des fonctions set_xxx() sont définies pour modifier une variable en imposant les invariants.

Exemple d'encapsulation

Fichier date.h

```
1   #include <ostream>
2   class Date{
3     public:
4     int get_year() const {return _y;}
5     int get_month() const {return _m;}
6     int get_day() const {return _d;}
7
8     void set_year(int);
9     void set_month(int);
10    void set_day(int);
11
12    friend std::ostream& operator<<(
13     std::ostream& stream, const Date& date);
14
15    private:
16    int _y{0}, _m{0}, _d{0};
17    };
```


Exemple d'encapsulation

Fichier date.cpp

```
1   #include "date.h"
2
3   void Date::set_year(int y){
4     if (y > 1900) _y = y;
5   }
6   void Date::set_month(int m){
7     if ((m > 0) && (m < 13)) _m = m;
8   }
9   void Date::set_day(int d){
10    if ((d > 0) && (d < 32)) _d = d;
11  }
12
13  std::ostream& operator<<(
14    std::ostream& stream, const Date& date)
15  {
16    stream << date._d << "/"
17           << date._m << "/"
18           << date._y;
19    return stream;
20  }
```

Classe dans une classe

Nous avons vu que les class sont des UDT et se comportent comme les type fondamentaux. Entre autre, il est possible de vouloir introduire un objet d'une classe définie comme donnée membre d'une autre classe.
Ex :

- Une bibliothèque et ses livres
- Une société et son siège social
- Une voiture et ses roues

Définitions

- L'objet contenant l'autre est nommé *agrégateur* ou *agrégat*
- L'objet contenu est nommé *agrégé*

La question est : Existe-t-il une relation d'appartenance entre les deux classes ? Quel est le cycle de vie de l'agrégé ?

Types de relation

Association

Les deux objets sont indépendants mais partage une relation, souvent par l'intermédiaire d'une troisième classe agrégat.

Ex : Des cours, des étudiants et une mention

Agrégation (ou agrégation faible)

Les deux objets ne sont pas liés : L'agrégé existe indépendamment de l'agrégateur mais en fait partie, ce dernier définit un concept pouvant exister seul et pouvant même faire partie de plusieurs agrégateurs.

Ex : Entreprise et employés.

Composition (ou agrégation forte)

Relation asymétrique entre les deux objets : L'un ne peut exister sans l'autre. On parle aussi de *composant* et *composite*

Ex : Un bâtiment et ses salles.

Composition et agrégation en C++

Agrégation

L'objet agrégé vient de l'extérieur, l'agrégateur n'est pas obligatoirement responsable de sa construction, ni sa destruction.

L'agrégé est défini en tant que pointeur.

Composition

L'objet agrégé est créé et détruit par l'agrégateur.

L'agrégé est une donnée membre.

Destruction de l'agrégé

La destruction de l'agrégé peut avoir une influence sur l'agrégateur (ou pas) : Si 0 est un nombre d'agrégé autorisé, il ne se passe rien, sinon l'agrégateur est peut-être à détruire.
Ex : Le dernier employé d'une entreprise, le dernier livre d'une bibliothèque.

Création et accès aux agrégés

Création

Le constructeur de l'agrégateur :

- copie la valeur du pointeur dans le cas d'une agrégation
- appelle le constructeur en cas de composition

Il peut être nécessaire d'effectuer des initialisations supplémentaires dans le corps du constructeur pour respecter les invariants.

Accès

L'encapsulation doit être préservée :

- Si modification, on préférera passer par une fonction qui effectuera la modification elle-même plutôt que retourner une référence à l'agrégé et perdre le contrôle.
- Si on veut directement manipuler l'agrégé, il ne doit pas être modifiable : on retourne un pointeur d'objet constant (voire un pointeur constant d'objet constant).

Un mot sur le langage UML

Unified Modeling language (UML)

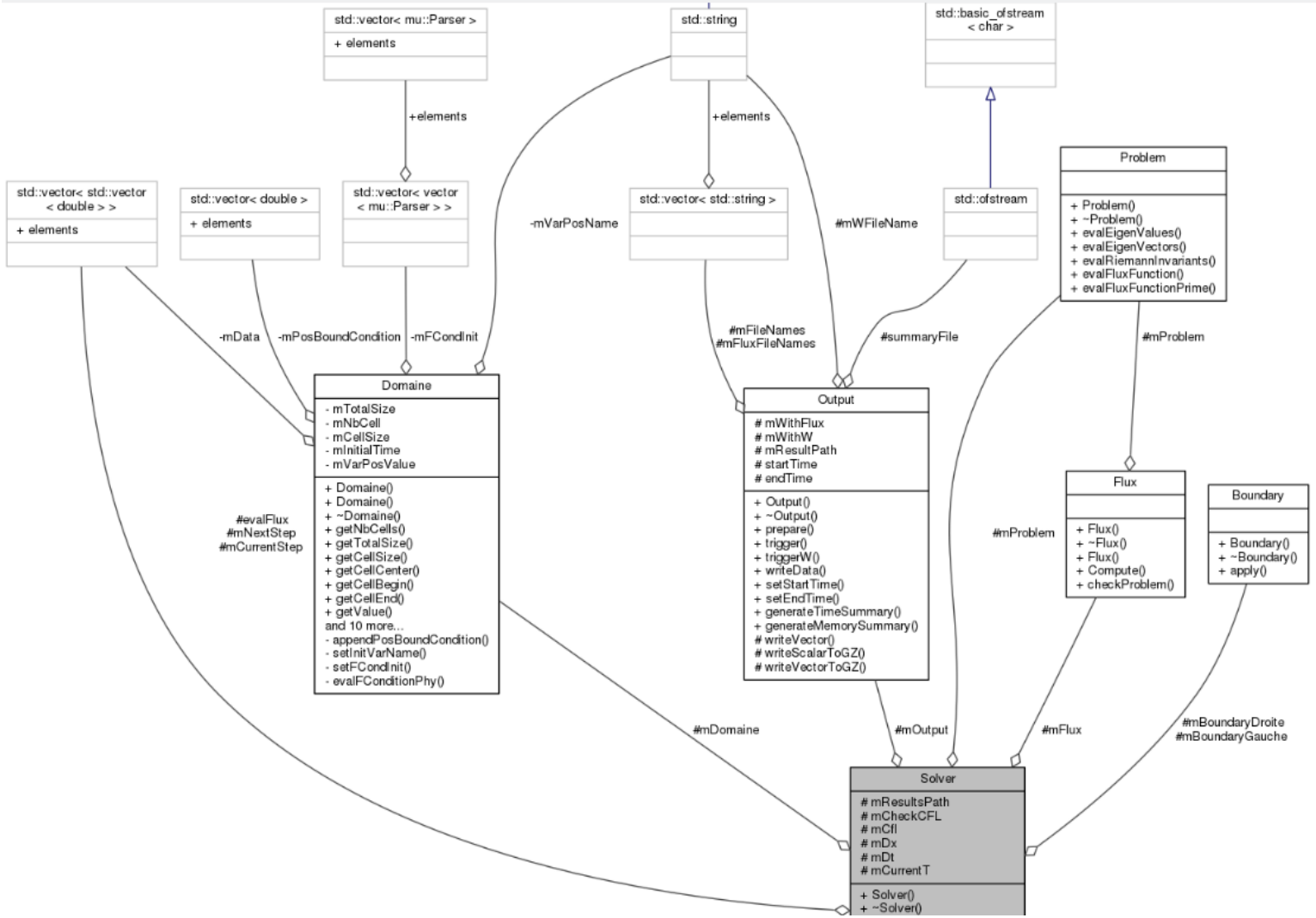
Langage du génie logiciel dédié à la description standardisée des logiciels et la visualisation des design logiciels choisis.

Conventions de notation

Dans ce standard, les classes sont des blocs. Concernant les relations :

- L'association est symbolisé par un ligne entre deux blocs
- L'agrégation est une ligne dotée d'un losange vide du côté de l'agrégat
- La composition est une ligne dotée d'un losange plein du côté de l'agrégat
- L'héritage est une flèche pointant vers la classe de base (cf. héritage)

Exemple UML



1 Encapsulation

2 Agrégation, composition

3 **Héritage**
- Classe dérivée
- Impact sur les fonctions

4 Polymorphisme

Contexte

L'agrégation répond à la relation « contient un », mais une classe peut être une variante d'une autre, une extension et répondre à la relation « est un ».

Ex : Les différents rôles des employés d'une entreprise, les différentes variantes d'un polyèdre.

Dans ce type de relation, on parle d'héritage : La classe va inclure les fonctionnalités d'une autre, elle va remplir un contrat "de base" et l'étendre, ou le remplir différemment.

Exemples :

- Une trottinette, un vélo, une voiture sont des véhicules mais roulent différemment.
- Un PDG, un manager et un manutentionnaire sont tous des employés d'une entreprise mais effectuent leur travail différemment.
- Un adulte et un enfant sont des personnes, mais uniquement l'adulte peut travailler.

Déclaration d'une classe dérivée

Classe de base, classe dérivée

La classe héritant d'une autre est appelée *classe dérivée* ou *classe fille*. La classe de départ est nommée *classe de base* ou *classe mère*.
On utilise aussi, mais plus rarement, les termes "subclass" et "superclass".

Déclaration d'une classe dérivée

La relation d'héritage est indiquée lors de la définition d'une classe :

```
1  class Fille: Mere{
2      ....
3  };
```


De quoi hérite-t-on et comment ?

Pour une classe de base B et une classe dérivée D.

Membres "private" de B

Ils restent privés à B et D n'y a pas accès (autrement que pas les accesseurs éventuels).

Membres "protected" et "public" de B

Tout dépend du niveau d'héritage de la B par D :

private	Les membres protected et public de B deviennent private dans D.
protected	Les membres protected et public de B deviennent protected dans D.
public	Les membres conservent leur niveau original : public de B devient public dans D, protected de B devient protected dans D.

En pratique

Le niveau d'héritage contrôle aussi la possibilité de convertir les pointeurs (cf. polymorphisme)

Un petit exemple complet

```
1  class B
2  {
3  public:
4      int x;
5  protected:
6      int y;
7  private:
8      int z;
9  };
10
11 class D1 : public B
12 {
13     // x est public
14     // y est protected
15     // z n'est pas accessible dans D1
16 };
17
18 class D2: protected B
19 {
20     // x est protected
21     // y est protected
22     // z n'est pas accessible dans D2
23 };
24
25 class D3: private B    // 'private' par défaut
26 {
27     // x est private
28     // y est private
29     // z n'est pas accessible dans D3
30 };
```


Données membres : "protected" ou "private" ?

Lors de la déclaration dans une classe B, soit :

- on décide que toutes les actions sur les données doivent se faire par des fonctions bien précises. Dans ce cas les données sont private et l'interface public contient ce qu'il faut.
- on sait que la classe sera dérivée et on ne souhaite pas imposer un contrôle des données, on les passe en protected

En pratique

Même si plus souple, les données en protected héritées ne sont plus soumises au contrôle de la classe de base :

- Le principe de l'encapsulation n'est plus respecté (données exposées en public) ;
- Les invariants éventuels doivent être de nouveau imposés par la classe dérivée.

Construction et destruction

Destruction

Le destructeur de la classe dérivée appelle automatiquement le destructeur de la classe de base après avoir fait son travail.

Constructeur

- Sans précision, les constructeurs de la classe dérivée appelle automatiquement le constructeur de base de la classe mère avant d'agir.
- La liste d'initialisation permet de préciser le constructeur.

Importer tous les constructeurs

Pour une classe de base B et une classe dérivée D :

```
using B::B;
```

surcharge les constructeurs (public) de D avec ceux de B. Leur corps est vide et la liste d'initialisation se limite au constructeur adéquat de la classe B.

Exemple

```
1  class Base{
2   public:
3    Base(int, int);
4
5   private:
6    int _x{0}, _y{0};
7  };
8
9  class Derive: public Base{
10  public:
11   using Base::Base;
12   Derive(int);
13
14  private:
15   int _x{0}; //in-class initialization
16 };
17
18 Derive::Derive(int i)
19   :Base{i, i}, _x{i}
20 {}
```


Redéfinition des fonctions héritées

Redéfinition ≠ Surcharge (override ≠ overload)

Une classe dérivée peut définir une fonction de même prototype qu'une fonction héritée. On parle de « redéfinition ».

```
1  class Forme{
2    public:
3    void afficher();
4  };
5
6  class Cercle: public Forme{
7    public:
8    void afficher();
9  };
10
11 class Rectangle: public Forme{
12   public:
13   void afficher();
14 };
```


Un mot sur l'héritage multiple

Héritage multiple

Rien n'interdit de faire hériter de plusieurs classes...

- Animal
- Mammifere et AnimalAquatique héritant d'Animal
- Dauphin héritant de Mammifere et AnimalAquatique

Dangers

- Héritage des grand-mères : Unicité ? ! ?
- Constructeur des grand-mères communes : Appel multiples ?
- Héritage de fonctions membres de noms identiques ?
- Redéfinition des fonctions de la grand-mère dans la mère puis héritées ?

En pratique

On attend d'être un vétéran du C++ pour l'héritage multiple !

Pointeurs et hiérarchie de classe

Type statique, type dynamique

- Le type statique est le type ayant servi à déclarer une variable. Il est connu à la compilation et invariant durant l'exécution.
- Le type dynamique est type courant de la variable, par définition il peut changer.

Les types statique et dynamique d'une variable non pointeur sont identiques et immuables.

Type dynamique d'un pointeur

Le type dynamique d'un pointeur est le type de l'objet pointé.

Les classes dérivées "étant aussi" de type de la classe de base, elles peuvent être manipulées par des pointeurs des deux types (sous conditions d'un héritage adéquat).

Exemples

```
1  class Forme;
2  class Rectangle;
3  class Cercle;
4
5  Forme* f;                  // statique : Forme*
6  f = new Rectangle();       // dynamique: Rectangle*
7  delete f;
8
9  f = new Cercle();          // dynamique: Cercle*
10
11 Rectangle* r;              // statique : Rectangle*
12 r = new Rectangle();       // dynamique: Rectangle*
```

En pratique

La substitution des pointeurs de Cercle et Rectangle par un pointeur de Forme n'est possible qu'en cas d'héritage public de Forme.

Appel statique, appel dynamique

Appel de fonction sur pointeur

Sauf mention contraire, un pointeur appelle les fonctions répondant à son type statique.

```
1  class Forme{
2    public:
3    void afficher();
4  };
5  class Retangle: public Forme{
6    public:
7    void afficher();
8  };
9
10 Forme* f;
11 f = new Rectangle();
12
13 f->afficher(); // Forme::afficher()
```


Appel statique, appel dynamique

Polymorphisme

Si on désire qu'une fonction s'adapte au type dynamique de son appelant, il est nécessaire dans la classe de base de lui associer le mot-clé virtual .
On parle alors de fonction virtuelle ou parfois de "méthode".

```
1  class Forme{
2   public:
3   virtual void afficher();
4  };
5  class Rectangle: public Forme{
6   public:
7   void afficher();
8  };
9
10 Forme* f;
11 f = new Rectangle();
12
13 f->afficher(); // Rectangle::afficher()
```


Quelques détails techniques

Prototype polymorphe

Le prototype de la fonction polymorphe doit respecter exactement celui de la fonction de base déclarée `virtual`.
Attention aux `const` !!

Résolution dynamique du type

À la compilation, le type statique est utilisé pour vérifier qu'à minima la fonction existe dans la classe de base.

Polymorphe mais pas automatiquement "morphée"

Rien n'oblige une classe dérivée à redéfinir une variante de la fonction polymorphe. Elle utilisera dans ce cas la version héritée de la classe de base.

En pratique

Le polymorphisme économise une multitude d'instruction `if {...} else {...}` sur les types.

Cas du destructeur

Destructeur virtuel

Si une classe indique qu'une méthode peut être polymorphe, alors le destructeur DOIT être déclaré explicitement virtual .

```
1  class Forme{
2    public:
3    ~Forme();
4  };
5
6  int main(){
7    Forme* f;
8    f = new Cercle();
9    delete f;              // ~Forme()
10   }
```

Destructeur "par défaut" polymorphe

```
virtual ~Forme()= default;
```


Héritages successifs : Propagation du virtual par héritage

```
1  class Base{
2    public:
3    void call();
4    virtual void vCall();
5  };
6  class Derived1: public Base{
7    public:
8    void call();
9    void vCall();
10 };
11 class Derived2: public Derived1{
12   public:
13   void call();
14   void vCall();
15 };
16
17 int main() {
18   Base*     base = new Derived2;
19   Derived1* der1 = new Derived2;
20   Derived2* der2 = new Derived2;
21
22   base->call();
23   der1->call();
24   der2->call();
25
26   base->vCall();
27   der1->vCall();
28   der2->vCall();
29 }
```


Classe abstraite, fonction virtuelle pure

Fonction virtuelle pure

Il est parfois très compliqué de trouver une définition pour une fonction polymorphe dans la classe de base : Il est possible de ne pas en donner !

```
virtual   afficher ()  = 0;
```

Classe abstraite

Est dite "abstraite" une classe contenant au moins une fonction virtuelle pure. Une des fonctions n'étant pas définie, on ne peut plus l'instancier (déclarer des objets de cette classe).

En pratique

Les classes abstraites servent de contrat : On impose ce que doit savoir faire une classe de ce type, libres aux dérivées de le faire comme elle le souhaitent, tant qu'elles le font.

Exemple

```
1  class Employe{
2    public:
3    virtual bool disponible()=0;
4
5    protected:
6    int matricule;
7  };
8
9  class Patron: public Employe{
10   public:
11   bool disponible(){return false;}
12 };
13
14 class Secretaire: public Employe{
15   public:
16   bool disponible(){return true;}
17 };
```


Type statique/dynamique pour le paramètre d'une fonction

Polymorphisme des paramètres

Le polymorphisme s'applique lors de l'appel d'une fonction membre, pas lors de la résolution du type d'un argument.
Le type statique est toujours utilisé pour résoudre une surcharge de fonction !

```
1  class Forme{
2    public:
3    virtual void inclure(Forme*);
4  };
5  class Cercle: public Forme{
6    public:
7    void inclure(Forme*);
8    void inclure(Cercle*);
9  };
10
11 int main(){
12   Forme* p1 = new Cercle();
13   Forme* p2 = new Cercle();
14   p1->inclure(p2);        // Cercle::inclure(Forme*)
15 }
```

Mais à quoi ça sert au final ?

Héritage de l'implémentation

La définition des fonctions membres dérivées profite de l'implémentation de la base.

Héritage de l'interface

Une fonction extérieure sachant travailler sur une classe de base, acceptera toutes les classes dérivées de la famille.

```
1   class Base;
2   class Derived1;
3   class Derived2;
4
5   void f1(Base& b){ b.vCall(); }
6   void f2(Base* b){ b->vCall();}
7
8   int main(){
9     Base*     base = new Base;
10    Derived*  der1 = new Derived1;
11    Derived2* der2 = new Derived2;
12
13    f1(*base); f1(*der1); f1(*der2)
14    f2(base);  f2(der1);  f2(der2)
15  }
```


1. Encapsulation
2. Agrégation, composition
3. Héritage
4. **Polymorphisme**
   - Polymorphisme "ad hoc"
   - Polymorphisme paramétrique

Retour sur la surcharge

Nous avons vu précédemment qu'il est possible de surcharger une fonction en conservant le nom mais en modifiant la liste des arguments :

```
1   int addition(int a, int b){
2     return a+b;
3   }
4   double addition(double a, double b){
5     return a+b;
6   }
7
8   int main(){
9     cout << addition(2, 5) << endl;
10    double a{2.5}, b{6.5};
11    cout << addition(a, b) << endl;
12  }
```

On note ici que le corps de addition() est strictement identique pour les types int et double ainsi que tous les autres types définissant operator+().
Il est possible d'indiquer en C++ un argument « générique » et de construire un « modèle » de fonction.

Fonction générique

Function template

```
template<typename monType>
monType addition(monType a, monType b)
```

Indique que la fonction est valable pour tout type "monType".
Le mot-clé typename peut être remplacé par le mot-clé class.

```
template <typename T>
T addition(T a, T b)
{
  return a+b;
}

int main(){
  cout << addition<int>(2, 5) << endl;
  double a{2.5}, b{6.5};
  cout << addition<double>(a, b) << endl;

  // Mais aussi directement
  cout << addition(2, 5) << "\t" << addition(a, b) << endl;

  return 0;
}
```


Classe générique : Déclaration

Class template [1] et "template instantiations"

On peut appliquer la paramétrisation à une classe, la contrainte est d'indiquer le template quand on qualifie l'espace de nom de la classe et de **provoquer** [2] la génération du code.

Fichier boite.h

```
1  #ifndef BOITE_H
2  #define BOITE_H
3
4  template<class T>
5  class Boite{
6  public:
7    Boite(int i=1);
8    ~Boite();
9    T& operator[](int n);
10
11 private:
12   T* list;
13   int taille;
14 };
15
16 #endif // BOITE_H
```

1. http://en.cppreference.com/w/cpp/language/class_template
2. https://isocpp.org/wiki/faq/templates#separate-template-class-defn-from-decl

Classe générique : Définition

Fichier boite.cpp

```
1  #include "boite.h"
2
3  template <class T>
4  Boite<T>::Boite(int i) : taille{i}
5  {
6    list = new T[i];
7  }
8
9  template <class T>
10 Boite<T>::~Boite()
11 {
12   delete[] list;
13 }
14
15 template <class T>
16 T& Boite<T>::operator[](int n)
17 {
18   return list[n];
19 }
```

Classe générique : Utilisation 1

Fichier main.cpp

```
1  #include "boite.h"
2
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8    Boite<double> bb{3};
9
10   for(int i=0; i<3; ++i)
11     bb[i]=i;
12
13   for(int i=0; i<3; ++i)
14     cout << bb[i] << endl;
15
16   return 0;
17 }
```

fin du fichier Boite.cpp

```
20
21 template class Boite<double>;
```

Classe générique : Utilisation 1bis

Fichier main.cpp

```
1  #include "boite.h"
2
3  #include <iostream>
4  using namespace std;
5
6  int main()
7  {
8    Boite<double> bb{3};
9
10   for(int i=0; i<3; ++i)
11     bb[i]=i;
12
13   for(int i=0; i<3; ++i)
14     cout << bb[i] << endl;
15
16   return 0;
17 }
```

Fichier generateBoite.cpp

```
1  #include "boite.cpp"
2
3  template class Boite<double>;
```


Classe générique : Utilisation 2

Fichier main.cpp

```
1  #include "boite.h"
2  #include "boite.cpp"
3
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9    Boite<double> bb{3};
10
11   for(int i=0; i<3; ++i)
12     bb[i]=i;
13
14   for(int i=0; i<3; ++i)
15     cout << bb[i] << endl;
16
17   return 0;
18 }
```


Classe générique : Utilisation 3 (Meilleur avec inline [3])

Fichier main.cpp

```
1  // Déclaration ET définition dans boite.h
2  #include "boite.h"
3
4  #include <iostream>
5  using namespace std;
6
7  int main()
8  {
9    Boite<double> bb{3};
10
11   for(int i=0; i<3; ++i)
12     bb[i]=i;
13
14   for(int i=0; i<3; ++i)
15     cout << bb[i] << endl;
16
17   return 0;
18 }
```

3. https://isocpp.org/wiki/faq/inline-functions#inline-member-fns

Programmation générique et POO

Programmation générique

Technique de programmation qui s'intéresse à l'écriture d'algorithmes se généralisant au plus grand nombre de types. Seules les conditions à respecter par les types limitent l'application de ces algorithmes.

Différence avec la POO

L'utilisation des template provoque la génération du code adéquate *à la compilation*. Tout le code nécessaire est généré en rencontrant "l'instantiation" des template.

La POO quant à elle, par le polymorphisme et le typage dynamique, est résolue *à l'exécution*.

Manipulation des types et metaprogrammation

La programmation générique – qui s'exprime principalement par l'usage des template en C++ – privilégie la conception de l'algorithme et son efficacité, mais masque les types.

Duck typing

> *If it walks like a duck and it quacks like a duck, it's a duck.*

Les algorithmes supposent des conditions à respecter par les types manipulés, si les conditions sont suffisamment respectées, alors l'algorithme fonctionne sur le type.

En pratique

Les dernières évolutions du langage C++ (norme C++14 et prochainement C++17) se concentrent principalement sur cet aspect du C++ et à la *metaprogrammation* : Exploiter la génération proposée par les template pour produire le code automatiquement et à la compilation.