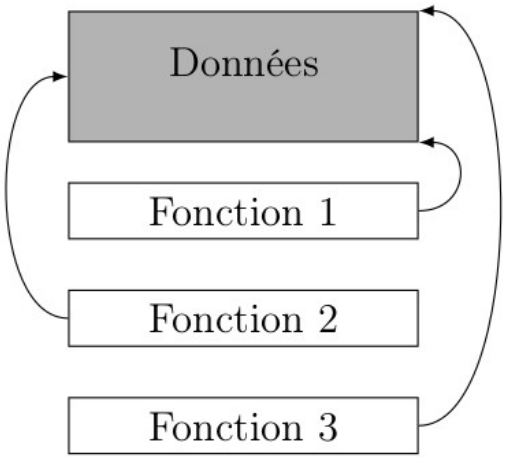
Bases de la programmation orientée objet (POO)

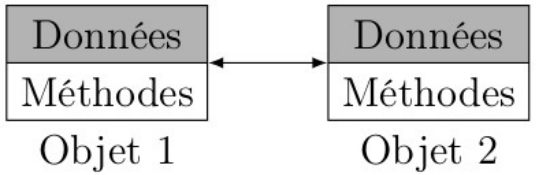
- Paradigme
- Implémentation en C++ : le type **`class`**

Paradigme

› Façon différente d'aborder la structure du code



Prog impérative



Prog objet

Paradigme

- Notions essentielles :
  - Encapsulation
  - Définition et instances de classes
  - Héritage
  - Polymorphisme

Paradigme

- Notions essentielles :

  - Encapsulation : toutes les parties d'un code n'ont pas besoin de connaître les détails d'implémentation d'un objet, ni d'avoir accès à ses données.
  - Certaines informations et interactions sont privées, d'autres publiques.

  - Ex machines à café : la machine a t-elle du café en grain, moulu, soluble ? Peu importe pour l'utilisateur qui veut juste appeler `makeCoffee()`.
  - La fonction `makeCoffee` modifie les données internes à la machine.
  - Interface avec l'utilisateur (autres parties du code) : indisponibilité, messages du monnayeur…

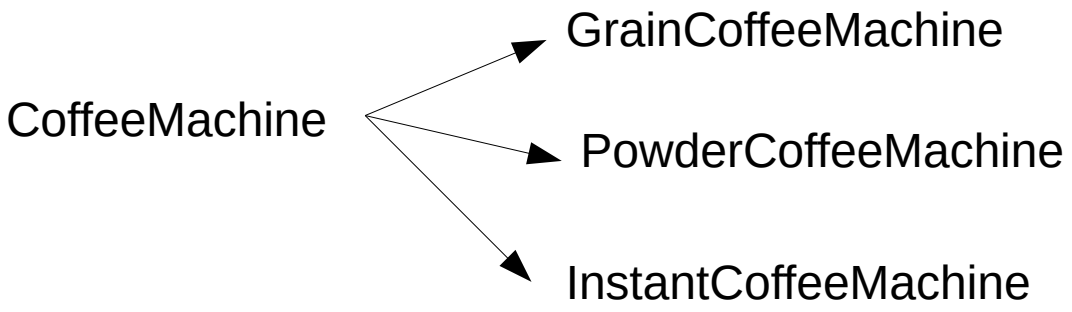
Paradigme

- Notions essentielles :

  - Définition et instances des objets
  - Un endroit où on définit un objet (une classe, comme au sens mathématique) : comment le construire : ce qu'il contient, quelle est l'interface. (machines à café : plan de montage)
  - Les objets eux-mêmes (les machines) : *instances de la classe*
  - Comme pour les types structurés (**`struct`**)

- Notions essentielles :

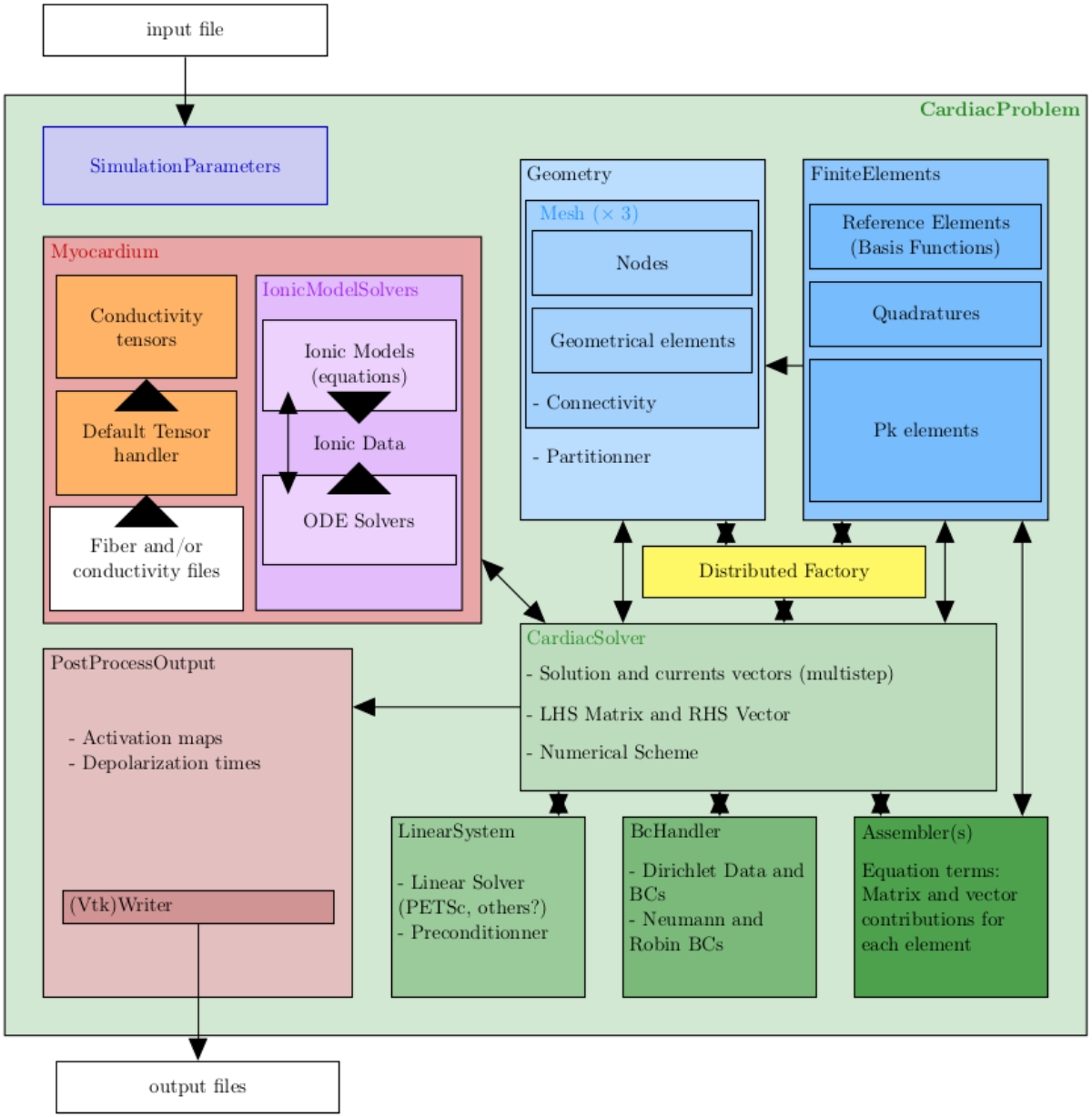
  - Héritage : les objets peuvent avoir des traits en commun et des éléments spécifiques.
  - Machines à café : toutes ont un monnayeur, une interface makeCoffee(), mais des façons différentes de faire le café.
  - Héritage : faire dériver une classe d'une autre classe



  - Polymorphisme : abilité d'une classe abstraite (coffeeMachine) à désigner plusieurs autres classes.

Paradigme

› Exemple de structure de code de calcul scientifique OO



Type class

- Extension des **`struct`**
- Un exemple minimal

```
struct Point {
  double _x,_y;
};

class Triangle {
  std::array<Point,3> _nodes;
  double _area;
}
```

- On peut créer des instances de la classe comme on déclarait de nouvelles variables de type structuré.

```
Point M;
M._x = 1.;
M._y = 2.;

Triangle T;
T._nodes[0] = M; // Won't work !
T._area = 0.; // Won't work !
```

- Vocabulaire : variables membres d'une classe : *attributs*

Type class

- Visibilité des membres d'une classe
  - Par défaut, tous les membres sont privés (seules les instances de cette classe peuvent y accéder.

```
class Triangle {
  public:
    // ...
  private:
    std::array<Point,3> _nodes;
    double _area;
}
```

  - Il existe aussi le niveau de permission **`protected`**, utile que lorsque l'on fait de l'héritage.
  - Passe-droit pour d'autres classes et fonctions avec le mot-clé **`friend`**

```
class Point {
  public:
    friend class Triangle;
    friend double dotProd(const Point&,const Point&);
  private:
    double _x,_y;
}
```

  - Attribut privé caché : **`this`** est un pointeur vers l'instance de la classe elle-même.

Type class

Méthodes membres (fonctions membres)

- Associées à chaque instance de la classe
- Appel avec la même syntaxe que pour les attributs

geom.hpp

```
1  #ifndef _GEOM_HPP_
2  #define _GEOM_HPP_
3
4  #include <array>
5
6  class Point {
7    private:
8      double _x,_y;
9  };
10
11 class Triangle {
12
13   public:
14
15     // Returns point of index i
16     Point getNode (int i);
17     // Returns all three points
18     std::array<Point,3> getNodes();
19     // Returns the area of the triangle
20     double getArea();
21
22   private:
23
24     // Internal method used to compute the area only once
25     void computeArea();
26
27     std::array<Point,3> _nodes;
28     double _area;
29     bool   _computedArea;
30
31 };
32
33 #endif // _GEOM_HPP_
```

Type class

Méthodes membres (fonctions membres)

- Le corps des fonctions peut-être placé dans le fichier hpp, mais il vaut mieux mettre les blocs longs dans un fichier source correspondant.

geom.hpp

```
1  #ifndef _GEOM_HPP_
2  #define _GEOM_HPP_
3
4  #include <array>
5
6  class Point {
7    private:
8      double _x,_y;
9  };
10
11 class Triangle {
12
13   public:
14
15     // Returns point of index i
16     Point getNode (int i);
17     // Returns all three points
18     std::array<Point,3> getNodes() {return _nodes;};
19     // Returns the area of the triangle
20     double getArea();
21
22   private:
23
24     // Internal method used to compute the area only once
25     void computeArea();
26
27     std::array<Point,3> _nodes;
28     double _area;
29     bool   _computedArea;
30 };
31
32 #endif // _GEOM_HPP_
```

Type class

› Méthodes membres (fonctions membres)

- Le corps des fonctions peut-être placé dans le fichier hpp, mais il vaut mieux mettre les blocs longs dans un fichier source correspondant.

geom.cpp

```cpp
#include <iostream>
#include "geom.hpp"

// ----------------
// Public methods

Point Triangle::getNode(int i) {
  if (i<0 or i>2) {
    std::cerr << "Cannot access point #" << i << " in a triangle !" << std::endl;
    abort();
  }
  return _nodes[i];
}

double Triangle::getArea() {
  if (not _computedArea)
    computeArea();
  return _area;
}

// ----------------
// Private methods

void Triangle::computeArea() {
  // do area computation (we'll complete later)
  // _area = ...
  _computedArea = true;
}
```

› Méthodes membres (fonctions membres)

› Utilisation de la classe

```
// ...
#include "geom.hpp"

Triangle t;
Triangle* t2;
// ... we'll complete later

double a  = t.getArea();
// 1. getArea() is public, we can call it from outside
// 2. we don't care how area is computed (which formula, whether the result is
     stored...)
double a2 = t2->getArea();
```

```
std::vector<double> v;

v.push_back(2.);
int s = v.size();
double a = v[0];

// push_back, size and operator[] are methods !
```

Type class

- Constructeurs et destructeur.
  - Le(s) constructeur(s) permet(tent) l'initialisation d'une instance de la classe.

```
// We will write constructors to be able to do things like that:
Point a(1.,2.) , b(0.5,-1.), c(2.,3.4);
a = Point(3.,-1.);
Triangle t(a,b,c);
```

  - Un constructeur bien implémenté initialise les attributs de la classe et si possible alloue l'espace mémoire nécessaire.

  - Chaque classe possède un constructeur par défaut, qui ne prend pas d'argument. S'il n'est pas précisé, le compilateur crée ce constructeur.

```
Point a(); // equivalent to Point a;
```

  - Il peut y avoir plusieurs constructeurs. Comme pour les surdéfinitions des fonctions, les méthodes (et donc les constructeurs) peuvent être surdéfinies.

Constructeurs et destructeur.

Implémentation

foo.hpp

```
1 class foo {
2
3   public:
4
5     // We can specify a default constructor
6     foo();
7     // Another constructor with arguments
8     foo(int, double, std::string, int, ...);
9 };
```

foo.cpp

```
1 foo::foo() {
2   // ...
3 }
4 foo::foo(int i, double a, std::string name, int b, ...) {
5   // ...
6 }
```

Type class

- Constructeurs et destructeur.
  - Constructeur par copie. Permet de faire **`Triangle t2(t1) ;`**

foo.hpp

```
class foo {
  public:
    foo(const foo& that) {
      // copy attribute
      this->attr1 = that.attr1;
      this->attr2 = that.attr2;
    }
};
```

Type class

- Constructeurs et destructeur.
  - Le destructeur : il ne peut y en avoir qu'un par classe. S'il n'est pas écrit, le compilateur en crée un par défaut.
  - Appelé automatiquement dès que l'instance de la classe est détruite (sortie de bloc, ou opérateur **`delete`**)
  - Ne prend aucun argument.
  - Appelle automatiquement les destructeurs des attributs de la classe, sauf ceux alloués dynamiquement.
  - Implémentation

foo.hpp

```
class foo {
  public:
    foo(); // constructor
   ~foo(); // destructor
};
```

foo.cpp

```
foo::~foo() {
  // do something
}
```

  - Le destructeur sert notamment à libérer l'espace mémoire qui a été alloué dynamiquement (**`delete`**)

Type class

› Exemple.

mesh.hpp

```
#ifndef _MESH_HPP_
#define _MESH_HPP_
#include "geom.hpp"

// A basic mesh made of triangles
class Mesh {
  public:

    // Default constructor, does almost nothing
    Mesh() {
      _Nt = 0;
    }

    // Constructor with number of points (all triangles are initd with default
     constructor of Triangle)
    Mesh(int N) {
      _Nt   = N;
      _tris = new Triangle[N];
    }

   // Destructor: the delete operation calls automatically the destructor of each
      Triangle as well
   ~Mesh() {
     if (_tris != nullptr) delete[] _tris;
   }

   // ... add here addTriangle(...), getTriangle(...), getNumberOfTriangles(...),
      etc

  private:

    int       _Nt;   // Number of triangles
    Triangle* _tris; // The triangles (allocated without vectors)
};

};

#endif // _MESH_HPP_
```

Type class

› Exemple (utilisation)

```
{ // block begin

  int N = 100;
  Mesh m(N); // N triangles will be created

  for(int i=0; i<m.getNumberOfTriangles(); i++) {
    Triangle *t = m.getTriangle(i);
    // Do something with t
    // ...
  }

} // end of block, ~Mesh() is called here, and the triangles are destroyed as
     well
```

Type class

- Surcharge d'opérateur.
  - Comme pour les autres méthodes membres, on peut surdéfinir les opérateurs agissant sur des classes.
  - Par exemple, si on a une classe qui décrit les nombres complexes :

```
complex c1(0.,1.),c2(-2.,1.),c3;
c3 = c1 + c2;
std::cout << c3 << std::endl;
```

Type class

- Surcharge d'opérateur.
  - Deux façons de faire. Soit en créant une fonction membre de la classe

```
class complex {
  pulic :
    // Constructors, destructors, getters, setters, module, conjugate, etc.

    complex operator+(const complexe& c2);
    complex operator=(const complexe& c2);  // Assign by copy

  private :
    double _re,_im;

}

complex complex::operator+(const complexe& c2) {
  return complex(_re + c2._re, _im + c2._im);
}

complex& complex::operator=(const complex& that) {
  if (*this != that) { // Do something only if we don't have c = c;
    _re = that._re;
    _im = that._im;
  }
  return *this; // This is MANDATORY to allow c3 = c2 = c1;
}
```

  - Un seul argument. L'autre est implicite, puisqu'il s'agit de **`this`**

Type class

- Surcharge d'opérateur.
  - Soit en utilisant une fonction externe, qui doit être déclarée amie.

```
class complex {
  pulic :
    complex operator=(const complexe& c2);  // Assign by copy
    // Constructors, destructors, getters, setters, module, conjugate, etc.
    friend complex operator+(const complex&, const complex&);
    friend ostream& operator<<(ostream& os, const complex&);
  private :
    double _re,_im;
}

complex operator+(const complex& c1, const complex c2) {
  return complex(c1._re + c2._re, c1._im + c2._im);
}

ostream& operator<<(ostream& os, const complex& c) {
  os << c._re << "+ " << c._im << "i";
  return os;
}
```

- Note : différence entre opérateur d'assignation et constructeur par copie. Dans le cas de l'assignation, l'objet à gauche du signe = est déjà créé ! Attention aux potentielles fuites mémoire.