Vu le principe d'encapsulation, il est formellement interdit d'atteindre directement les attributs d'une classe. Il faut systématiquement passer par une méthode. Cette technique est judicieuse puisque le changement d'état d'un objet passe d'abord par un changement de comportement. Du coup, il n'est pas possible de réaliser une initialisation explicite en utilisant les accolades comme nous avions l'habitude de procéder avec les structures. Ne vous inquiétez pas, il existe des méthodes adaptées pour résoudre les initialisations explicites pour que l'objet soit dans l'état désiré. Ces méthodes spécifiques s'appellent des constructeurs.

```
#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;
}
```

Constructeur

Un constructeur est une méthode qui sera appelée automatiquement à chaque création d'un objet. Ceci se fera quel que soit la classe d'allocation de l'objet : statique, automatique (***pile***) ou dynamique (***tas***). Le constructeur se reconnaît parce qu'il porte le même nom que la classe. Attention, un constructeur ne doit rien renvoyer. Cela n'aurait aucun sens puisque c'est la phase de création.

Création des objets

Un constructeur est une méthode, donc, lorsque nous créons un objet, nous devons faire appel explicitement à cette méthode, en utilisant la syntaxe habituelle de l'appel des fonctions, c'est-à-dire en utilisant les parenthèses. Pour le nombre complexe, nous avons besoin de préciser les deux arguments vu la signature du constructeur que nous avons choisi.

Nous savons que nous avons la possibilité d'initialiser explicitement une variable dynamique. Dans le cas d'une variable primaire (***int*** - par exemple), il s'agit dans la syntaxe de préciser le type de la variable et de donner entre parenthèse la valeur initiale. Dans le cas d'un objet, il est nécessaire de passer par la phase de création. Nous faisons donc appel explicitement au constructeur avec le nombre d'arguments requis.

```
#ifndef ComplexeH
#define ComplexeH                          Complexe.h

#include <math.h>

class Complexe
{
  double reel;
  double imaginaire;          Déclaration du constructeur
public:
  Complexe(double x, double y);
  double module();
  double argument();
};                            Définition du constructeur

inline Complexe::Complexe(double x, double y)
{
  reel=x;  imaginaire=y;
}

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

#endif
```

> ***Nota***
> Lorsque nous écrivons : « ***int i = 5 ;*** » le compilateur transforme cette ligne en « ***int i(5) ;*** » parce qu'en C++ (contrairement au C) tous les types sont considérés comme des classes. Lors d'une initialisation explicite, c'est le constructeur préfabriqué qui comporte un seul paramètre qui est appelé.

```
#include "Complexe.h"

void fonction()
{                                  Création des objets
  static Complexe a;
  const Complexe i(0.0, 1.0);
  Complexe b(3.2, -5.9);
  Complexe *p = &b;
  Complexe *d = new Complexe(3.2, 5.0);
  Complexe t[15];
  ...
  double nombre = p->module();
  nombre = d->argument();
  a = b;
  ...
  delete d;
}
```

Vous remarquez dans le code précédent que certaines lignes ne fonctionnent plus correctement. Deux lignes provoquent des erreurs de compilation. Avant d'évaluer la raison de ce problème, nous allons étudier la forme canonique d'une classe.

Forme canonique d'une classe

Le mécanisme des classes est très puissant. Au moment où nous créons une nouvelle classe, quatre méthodes sont automatiquement intégrées pour assurer un comportement minimum de la classe sans aucune écriture explicite. C'est ce qui s'appelle la « forme canonique d'une classe ». Ces quatre méthodes sont :

```
//-----------------------------------------------------------
class T { }; // lorsque nous créons cette classe vide
//-----------------------------------------------------------
// nous avons en fait
class T
{
public:
  T();                          // constructeur par défaut
  T(const T&);                  // constructeur de copie
  ~T();                         // destructeur
  T& operator= (const T&); // opérateur d'affectation
};
//-----------------------------------------------------------
int main()
{
  T t1;        // t1 objet de T - construction par défaut
  T t2 = t1;   // t2 est construit à partir de t1 (copie)
  t1 = t2;     // utilisation de l'opérateur d'affectation
}              // appel des destructeurs à ce niveau
//-----------------------------------------------------------
```

- ***Constructeur par défaut*** : le constructeur par défaut est un constructeur qui ne possède pas de paramètre et donc qui n'attend pas d'argument. Par défaut, ce constructeur ne fait rien. Si vous désirez avoir un comportement particulier, il sera nécessaire de le redéfinir. Toutefois, ce constructeur n'existe plus à partir du moment au vous définissez un nouveau constructeur qui possède un nombre quelconque d'arguments. Si vous désirez, malgré tout, posséder, en plus, un constructeur par défaut, il sera également nécessaire de le redéfinir.
- ***Constructeur de copie*** : il est possible de créer un objet à partir d'un autre objet (bien entendu, de la même classe). Ce constructeur attend l'objet en paramètre et copie chacun de ces attributs pour les associés à ses propres attributs. Le comportement par défaut est donc une copie membre à membre.

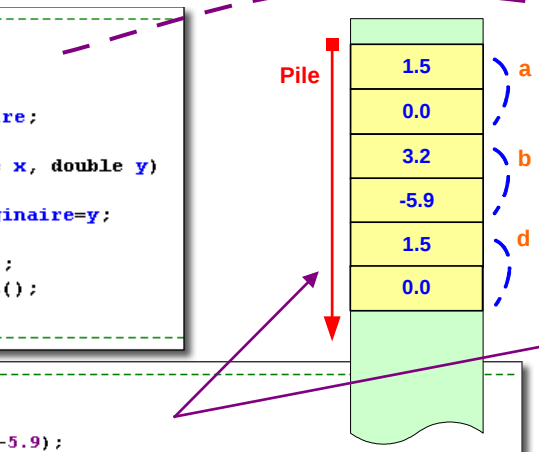
- ***Destructeur*** : de même qu'il existe une phase de construction, il existe systématiquement une phase de destruction. Ce mécanisme est utile lorsque nous avons besoin de gérer des variables dynamiques pour notamment leurs libérations. Un destructeur se reconnaît parce qu'il porte le même nom que la classe précédé du tilde « ~ ».
- ***Opérateur d'affectation*** : nous avons déjà utilisé cet opérateur lorsque que nous avons réalisé des affectations (des copies) de structures. Par défaut, comme pour le constructeur de copie, l'affectation propose une copie membre à membre. Ce comportement est très intéressant puisque nous n'avons pas besoin d'écrire quoi que se soit et c'est généralement ce que nous avons besoin. Il existe, malgré tout, des situations où ce comportement n'est pas souhaitable. Dans ce cas là, il sera nécessaire de redéfinir cet opérateur.

Toute l'étude que nous avons faite sur les fonctions s'applique pour les méthodes. A ce sujet, il est donc possible de faire de la sur-définition et donc d'avoir, par exemple, plusieurs constructeurs. C'est justement ce qui se passe avec la forme canonique puisque deux constructeurs coexistent. Nous pouvons également utiliser des paramètres par défaut.

Retour sur la classe « Complexe » avec le principe de la forme canonique

Lorsque nous écrivons la classe « ***Complexe*** » comme ci-dessous à gauche, en réalité, nous avons ce qui est décrit sur la partie droite.

```
//----------------------------------
class Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe(double x, double y)
  {
    reel=x;  imaginaire=y;
  }
  double module();
  double argument();
};
//----------------------------------
```




```
//-----------------------------------------------------
void test()
{
  Complexe a(3.2, -5.9);
  Complexe b = a; // construction de copie, autre écriture b(a);
  Complexe c;  // non admis, plus de construction par défaut
  Complexe d(1.5, 0.0);
  a = d;  // affectation, copie des attributs
}  // trois destructions sont demandées
//-----------------------------------------------------
```

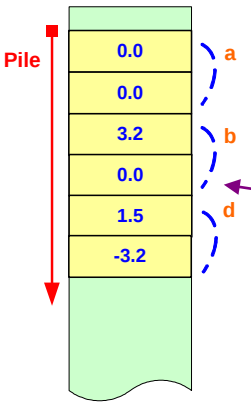
```
//-----------------------------------
class Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe() { }
  Complexe(const Complexe &c)
  {
    reel = c.reel;
    imaginaire = c.imaginaire;
  }
  ~Complexe() { }
  Complexe& operator= (const Complexe &c)
  {
    reel = c.reel;
    imaginaire = c.imaginaire;
    return *this;
  }
  Complexe(double x, double y)
  {
    reel=x;  imaginaire=y;
  }
  double module();
  double argument();
};
//-----------------------------------
```

Le constructeur par défaut est occulté par le constructeur défini par l'utilisateur

Définitions de plusieurs constructeurs

Grâce à la sur-définition, il serait souhaitable de répondre à l'attente des utilisateurs, en proposant d'autres constructeurs, pour répondre à tous les cas de figure, c'est-à-dire :

- Pouvoir construire un nombre complexe sans préciser de valeurs initiales. Cela consiste à définir un constructeur par défaut. Ce nombre complexe doit représenter le point origine.
- Pouvoir construire, tout simplement un nombre réel, puisque les réels sont compris dans l'ensemble de définition des nombres complexes. Ce constructeur prendra un seul argument. La partie imaginaire doit être nulle.




```
//-----------------------------------------------------------
class Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe()                    // constructeur par défaut
  {
    reel = imaginaire = 0.0;
  }
  Complexe(double x)        // constructeur avec un argument
  {
    reel = x;
    imaginaire = 0.0;
  }
  Complexe(double x, double y) // constructeur avec 2 arguments
  {
    reel=x;  imaginaire=y;
  }
  double module();
  double argument();
};
//-----------------------------------------------------------
void test()
{
  Complexe a;             // construction par défaut
  Complexe b = 3.2;       // équivalent à b(3.2);
  Complexe c(1.5, -3.2);
}
//-----------------------------------------------------------
```

Arguments par défaut

Au lieu d'avoir trois constructeurs différents, nous pouvons utiliser les arguments par défaut, et du coup, définir un seul constructeur qui traitera tous les cas de figure désirés.

```
//-----------------------------------------------------
class Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe(double x = 0.0, double y = 0.0)   // Valeurs par défaut
  {
    reel=x;  imaginaire=y;
  }
  double module() const;
  double argument() const;
};
//-----------------------------------------------------
void test()
{
  Complexe a;            // construction par défaut
  Complexe b = 3.2;      // équivalent à b(3.2);
  Complexe c(1.5, -3.2);
}
//-----------------------------------------------------
```

Importance du constructeur par défaut

Quoi qu'il arrive, la création d'un objet doit toujours passer par une construction. C'est d'ailleurs pour cela qu'un constructeur par défaut existe, même si le programmeur n'en fabrique pas.

Revenons sur les tableaux d'objets. Deux cas peuvent se présenter. Soit nous devons déclarer un tableau d'objet sans initialisation explicite. A ce moment là, c'est le constructeur par défaut qui est sollicité pour construire chacun des objets en particulier. Si vous désirez effectuer une initialisation explicite, il suffit d'utiliser la syntaxe classique, c'est-à-dire, en utilisant les accolades, et de préciser pour chacun des objets, le constructeur adapté.

```
#include "Complexe.h"
//-----------------------------------------------------
void fonction()
{
  Complexe t1[4];                    // Constructeurs par défaut
  ...
  Complexe t2[3] = { Complexe(2.3, 5.6), -8.9 };
  ...                                // Initialisation explicite d'un tableau de trois complexes
  ...                                //   t2[0] <-- (2.3, 5.6)
  ...                                //   t2[1] <-- (5.6, 0.0)
  Complexe *t3 = Complexe[3];        //   t2[2] <-- (0.0, 0.0)
  ...                                // Constructeurs par défaut
  delete[] t3;
}
//-----------------------------------------------------
```

> ***Nota***
> Souvenez-vous qu'il n'est pas possible d'initialiser explicitement chacune des cases d'un tableau dynamique. C'est bien sûr également vrai pour les tableaux d'objet, d'où l'importance d'avoir un constructeur par défaut.

Les objets constants

Nous pouvons avoir besoin de définir des objets constant comme cela a été le cas avec la variable complexe « ***i*** ». Le principe même d'un objet constant, c'est que ces attributs demeurent inchangés pendant toute sa durée de vie. Le principe de protection impose par défaut qu'un objet constant ne puisse utiliser aucune des méthodes, puisque l'une d'entre elle peut éventuellement proposer un changement d'état à l'objet, c'est-à-dire, modifier ses attributs.

Toutefois, il existe des méthodes qui ont un rôle uniquement consultatif, qui donc, par essence, ne change pas l'état de l'objet. Il serait intéressant que ces méthodes puissent être utilisées par ces objets constants. C'est notamment le cas pour la classe complexe qui utilisent deux méthodes consultatives '***module***' et '***argument***'.

Lorsque vous décidez qu'un objet constant puisse utiliser certaines méthodes, ces dernières doivent être qualifiées de constantes. Les autres méthodes de la classe ne pourront donc pas être utilisées. Pour les objets qui ont le statut de variable (c'est-à-dire, non constant), ils pourront utiliser indifféremment les méthodes classiques et les méthodes constantes. Pour qu'une méthode devienne constante, il suffit de placer le suffixe « ***const*** » après sa signature.

> ***ATTENTION***
> Une méthode constante ne doit en aucun cas modifier un des attributs sinon le compilateur donnerait une erreur de compilation, ce qui va, bien entendu, dans le sens de nos propos.

> ***Nota***
> Finalement, il peut être judicieux de définir toutes les méthodes consultatives comme des méthodes constantes, ce qui permettra par leurs signatures de les reconnaître.

```
#include "Complexe.h"
//-----------------------------------
void fonction()
{
  Complexe a;
  const Complexe i(0.0, 1.0);
  ...
  double nombre = i.module();
  nombre = i.argument();
  i.decalage(5.0, -3.2);
  ...                    // Cette opération est interdite
  double nombre = a.module();
  nombre = a.argument();
  a.decalage(5.0, -3.2);
  ...
}
//-----------------------------------
```

```
#ifndef ComplexeH
#define ComplexeH

#include <math.h>
//-----------------------------------------------------
class Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe(double x = 0.0, double y = 0.0);
  double module() const;
  double argument() const;             // Méthodes constantes
  void decalage(int dx, int dy);
};
//-----------------------------------------------------
inline Complexe::Complexe(double x, double y)
{
  reel=x;  imaginaire=y;
}
//-----------------------------------------------------
inline double Complexe::module() const
{
  return sqrt(reel*reel + imaginaire*imaginaire);
}                                      // Cette méthode ne doit pas changer la valeur d'un attribut
//-----------------------------------------------------
inline double Complexe::decalage(int dx, int dy)
{
  reel += dx; imaginaire += dy;
}
//-----------------------------------------------------
#endif
```

Les classes en C++11

Les classes sont un élément essentiel du C++. Les changements apportés par la norme C++11 facilitent et sécurisent la mise en œuvre d'une hiérarchie de classe.

En C++ standard, un constructeur ne peut pas en appeler un autre de la même classe : chacun de ces constructeur a la responsabilité d'initialiser tous les attributs de la classe (afin d'éviter que l'objet soit dans un état quelconque), ce qui implique souvent de dupliquer le code d'initialisation.

Grâce au C++11, un constructeur peut en appeler d'autres. Ce mécanisme est connu sous le nom de **«délégation»** et réutilise le comportement d'un constructeur existant.

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

class Complexe
{
  double reel;
  double imaginaire;
public:
  Complexe(double r, double i) { reel = r; imaginaire = i; }
  Complexe() : Complexe(0.0, 0.0) {}
  void affiche() const { cout << "<" << reel << ", " << imaginaire << '>'; }
};

int main()
{
  Complexe a;
  const Complexe i(0.0, 1.0);
  Complexe *p = new Complexe(5.3, -2.8);
  Complexe tabComplexes[5];
  cout << "a = "; a.affiche();
  cout << endl << "i = "; i.affiche();
  cout << endl << "p = "; p->affiche();
  cout << endl << "tabComplexe = [";
  for (Complexe nombre : tabComplexes) { nombre.affiche(); cout << ' '; }
  cout << "\b]\n";
  delete p;
  return 0;
}
```

Délégation de constructeur : Le constructeur par défaut appelle explicitement le premier constructeur avec ses deux paramètres. Les arguments proposés sont respectivement 0,0 et 0,0.

```
Terminal
a = <0, 0>
i = <0, 1>
p = <5.3, -2.8>
tabComplexe = [<0, 0> <0, 0> <0, 0> <0, 0> <0, 0>]
Appuyez sur <ENTRÉE> pour fermer cette fenêtre...
```

Si les objets d'une même classe doivent systématiquement être dans un même état au départ, il est dorénavant possible ***d'initialiser directement les attributs***. Dans ce cas là, vous n'êtes pas obligé de passer par un constructeur. Toutefois, bien entendu, nous pouvons mixer les deux approches, en prévoyant le cas général, et en rajoutant un ou plusieurs constructeurs qui permettent d'initialiser l'objet pour des cas particuliers.

```cpp
class Complexe
{
  double reel = 0.0;
  double imaginaire = 0.0;
public:
  Complexe(double r, double i) { reel = r; imaginaire = i; }
  Complexe() {}
  void affiche() const {
      cout << "<" << reel << ", " << imaginaire << '>';
  }
};
```

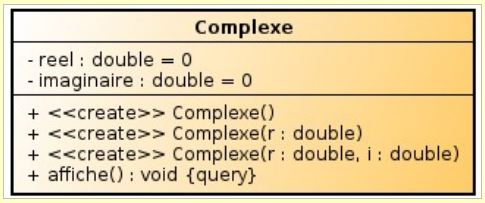
Initialisation explicite des attributs : Tous les objets créés à l'aide du constructeur par défaut (qui ne fait rien ici) auront les attributs assignés à 0,0 et 0,0. Si nous utilisons l'autre constructeur, les attributs prennent les valeurs données en paramètre, les valeurs nulles ne sont alors pas pris en compte.

Nous avons vu au début de cette étude que dès que vous définissez un nouveau constructeur, le constructeur par défaut n'est plus accessible, vous êtes obligé de le redéfinir si vous souhaitez avoir un comportement par défaut. C++11 permet de forcer (ou d'empêcher – voire dans une autre étude) la génération des méthodes par défaut comme le constructeur par exemple. Il suffit pour cela de faire une déclaration explicite à l'aide du mot réservé ***default***.

C++11 met en place une uniformisation de l'initialisation des types. Ainsi maintenant nous pouvons initialiser une classe comme nous le faisions avec une structure, en utilisant le signe égal et les accolades.

```cpp
class Complexe
{
  double reel = 0.0;
  double imaginaire = 0.0;
public:
  Complexe(double r, double i) { reel = r; imaginaire = i; }
  Complexe(double r) { reel = r; }
  Complexe() = default;
  void affiche() const {
      cout << "<" << reel << ", " << imaginaire << '>';
  }
};

int main()
{
  Complexe a;
  Complexe r = 5.3;
  const Complexe i = {0.0, 1.0};
  Complexe *p = new Complexe(5.3, -2.8);
...
}
```



Avec default, vous indiquez que vous désirez utiliser le constructeur par défaut, qui doit donc exister. Cela permet d'y accéder même si vous avez défini un autre constructeur (qui, en temps normal, masque ce constructeur par défaut).

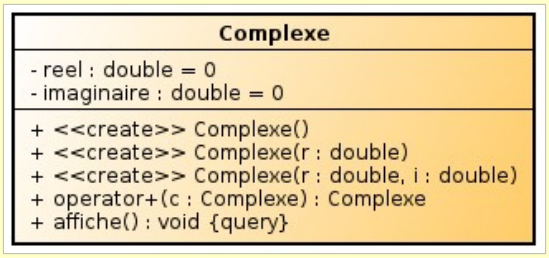
Initialisation de l'objet constant (appel au constructeur) avec la même syntaxe utilisée lors de l'initialisation d'une structure.

Cette uniformisation de l'initialisation des types peut être utilisée dans d'autre cas de figure. Il s'agit en réalité d'une conversion implicite automatique qui nous permet d'omettre le type (la classe) éventuellement. Ainsi, nous pouvons maintenant utiliser l'initialisation à tout moment, pas seulement durant la phase de création de l'objet, avec l'opérateur d'affectation par exemple (à ne pas confondre avec l'opérateur d'initialisation). Le compilateur va alors rechercher si il existe un constructeur qui prend comme arguments ce qui est proposé dans la liste d'initialisation (***nous voyons bien que cette initialisation est bien finalement une phase de création – d'où son nom - quelque soit l'endroit où nous l'utilisons puisque le compilateur fait systématiquement appel à un constructeur***).

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

class Complexe
{
  double reel = 0.0;
  double imaginaire = 0.0;
public:
  Complexe(double r, double i) { reel = r; imaginaire = i; }
  Complexe(double r) { reel = r; }
  Complexe() = default;
  Complexe operator+ (const Complexe &c)
  {
     return { reel+c.reel,  imaginaire+c.imaginaire };
  }
  void affiche() const
  {
      cout << "<" << reel << ", " << imaginaire << '>';
  }
};

int main()
{
  Complexe a, b;
  Complexe r = 5.3;
  const Complexe i = {0.0, 1.0};
  Complexe *p = new Complexe(5.3, -2.8);
  a = {3.5, 7.2}; // équivalent à =>  a = Complexe(3.5, 7.2)
  b = r + i;
  cout << "a = "; a.affiche();
  cout << endl << "r = "; r.affiche();
  cout << endl << "i = "; i.affiche();
  cout << endl << "b = "; b.affiche();
  cout << endl << "p = "; p->affiche();
  cout << endl;
  delete p;
  return 0;
}
```



Il est possible de créer un nouvel objet de type *Complexe* uniquement en se servant d'une liste d'initialisation. Le compilateur reconnaît qu'il existe effectivement un constructeur avec deux paramètres de type *double*.

Ici aussi, grâce à la liste d'initialisation, un nouvel objet de type *Complexe* est automatiquement généré pour que l'affectation puisse se faire correctement. Il doit y avoir deux éléments de même type de chaque côté du signe « = ».