De manière superficielle, le terme « ***orienté objet*** » signifie que l'on organise le logiciel comme une collection d'objets dissociés comprenant à la fois une structure de données - ***attributs*** - et un comportement – ***méthodes*** - dans une même entité.

Exemple : une voiture peut avoir une certaine couleur et en même temps possède un comportement qui sera le même pour toutes les autres voitures, comme accélérer. Ce concept est différent de la programmation conventionnelle dans laquelle les structures de données et le comportement ne sont que faiblement associés.

Les objets :

Chaque objet possède une identité et peut être distingué des autres. Deux pommes ayant les mêmes couleur, forme et texture sont des pommes individuelles ; une personne peut manger l'une, puis l'autre. De la même façon, des jumeaux sont deux personnes distinctes, même si elles se ressemblent. Le terme identité signifie que les objets peuvent être distingués grâce à leurs existences inhérentes et non grâce à la description des propriétés qu'ils peuvent avoir. Nous utiliserons l'expression « ***instance*** » d'objet pour faire référence à une chose précise, et l'expression « ***classe*** » d'objets pour désigner un groupe de choses similaires. En d'autres termes, deux objets sont distincts même si tous leurs attributs (nom, taille et couleur par exemple) ont des valeurs identiques (deux pommes vertes sont deux objets distincts). Par ailleurs, un objet évolue au cours du temps, comme, par exemple, une pomme peut mûrir. Chaque objet possède un état qui correspond à la valeur de ces attributs à un instant donné.

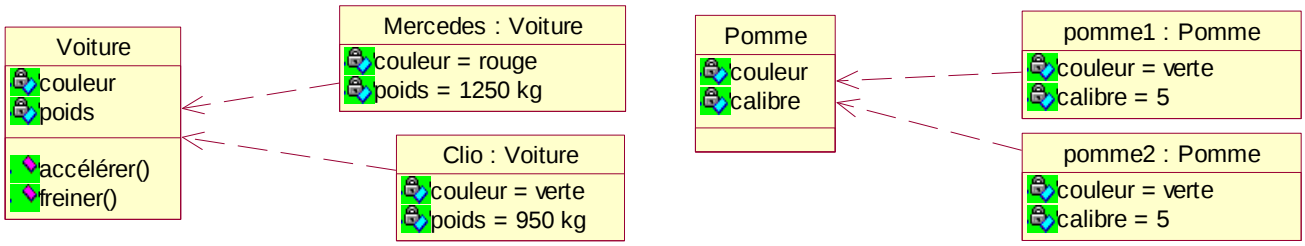
Objet = Identité + État + Comportement

Les classes :

La classification signifie que les objets ayant la même structure de donnée – ***attributs*** - et le même comportement - ***méthodes*** - sont regroupés en une classe. Les objets d'une classe ont donc le même type de comportement et les mêmes attributs. En groupant les objets en classe, on abstrait un problème. Les définitions communes (telles que le nom de la classe et les noms d'attributs) sont stockées une fois par classe plutôt qu'une fois par instance. Les méthodes peuvent être écrites une fois par classe, de telle façon que tous les objets de la classe bénéficient de la réutilisation du code. Par exemple, toutes les ellipses partagent les procédures de dessin, de calcul d'aire ou de test d'intersection avec une ligne.

> ***Nota***
> Une classe est un modèle utilisé pour créer plusieurs objets présentant des caractéristiques communes.

Chaque objet possède ses propres valeurs pour chaque attribut mais partage noms d'attributs et méthodes avec les autres objets de la classe. Par exemple une classe ***Voiture*** peut être définie comme ayant un des attributs ***couleur*** et comme une des méthodes ***freiner*** ; les objets associées à la classe voiture peuvent être : Mercedes rouge, Clio verte, etc. Ces objets ont le même comportement (***freiner***). On peut acheter un kilo de pommes vertes calibrées, chacune de ces pommes est un objet unique qui correspond à la classe ***Pomme*** avec comme attributs (***couleur***, ***calibre***).



Dans le langage C++, la relation qui existe entre la classe et les objets est la même que celle qui existe entre un type de données et des variables de ce type. On dit que l'objet est une ***instance*** (**variable**) de sa ***classe*** (**type**).

> ***Nota***
> Rappelons que la notion de variable informatique est identique à la variable mathématique et que la notion de type correspond à l'ensemble de définition. De façon plus concrète, la variable correspond à un emplacement mémoire, et il est nécessaire de connaître son type pour déterminer sa taille.

Les attributs :

Un attribut est une valeur de donnée détenue par les objets de la classe. ***Couleur*** et ***Poids*** sont des attributs des objets relatifs à ***Voiture***. Chaque attribut à une valeur pour chaque instance d'objet. Par exemple, l'attribut ***Couleur*** porte la valeur ***rouge*** dans l'objet ***Mercedes*** alors que pour l'objet ***Clio***, la valeur de l'attribut ***Couleur*** est ***verte***. Les instances peuvent avoir des valeurs identiques ou différentes pour un attribut donné. Chaque nom d'attribut est unique à l'intérieur d'une classe. Ainsi, la classe ***Voiture*** et la classe ***Pomme*** peuvent avoir chacune un attribut ***Couleur***.

> ***Nota***
> Dans une classe, les attributs sont définis par des variables. Les attributs peuvent être considérés comme des variables globales pour chaque objet de cette classe. Comme pour toutes variables, il est nécessaire de connaître le type correspondant et c'est la classe de l'objet qui indique de quel type d'attribut (variable) il s'agit. Chaque objet stocke sa propre valeur pour chacune de ses variables.

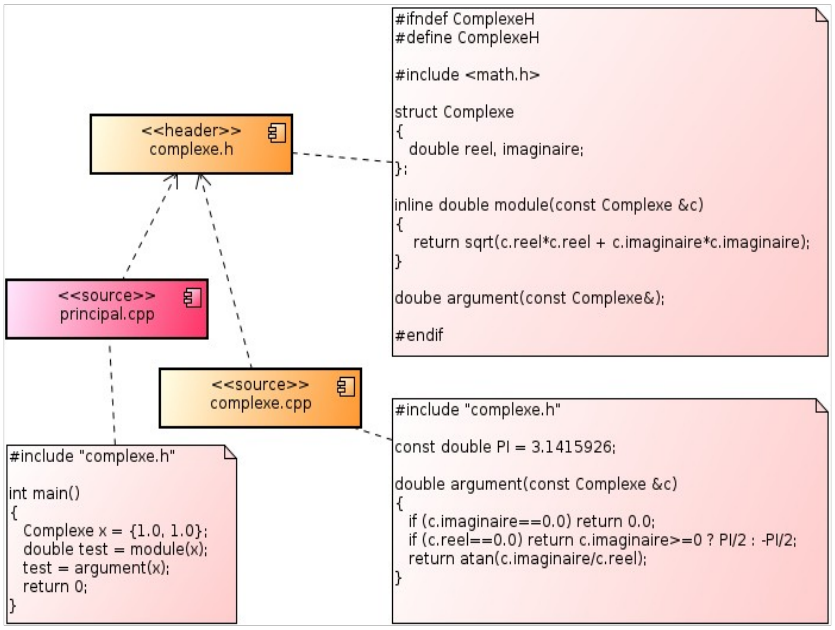
Les méthodes :
Une méthode est une fonction ou une opération qui peut être appliquée aux objets ou par les objets dans une classe. ***Accélérer*** et ***freiner*** sont des méthodes de la classe ***Voiture***. Tous les objets d'une même classe partagent les mêmes méthodes. Chaque méthode a un objet cible comme argument implicite (c'est l'objet lui-même « ***this*** », elle peut donc accéder à chacun des attributs). Le même nom de méthode peut s'appliquer à des classes différentes, vu que la portée de la méthode est sa classe. Une méthode peut avoir des arguments, en plus de son objet cible.

Retour à la programmation structurée

Avant de créer nos premiers objets, nous allons mettre en œuvre une structure qui représente un nombre complexe. A cette structure, nous allons lui associer des fonctions qui vont respectivement permettre de déterminer le module et l'argument de ce nombre complexe.

Ce programme fonctionne très bien, mais cette approche présente quelques inconvénients ; En effet, nous avons une séparation entre la structure d'une part, et les fonctions associées à cette structure d'autre part, alors que normalement tous ces éléments s'intéressent au même problème, c'est-à-dire, aux traitements des nombres complexes.

Si nous prenons le nom de la fonction ***module()*** ; ce nom peut aussi bien évoquer le module d'un nombre complexe ou peut-être le module d'un vecteur. Heureusement, la signature de la fonction nous indique qu'il s'agit bien d'une connexion à une structure ***Complexe***.



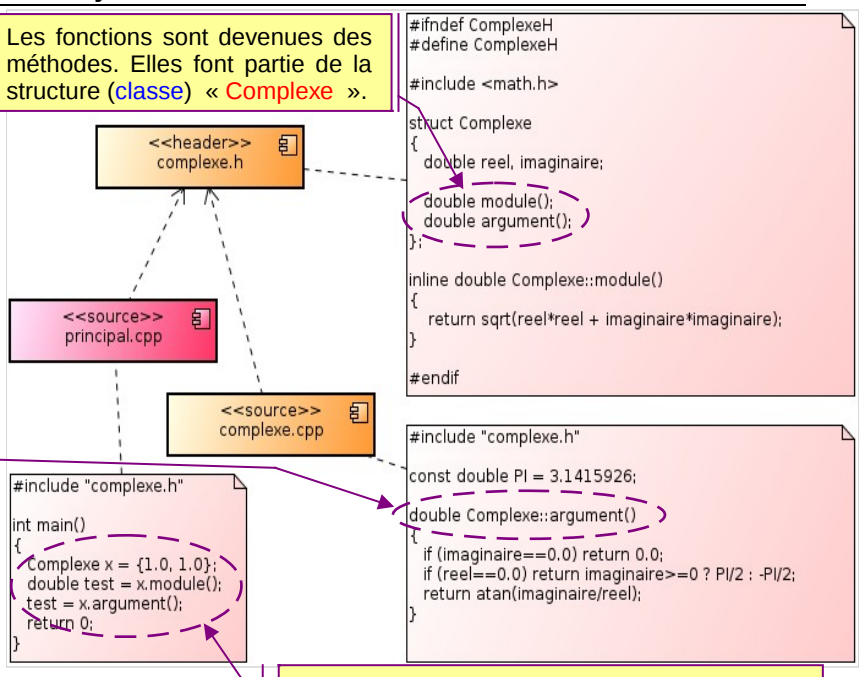
Enfin, lorsque nous fabriquons des fonctions qui gèrent des structures, il est systématiquement nécessaire de passer en paramètre la structure concernée. Cette façon de procéder présente l'énorme inconvénient d'avoir un temps de réponse conséquent et d'utiliser de la mémoire supplémentaire pour stocker momentanément cette structure sur la pile.

Même exemple avec une approche orientée objet

Pour toutes les raisons que nous venons d'évoquer, il serait souhaitable que les fonctions ***module()*** et ***argument()*** soient intégrées directement dans la structure. D'une part, le traitement demandé concerne cette structure. Par ailleurs, vu que ces fonctions sont à l'intérieur de la structure, elles peuvent atteindre directement les champs ***reel*** et ***imaginaire***. Le fait que tout soit intégré, les champs sont appelés des « ***attributs*** », les fonctions sont appelées des « ***méthodes*** ».

Les fonctions sont devenues des méthodes. Elles font partie de la structure (classe) « Complexe ».

argument n'est plus accessible directement puisque c'est une méthode. Pour sa définition, il faut préciser que cette méthode appartient à la classe ***Complexe***. L'opérateur de portée « **::** » spécifie l'appartenance.



Vous remarquez que, cette fois-ci, les méthodes ne possèdent pas de paramètres. Tout ce qu'elles ont besoin se situe dans la classe. Les attributs sont directement accessibles aux méthodes (et uniquement par elles d'ailleurs). En fait, les attributs et les méthodes sont sur la même portée, c'est-à-dire, la classe.

x devient un objet le la classe ***Complexe***. Vu que ***module()*** et ***argument()*** sont intégrés dans l'objet, au même titre d'ailleurs que les attributs ***reel*** et ***imaginaire***, l'utilisation d'un des éléments se passe comme d'habitude en spécifiant l'opérateur de séparation « **.** ».

Ces méthodes sont là pour permettre la communication avec l'extérieur. Souvent deux cas se présentent :

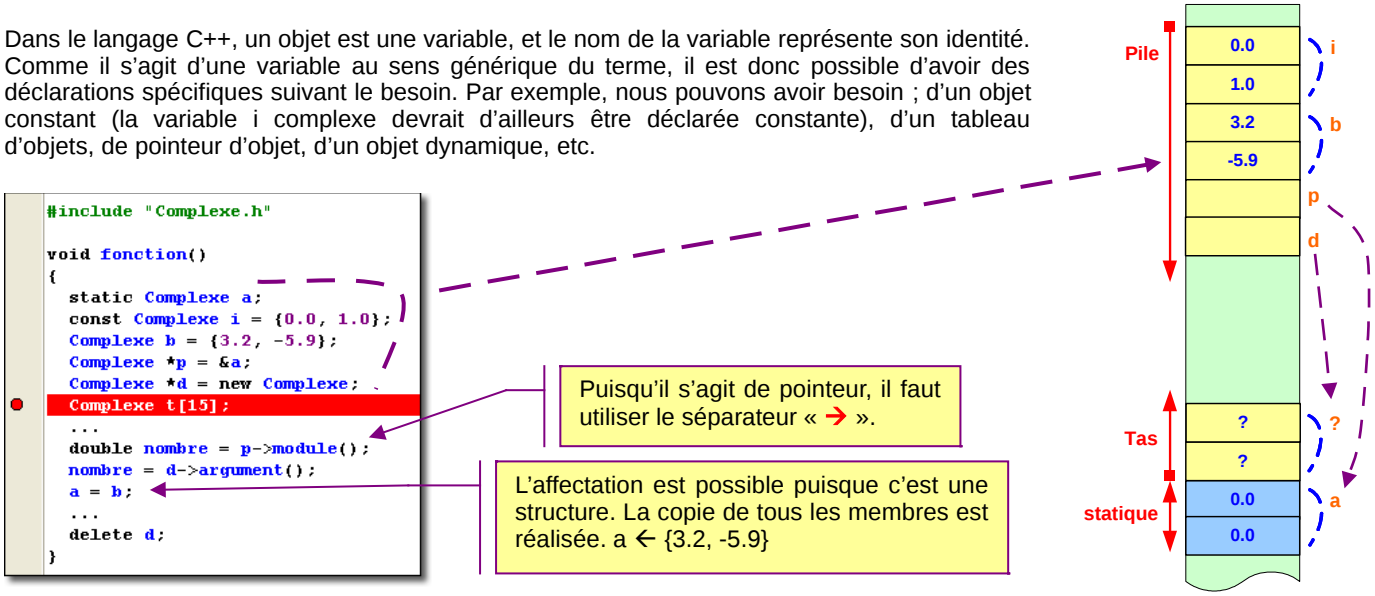
- Un élément externe a besoin d'un renseignement, il fait donc appel à la méthode adaptée (généralement elle ne possède pas de paramètres) qui retourne l'information désirée. C'est le cas des méthodes ***module()*** et ***argument()***.

- L'objet doit changer d'état. Il faut alors utiliser une méthode qui récupère une ou plusieurs valeurs depuis l'extérieur. Cela sous-entend que nous ayons besoin cette fois-ci de paramètres à la méthode pour récupérer les arguments demandés. Généralement ce type de méthode ne renvoie rien, puisque le but poursuivi est de modifier la valeur des attributs pour arriver à ce changement d'état.

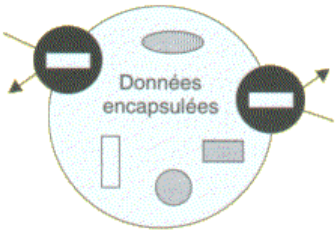
Il existe d'autres types de méthode comme les constructeurs et les destructeurs. Ces méthodes seront traitées ultérieurement. Tout ce que nous avons appris sur les fonctions s'applique pour les méthodes. Du point de vue du langage C++, une méthode reste une fonction, sauf qu'elle est intégrée dans une classe.

Différentes utilisations des objets

Dans le langage C++, un objet est une variable, et le nom de la variable représente son identité. Comme il s'agit d'une variable au sens générique du terme, il est donc possible d'avoir des déclarations spécifiques suivant le besoin. Par exemple, nous pouvons avoir besoin ; d'un objet constant (la variable i complexe devrait d'ailleurs être déclarée constante), d'un tableau d'objets, de pointeur d'objet, d'un objet dynamique, etc.



Encapsulation



L'encapsulation est le principe qui permet de regrouper les attributs et méthodes au sein d'une classe. Cette notion est aussi associée au dispositif de protection qui permet de contrôler la visibilité d'un attribut ou d'une méthode. En d'autres termes, cela signifie que chaque fois que vous définissez un ***membre*** d'une classe (***attribut*** ou ***méthode***), vous devez indiquer les droits d'accès quant à l'utilisation de ce membre.

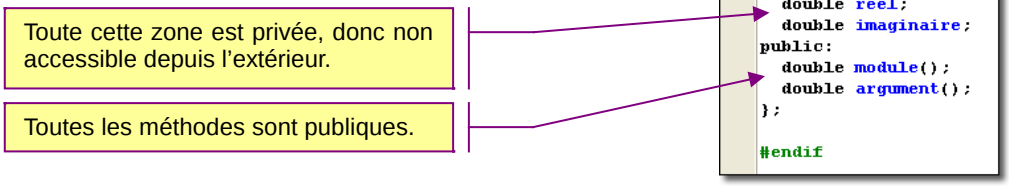
Ce mécanisme d'encapsulation permet surtout de protéger l'objet de toute malveillance externe. Pour cela, la plupart du temps, il faut interdire l'accès direct aux attributs et passer systématiquement par les méthodes. Par exemple : si l'on désire changer la couleur d'une voiture, cela ne se fait pas par enchantement, il est nécessaire de passer par tout un processus (décaper, poncer, passer plusieurs couches, etc...), et dans ce cas de figure l'attribut ***couleur*** ne doit pas être accessible directement.

Comme nous venons de l'évoquer, le changement d'état d'un objet passe nécessairement par la méthode associée. Cela n'a aucun sens de vouloir atteindre directement un attribut.

Il existe trois niveaux de protection :

- **public** : Tous les attributs ou méthodes d'une classe définies avec le mot clé ***public*** sont utilisables par tous les objets. Il s'agit du niveau le plus bas de protection. Ce type de protection est employé pour indiquer que vous pouvez utiliser sans contrainte les attributs et les méthodes d'une classe.
- **private** : Tous les membres d'une classe définis avec le mot clé ***private*** sont utilisables uniquement par les méthodes de la classe. Cette étiquette de protection constitue le niveau le plus fort de protection. Généralement les attributs doivent être déclarés comme privées.
- **protected** : Tous les membres d'une classe définis avec le mot clé ***protected*** sont utilisables uniquement par les méthodes de la classe et par les méthodes des classes dérivées (par les enfants). Cette technique de protection est fortement associée à la notion d'héritage (voir ultérieurement).

Par défaut, une structure possède le niveau de protection le plus faible, c'est-à-dire que tous les membres sont « ***publics*** ». Il faut donc rajouter à notre implémentation les qualificatifs nécessaires pour améliorer la protection et donc respecter le contrat d'encapsulation.




```
#ifndef ComplexeH
#define ComplexeH

struct Complexe
{
private:
  double reel;
  double imaginaire;
public:
  double module();
  double argument();
};

#endif
```

En fait, pour être sûr de respecter le principe d'encapsulation, il est généralement préférable d'utiliser une structure qui est privée par défaut. Il s'agit de la structure « ***class*** ». Dorénavant, nous utiliserons le mot réservé « ***class*** » plutôt que « ***struct*** ».

```
#ifndef ComplexeH
#define ComplexeH

class Complexe
{
  double reel;
  double imaginaire;
public:
  double module();
  double argument();
};

#endif
```

Zone privée par défaut.

Attention, ces écritures ne sont plus permises, puisque nous essayons d'atteindre directement les attributs. Il faut impérativement passer par des méthodes. Heureusement, il existe des méthodes adaptées pour répondre à ce genre de situation. Il s'agit des constructeurs qui seront traités ultérieurement.

```
#include "Complexe.h"

void fonction()
{
  static Complexe a;
  const Complexe i = {0.0, 1.0};
  Complexe b = {3.2, -5.9};
  Complexe *p = &a;
  Complexe *d = new Complexe;
  Complexe t[15];
  ...
  double nombre = p->module();
  nombre = d->argument();
  a = b;
  ...
  delete d;
}
```

Méthode « inline » :

Lorsque la définition d'une méthode est très courte, il est préférable de l'écrire « ***inline*** » comme pour une fonction. Il est impératif, bien entendu, qu'elle soit décrite dans le fichier en-tête. Il est même possible de donner la définition directement dans la déclaration de la classe. Il est souvent souhaitable de faire une séparation entre la déclaration et la définition.

```
#ifndef ComplexeH
#define ComplexeH

#include <math.h>

class Complexe
{
  double reel;
  double imaginaire;
public:
  double module();
  double argument();
};

inline double Complexe::module()
{
  return sqrt(reel*reel + imaginaire*imaginaire);
}

#endif
```

Définition de la méthode « inline ».

Déclaration de la méthode. Sa définition est décrite dans le fichier « Complexe.cpp » correspondant.

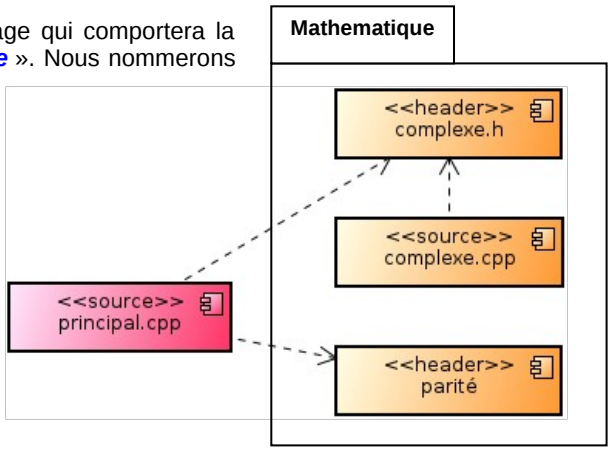
```
#ifndef ComplexeH
#define ComplexeH

#include <math.h>

class Complexe
{
  double reel;
  double imaginaire;
public:
  double module()
  {
    return sqrt(reel*reel + imaginaire*imaginaire);
  }
  double argument();
};

#endif
```

Les paquetages – espaces de noms

La programmation orientée objet est très séduisante. Elle est tellement séduisante que tous les langages de programmation qui sortent actuellement intègrent systématiquement cette philosophie. Ce type de programmation consiste à découper le projet en module. Chacun des modules est représenté par une classe. Au fur et à mesure du développement de projets multiples, nous pouvons nous retrouver avec une multitude de classes. Il peut même arriver que nous développions des classes différentes mais qui portent le même nom. Il faut pouvoir gérer ce genre de situation. Les espaces de nom sont là pour palier à ce problème.

Un espace de nom est un paquetage qui regroupe des classes et des fonctions dans une même entité, généralement représentant le même domaine. Le paquetage fait un peu penser à une bibliothèque. Il peut d'ailleurs être judicieux d'associer un paquetage à une bibliothèque. Le symbole du paquetage est un répertoire. Ainsi pour atteindre un des éléments (classes ou fonctions), il faut le référencer au travers du paquetage. Si deux classes comportent le même nom, la localisation ne pose plus de problème puisqu'elles sont situées dans des paquetages différents ce qui évite tout conflit de nom.

Pour illustrer mes propos, je vous propose de fabriquer un paquetage qui comportera la classe « ***Complexe*** » ainsi que deux fonctions « ***paire*** » et « ***impaire*** ». Nous nommerons ce paquetage « ***Mathématique*** ».



Le programme principal se situe à l'extérieur du paquetage. Il sera donc nécessaire de référencer le paquetage pour atteindre la classe et les fonctions de parités.

Définition d'un espace de noms :

Pour définir un nouveau paquetage (espace de nom), il suffit d'utiliser le mot réservé « ***namespace*** » et de placer entre les accolades tous les éléments qui font partis de ce paquetage. Il est possible de spécifier le même paquetage dans plusieurs fichiers. Il suffit, tout simplement d'indiquer l'espace de nom et, à chaque fois, d'introduire les éléments concernés entre les accolades.

Complexe.h

```
#ifndef ComplexeH
#define ComplexeH
#include <math.h>

namespace Mathematique {   // début du namespace

  class Complexe {
    double reel, imaginaire;
  public:
    double module();
    double argument();
  };

  inline double Complexe::module()
  {
    return sqrt(reel*reel + imaginaire*imaginaire);
  }
}    // fin du namespace

#endif
```

Complexe.cpp

```
#include "Complexe.h"

namespace Mathematique {  // début du namespace

  const double PI = 3.1415926;

  double Complexe::argument()
  {
    if (imaginaire==0.0) return 0.0;
    if (reel==0.0) return imaginaire>0 ? PI/2.0 : -PI/2.0;
    return atan(imaginaire/reel);
  }
}    // fin du namespace
```

Parite.h

```
#ifndef PariteH
#define PariteH

namespace Mathematique {
  inline bool impaire(int x) { return x%2; }
  inline bool paire(int x) { return !impaire(x); }
}

#endif
```

Principal.cpp

```
#include "Complexe.h"
#include "Parite.h"
//-------------------------------------
int main()
{
  Mathematique::Complexe c;
  double mod = c.module();
  bool test = Mathematique::impaire(13);
  return 0;
}
//-------------------------------------
```

Opérateur de portée « :: » :

La déclaration d'un membre d'espace de noms est cachée dans son espace de nom. Ainsi, la classe « ***Complexe*** » est cachée dans l'espace de nom « ***Mathematique*** ». A moins de préciser au compilateur dans quel espace de noms rechercher une déclaration, le compilateur cherche uniquement dans la portée locale.

Lorsque nous définissons une méthode, nous devons qualifier la classe qui gère cette méthode grâce à l'opérateur de portée « **::** ». De la même façon, nous devons qualifier l'espace de nom pour atteindre l'élément qui se trouve à l'intérieur, de nouveau grâce au même opérateur de portée « **::** ».

```
#include "Complexe.h"
#include "Parite.h"
//-------------------------------------
int main()
{
  using Mathematique::Complexe;
  using Mathematique::impaire;

  Complexe c;
  double mod = c.module();
  bool test = impaire(13);
  return 0;
}
//-------------------------------------
```

Déclaration « using » :

Avec cette écriture, il n'est plus possible d'avoir de conflit, puisque nous précisons bien que nous prenons tel élément de tel espace de noms. Toutefois, lorsque nous sommes sûr de ne pas avoir de conflit pour certains éléments, il est embêtant d'avoir une syntaxe aussi lourde. Il serait préférable d'indiquer dès le départ les éléments que nous souhaitons utiliser. Vous devez alors, grâce au mot réservé « ***using*** » lister les éléments que vous souhaitez atteindre directement sans le préfixe de l'espace de nom.

```
#include "Complexe.h"
#include "Parite.h"
using namespace Mathematique;
//-------------------------------------
int main()
{
  Complexe c;
  double mod = c.module();
  bool test = impaire(13);
  return 0;
}
//-------------------------------------
```

Utilisation de tous les éléments du paquetage

Directive « using » :

Finalement, il est peu fréquent d'avoir des conflits. Il est quand même souhaitable de prévoir au cas où. Cela ne coûte pas grand-chose. Toutefois, si nous sommes sûr qu'aucun des éléments de l'espace de noms ne présente de conflit avec l'extérieur, il serait plus intéressant de les rendre accessible en spécifiant le paquetage tout entier.

```
namespace {
  void interne() {...}
}
```

Espace de nom non nommé

Espace de nom non nommé :

En C++, nous pouvons utiliser un espace de noms non nommé pour déclarer une entité locale à un fichier. Cela sous-entend que cette entité ne sera plus du tout accessible depuis un autre fichier. Cette entité est donc « ***privée*** ». Elle ne peut être utilisée que par les autres entités du même fichier.

Espace de noms standard « std » :

En fait, tous les composants de la bibliothèque standard C++ sont déclarés et définis dans un espace de nom appelé « ***std*** ». Toutefois, avec la compilation conditionnelle, cet espace de nom n'est pas pris en compte lorsque nous effectuons l'inclusion suivante :

```
#include <iostream.h>
```

Si nous désirons que l'espace de nom « ***std*** » soit effectivement intégré, il faut plutôt préciser l'inclusion suivante :

```
#include <iostream>
```

Du coup, il sera nécessaire de préfixer chacun des éléments pour pouvoir les atteindre. Ainsi, pour atteindre l'objet « ***cout*** », il sera nécessaire d'écrire « ***std ::cout*** ». Du coup, pour éviter cette spécification, il vaut mieux utiliser la directive « ***using*** ». En fait, nous utiliserons fréquemment la syntaxe suivante :

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

Directive « using » pour définir de nouveaux types (C++11) :

Une des nouveautés du C++ 11 est de permettre de définir de nouveaux types à partir de ceux existants à l'aide de cette directive **using** qui bénificie de plusieurs fonctionnalités supplémentaires. Dans les anciennes versions du langage, nous utilisions la directive **typedef** issue du langage C. Cette ancienne approche n'est pas satisfaisante puisqu'elle ne fait que du traitement de texte. Voici un exemple qui permet de définir un nouveau type Octet représentant des caractères non signés.

```
using Octet = unsigned char ;
```