Université de Sherbrooke

Jeux et Défis Informatiques de Sherbrooke

Atelier d'introduction au C++

Auteurs

Hugo Bédard
Émile Fugulin
David Lalancette
Julien Rossignol

30 janvier 2016

Table des matières

Résumé

Ce document a été réalisé dans le but d'aider les débutants en C++ à plonger plus facilement dans ce language complexe, mais tellement performant et complet. À la base, ce document a servi d'appui lors d'un atelier d'introduction au C++ donné à l'Université de Sherbrooke à l'hivers 2016 principalement pour les étudiants de première année. Il est à noter que le document a été préparé par des étudiants (nous) pour des étudiants. Par conséquent, le département de Génie Électrique et Informatique de l'Université de Sherbrooke n'est ni partenaire, ni ne considère ce document comme une référence. Par contre, l'information qu'il contient est tout à faite valide et, à notre avis, mieux expliquée que dans les documents officiels...

Nous espèrons que ce document pourra vous aider à approcher d'une façon plus rapide et moins douloureuse le language C++. Bonne Lecture !

Hugo Bédard
Émile Fugulin
David Lalancette
Julien Rossignol

Jeux et Défis Informatiques de l'Université de Sherbrooke (`www.jdis.ca`)

Première partie

INTRODUCTION AU C++

1 Entrée et sortie d'un programme c++

1.1 cin

cin est un objet de la librairie standard iostream. Il permet facilement d'assigner des variables avec l'entrée d'une console. Il suffit d'utiliser l'opérateur (»).

```
1 #include <iostream>
2 int main()
3 {
4   int bar = 0;
5   std::cin >> bar; //Assigne la variable selon l'entree de la console
6 }
```

Vous pouvez remarquer que cin est intelligent. Il essaiera automatiquement de faire l'assignation de la variable avec le bon type.

1.2 cout

cout est un objet de la librairie standard iostream. Il permet facilement d'imprimer des variables ou du texte dans la console. Il suffit d'utiliser l'opérateur («).

```
1 #include <iostream>
2 int main()
3 {
4   int bar = 0;
5   std::cout << "Voici la variable bar: " << bar << std::endl;
6   //Imprime "Voici la variable bar: 0" avec un saut de ligne a la fin
7 }
```

1.3 std namespace

Comme vous l'avez peut-être remarqué dans les exemples précédents, on utilise toujours le préfixe std : : avant d'utiliser cout, cin ou endl. Cela évite les collisions de noms de fonctions/variables si vous auriez par exemple défini vous même des variables cout, cin ou endl.

Cependant, vous pouvez évitez de le réécrire le préfixe à chaque en fois en spécifiant au programme que vous voulez toujours utiliser les fonctions dans le namespace std.

```
1 #include <iostream>
2 int main()
3 {
4   using namespace std;
5   int bar = 0;
6   cout << "Voici la variable bar: " << bar << endl;
7 }
```

2 Pointeurs

Les pointeurs permettent d'assigner l'adresse d'une variable avec l'opérateur (&) dans une autre variable (pointeurs), cela permet de modifier la valeur de l'adresse pointé en déréférencant le pointeur avec l'opérateur (*).

```
1  #include <iostream>
2  int main()
3  {
4    int bar = 0;
5    int* foo = &bar; // Assigne l'adresse de bar a foo
6    *foo = 1; //Assigne 1 à l'adresse que contient foo
7    std::cout << bar; //prints 1
8  }
```

Vous pouvez également définir les pointeurs dans les signatures de fonctions :

```
1  #include <iostream>
2  void foo(int* bar)
3  {
4    *bar = 1;
5  }
6
7  int main()
8  {
9    int bar = 0;
10   foo(&bar);
11   std::cout << bar; //prints 1
12   std::cin >> bar;
13 }
```

Pour modifier des valeurs à l'intérieur d'un pointeur d'objet ou de struct vous devez déférencez ces valeurs avec l'opérateur flèche (->).

```
1  #include <iostream>
2  struct Foo
3  {
4    int bar;
5  };
6
7  int main()
8  {
9    Foo foo;
10   foo.bar = 0;
11   Foo* pointerFoo = &foo;
12   pointerFoo->bar = 1;
13   std::cout << foo.bar; //prints 1
14   std::cin >> foo.bar;
15 }
```

L'opérateur flèche (->) est l'équivalent de déréférencer le pointeur avant d'accèder à la variable de la struct ou de la classe. Il est également possible de l'écrire de la manière suivante :
(*pointerFoo).bar = 1 ;
Par contre, cette syntaxe est beaucoup plus laide !

3 Références

Les références permettent de copier l'adresse et la valeur d'une variable dans une autre variable. Il existe trois différences majeures avec les pointeurs :

- Une référence ne peut être null
- Une fois que la référence est initialisé, elle ne peut être assigner à une autre variable.
- Une référence doit être initialisé lorsqu'elle est défini.

```
1 #include <iostream>
2 int main()
3 {
4   int bar = 0;
5   int& foo = bar; // Initialise lorsque defini.
6   foo = 1;
7   std::cout << bar; //prints 1
8 }
```

Vous pouvez également définir les références dans les signatures de fonctions :

```
1  #include <iostream>
2  void foo(int& bar)
3  {
4    bar = 1;
5  }
6
7  int main()
8  {
9    int bar = 0;
10   foo(bar);
11   std::cout << bar; //prints 1
12 }
```

4 Appel de fonction

Lorsqu'une fonction est appelé, l'ensemble des variables qui sont passées en paramètres sont copiés dans la mémoire. La fonction travaillera donc avec les mêmes valeurs, mais à des endroits différent de la mémoire.

```
#include <iostream>
int Foo(int bar)
{
  bar += 1;
  return bar;
}

int main()
{
    bar = 0;
    foobar = foo(bar);
    std::cout << bar; //prints 0
    std::cout << foobar; //prints 1
}
```

Ainsi, dans cet exemple, la valeur de bar est copié lorsque la fonction Foo est appelé. La fonction foo modifie sa propre copie de bar, mais la valeur de bar de la fonction main reste inchangé.

Dans ce deuxième exemple, nous allons vous montrer une utilisation courante des pointeurs.

```
#include <iostream>
void Foo(int* foobar)
{
  *foobar += 1;
}

int main()
{
    bar = 0;
    foo(&bar);
    std::cout << bar; //prints 1
}
```

Cet un exemple similaire à celui que vous trouvez dans la section des pointeurs. Ici, la valeur qui est copié lors de l'appel de Foo n'est pas la valeur de bar, mais bien son adresse. Ainsi, la fonction Foo modifie la valeur à l'adresse contenu dans foobar (c'est l'adresse de bar !). Il modifie donc bar.

Finalement, supposons que j'ai un objet environnement qui contient des milliers de paramètres sur la faune, la flore et la météo de Sherbrooke.

```
1  void Foo(environnement ville)
2  {
3      ...
4  }
5
6  int main()
7  {
8      environnement Sherbrooke();
9      Foo(Sherbrooke);
10 }
```

Dans cet exemple, TOUTES les informations de Sherbrooke sont copiés dans une nouvelle variable ville. Compte tenu de la grosseur de Sherbrooke, la copie peut-être très longue et ralentir considérablement votre programme. C'est donc dans cet optique que vous utiliserez le plus souvent les pointeurs.

```
1  void Foo(environnement* ville)
2  {
3      ...
4  }
5
6  int main()
7  {
8      environnement* Sherbrooke = new environnement();
9      Foo(Sherbrooke);
10 }
```

Ici, seule la valeur de l'adresse de Sherbrooke est copiée, il est donc beaucoup plus rapide de procéder ainsi.

Deuxième partie

PROGRAMMATION ORIENTÉ-OBJET

5 Classes

5.1 Définitions

5.1.1 Membres

On qualifie de *membre* toute fonction ou variable qui fait partie d'une classe. Pour marquer la différence entre une variable normale et une variable membre, on utilisera généralement le **m_** ou simplement l'underscore (_) au début du nom de la variable. Par convention, les noms des fonctions commenceront généralement par une minuscule et chaque mot suivant aura sa première lettre en majuscule. Notons que le nom *méthode* est aussi utilisé pour parler d'une fonction membre.

```
1 int maFonction();
2 int m_maVariable;
```

5.1.2 Portée

Par portée, on veut signifier ce qui peut ou ne peut pas être accéder à l'extérieur de la classe. Il existe trois types de portées en C++ :

Public : La portée *Public* permet aux agents externes d'accéder sans restriction à ses variables et à ses fonctions membres.

Private : La portée *Private* empêche tout agent extérieur à la classe d'accéder à ses variables et à ses méthodes. Cette portée est à la base de l'encapsulation tel que discuté précédemment.

Protected : La portée *Protected* est très similaire à la portée *Private*. La seule différence se trouve au niveau des classes filles (Voir la section 6 sur l'héritage). En effet, la portée *Protected* permet aux classes filles d'accéder aux variables et fonctions membres protected de la classe mère. Pour les autres agents, elle agit de la même façon que la portée *Private*.

Si aucune portée n'est spécifiée, la portée *Private* est celle par défaut dans les classes. Dans les structures, c'est plutôt la portée *Publique* qui est la portée par défaut.

NOTE IMPORTANTE : La norme en C++ est de ne jamais exposer les variables membres de façon publique sauf cas exceptionnel, car cela contrevient à l'esprit même de l'orienté-objet qui veut faire abstraction des données pour se concentrer sur les services ! ! !

5.2 Structure

La création d'une classe comporte généralement deux parties. D'abord, la déclaration de la classe qui se fait dans le ficher header (le fichier **.h**) et l'implémentation qui se fait dans le fichier source (le fichier **.cpp**).

5.2.1 Déclaration

La déclaration d'une classe est très simple. Il s'agit d'abord de déclarer avec le nom *Class* que l'on désire créer une classe. Ensuite, on lui donne un nom. Généralement on préférera un nom commençant par une

majuscule et dont chaque mot supplémentaire commencera aussi par une majuscule. Entre les accolades, on doit d'abord spécifier la portée puis les variables et fonctions membres. Finalement, il ne faut pas oublier le point-virgule de fin.

```
#ifndef FOOBAR_H
#define FOOBAR_H

class FooBar
{
  public:
    int fonctionBidon(//parametres);
  private:
    int m_maVariable;
};

#endif
```

NOTE IMPORTANTE : Vous vous demandez peut-être à quoi servent les mots qui commencent par un #. Il s'agit d'abord de les nommer, ce sont des directives de pré-processeur. En français, cela veut que ce sont des opérations qui sont exécutées avant que le compilateur commence son travail. Les directives utilisées ici sont communément appelées header gard. Elles permettent d'éviter les boucles infinies d'inclusions. Késako ? Prenons un exemple simple pour illustrer ce problème. Imaginez que vous avez deux classes A et B comme suit :

```
//Fichier A.h
#include "B.h"
class A
{
};

//Fichier B.h
#include "A.h"
class B
{
};
```

Ici, quand on vient pour exécuter le include, le pré-processeur regarde *A.h* et voit qu'il doit inclure *B.h*. Il va donc voir *B.h* et voir qu'il doit inclure *A.h*. Il retourne donc dans *A.h* et tourne en rond indéfiniment. Le header garde prévient ce problème, car la première fois *FOOBAR_H* n'est pas défini donc le pré-processeur continue de lire le fichier, mais la deuxième fois qu'il revient, *FOOBAR_H* est défini donc il arrête tout de suite de lire le fichier.

5.2.2 Implémentation

Une fois la classe déclarée, il faut implémenter ses différentes fonctions. Cela se fait dans le fichier source :

```
#include FooBar.h

int FooBar::fonctionBidon(//parametres)
{
    //Faire quelque chose et retourner un int
}
```

Ici, nous avons un nouvel opérateur, le **::**. Ce dernier permet de spécifier au compilateur que nous sommes en train d'implémenter la méthode *fonctionBidon* de la classe *FooBar*. En effet, nous pourrions très bien créer une fonction nommée *fonctionBidon* en dehors toute classe (en C++, nous appelons ce genre de

fonctions des fonctions libres) et le compilateur ne pourrait alors pas savoir si nous voulons implémenter la fonction libre ou la méthode de la classe *FooBar*.

5.3 Constructeur/Destructeur

Le constructeur et le destructeur jouent un rôle primordiale dans la création des classes. Le premier est exécuté lors de la création de l'objet et sert à initialiser la classe. Le second est appelé lorsque l'objet est détruit et sert principalement à détruire des variables qui auraient été initialisée dynamiquement (avec le *new*).

```
1  //Header file
2  class FooBar
3  {
4    public:
5      Foobar(//Parametres);
6      ~FooBar():
7  };
```

En C++, le constructeur porte le même nom que la classe et n'a pas de valeur de retour. Le destructeur, quant à lui, porte aussi le même nom que la classe, mais est précédé d'un tilde. Il est à noter que l'on veut généralement donner les paramètres d'entrée au constructeur afin d'initialiser les variables membres de la classe. Pour initialiser ces variables membres, nous avons d'ailleurs deux choix. La première est l'assignation telle que nous la connaissons :

```
1  //Source File
2  FooBar::FooBar(int parametre1, float parametre2)
3  {
4    m_parametre1 = parametre1; //Assumons que les types correspondent
5    m_parametre2 = parametre2;
6  }
```

La deuxième façon est d'utiliser la liste d'initialisation. Cette méthode possède plusieurs avantages, notamment elle permet d'initialiser les références ou variables constantes, ce qui est impossible avec la méthode précédente :

```
1  //Header file
2  class FooBar
3  {
4    public:
5      Foobar(int& parametre1, float parametre2);
6
7    private:
8      int& m_ref;
9      const float m_const;
10 };
11
12 //Source File
13 FooBar::FooBar(int& parametre1, float parametre2) : m_ref(parametre1), m_const(parametre2)
14 {
15 }
```

Tel qu'écrit, la liste d'initialisation commence par un **:**. Ensuite, chaque variable membre qu'on veut initialiser est écrite avec sa valeur entre parenthèses. Les variables sont séparées par une virgule.

5.4 Instanciation

Retournons maintenant à un niveau plus élevé ! Imaginez que vous avez fini par créer une classe toute belle et complète. Il est maintenant tant d'apprendre à l'utiliser dans nos programmes. Pour le besoin de la cause disons que nous avons la classe suivante :

```
1  //Header file
2  class Etudiant
3  {
4    public:
5      Etudiant(int p_energie, float p_sommeil) : m_energie(p_energie), m_sommeil(p_sommeil)
6      bool manger(int calories); //Avantage de retourner un bool: verifier les erreurs ←›
           survenant dans la methode
7      bool dormir(float heures);
8
9    private:
10     int m_energie;
11     float m_sommeil;
12 };
```

L'implémentation n'est pas importante pour notre exemple. **Notons par contre au passage deux choses IMPORTANTES** :

1. La déclaration de n'importe quelle fonction peut être faites dans le ficher header. On ne le fait habituellement pas parce que les implémentations sont généralement de plusieurs lignes et ce serait le bordel si on plaçait tout dans le fichier header. Par contre, cela est parfaitement acceptable pour des méthodes simples.
2. Plusieurs conventions de nommages existent pour les paramètres des fonctions, mais utiliser **p__** devant le nom d'un paramètre permet de clairement lister quels sont les paramètres dans l'implémentation d'une fonction.

Nous sommes maintenant de retour dans le main de notre programme, par exemple, nous pourrions créer un objet de type *Etudiant*.
NOTE : Il faut voir la classe comme étant un plan de maison et l'objet comme étant la maison dans le monde physique.

```
1  #include "Etudiant.h"
2
3  int main()
4  {
5      Etudiant monEtudiant(1000, 4.5);
6
7      Etudiant* monEtudiant2 =  new Etudiant(1000, 4.5);
8      delete monEtudiant2;
9
10     return 0;
11 }
```

Nous voyons dans l'exemple ci-dessus deux façon de créer notre objet. Tout d'abord, de la façon à laquelle nous sommes habitués, c'est-à-dire le type de la variable (ici *Etudiant*) suivit du nom de la variable. Le seul ajout, c'est qu'on vient ajouter entre parenthèses les valeurs des arguments qui seront passés au constructeur. Naturellement, s'il n'y en a pas, pas besoin de mettre de parenthèses.
La deuxième façon est par allocation dynamique. On crée d'abord un pointeur vers un objet de type *Etudiant* puis on lui assigne l'adresse d'une objet créé avec l'opérateur *new*. Notons ici que la parenthèse va à côté du **type** de la variable et non plus à côté du **nom** ! Il faut alors ne pas oublier de le supprimer avec l'opérateur *delete* afin de libérer la mémoire.

5.5 Constructeur de copie

Lorsque l'on fait un appel de fonction avec un objet en paramètre, l'objet est automatiquement copié en mémoire. En absence d'un **constructeur de copie**, le compilateur va automatiquement assumé qu'il doit simplement crée un nouvel objet dont chacune des variables contiennent exactement les mêmes valeurs. Pour des classes simples, ceci n'est pas un problème, mais si vous utiliser des pointeurs il est souvent préférable de définir son propre constructeur de copie pour éviter des désagrément.

```
1  //Header file
2  class FooBar
3  {
4    public:
5      Foobar(int& parametre1, float parametre2);
6      Foobar(Foobar&); //constructeur de copie
7
8    private:
9      int& m_ref;
10     const float m_const;
11 };
12
13 //Source File
14 FooBar::FooBar(int& parametre1, float parametre2) : m_ref(parametre1), m_const(parametre2)
15 {
16 }
17 FooBar::FooBar(Foobar& other) : m_ref(other.parametre1), m_const(other.parametre2)
18 {
19 }
```

Comme vous pouvez le voir, la définition du constructeur de copie n'est pas très complexe. Il suffit de définir un constructeur dont le seul paramètre est une référence vers une autre instance de la même classe.

5.6 Accesseurs et mutateurs

Les accesseurs et les mutateurs sont des méthodes dans une classe qui permettent d'accéder ou de modifier la valeur d'une variable d'un objet. En effet, il est préférable de ne pas permettre aux utilisateurs de la classe de modifier directement les variables pour éviter qu'ils y assignent des valeurs invalides.

Nous définissons les accesseurs et les mutateurs en ajoutant les mots-clefs Set et Get en avant du nom de la variable.

```
1  //Header file
2  class Etudiant
3  {
4    public:
5      Etudiant(int p_energie, float p_sommeil) : m_energie(p_energie), m_sommeil(p_sommeil)
6      bool manger(int calories); //Avantage de retourner un bool: verifier les erreurs ↩
           survenant dans la methode
7      bool dormir(float heures);
8      int getEnergie();
9      void setEnergie(int value);
10     float getSommeil();
11     void setSommeil(float value);
12
13   private:
14     int m_energie;
15     float m_sommeil;
16 };
17
18 //Source File
19 int Etudiant::GetEnergie()
20 {
21     return m_energie;
22 }
23 void Etudiant::SetEnergie(int value)
24 {
25     if(value < 0)
26         value = 0;
27     m_energie = value;
28 }
29 float Etudiant::GetSommeil()
30 {
31     return m_energie;
32 }
33 void Etudiant::SetSommeil(float value)
34 {
35     if(value > 1)
36         value = 1;
37     m_sommeil = value;
38 }
```

Les accesseurs et mutateurs sont donc des fonctions très simples, mais qui sont très utiles pour s'assurer que personne ne modifie les variables de vos objets en dehors des limites que vous avez défini.

6 Héritage

L'héritage est une façon très utilisée pour réutiliser du code et des créer des liens logiques entre des classes. Par exemple, si nous avons les classes moto et auto dans notre programme, nous pouvons certainement trouver des propriétés et des fonctions communes aux deux classes. Chacun possède des roues, roule à une certaine vitesse, a un numéro de modèle, etc. Il serait donc redondant d'ajouter ces propriétés aux deux classes. En programmation, nous n'aimons tellement pas la redondance que nous lui avons donné un nom : DRY ou Don't Repeat Yourself. C'est dans cet objectif qu'on va utiliser l'héritage. Dans notre exemple précédent, nous pourrions créer une classe véhicule dont hériterait (ou dériverait, c'est un synonyme) la classe moto et la classe auto.
NOTE IMPORTANTE : L'héritage est un outil puissant, mais il faut faire attention à l'utiliser de façon intelligente. En effet, les classes héritées (aussi nommées filles) sont fortement couplées (liées) à la classe de base (aussi nommée mère). Si vous ne pouvez pas dire *classe fille* EST UN(E) *classe mère*, alors l'héritage n'est PAS approprié pour la situation.

6.1 Structure

6.1.1 Déclaration

Pour faire un héritage, il faut d'abord une classe de base. Reprenons notre exemple précédent (en restant simple pour les besoins de la cause) :

```
1  class Vehicule
2  {
3    public:
4      Vehicule(int p_nombreRoues, float p_vitesseMax, std::string p_modele)
5
6      std::string modele();
7      bool avance(float temps);
8
9    protected:
10     std::string m_modele;
11     int m_nombreRoues;
12     float m_vitesseMax;
13     float m_vitesse;
14 }
```

Maintenant, imaginons que nous voulons créer les classes filles auto et moto :

```
class Moto : public Vehicule
{
  public:
    Moto(float p_vitesseMax, std::string p_modele)

    void accelererEnfou();
}

class Auto : public Vehicule
{
  public:
    Auto(float p_vitesseMax, std::string p_modele, int p_portes)

    int portes();

  private:
    int m_portes;
}
```

La première chose qu'il faut remarquer ici, c'est la partie qui vient après le nom de la classe. Pour marquer qu'on désire faire hériter notre classe d'une autre classe, on commence par mettre un **:** et on met ensuite la portée de l'héritage (Voir le tableau 1 dans la section 6.1.3 pour autres types) suivi du nom de la classe de base. Notons que le C++ permet d'hériter de plusieurs classes (en ajoutant une virgule et en ajoutant le combo *Portée NomClasseMère*), mais nous ne traiterons pas le sujet ici.
La deuxième chose qu'il faut observer est l'utilisation de la portée *Protected*. Si vous avez besoin d'un petit rafraîchissement, voir la section 5.1.2.

Vous vous demandez peut-être pourquoi on a décidé de mettre, par exemple, le paramètres *m_portes* seulement dans la classe *Auto* au lieu de la classe *Vehicule*. En fait, toutes les propriétés de la classe mère sont héritées par les classes filles. Ainsi, nous aurions la classe moto qui aurait une propriété *m_portes* ce qui ne fait aucun sens !

6.1.2 Constructeur

L'implémentation du constructeur de la classe fille comporte une seule particularité, il faut appeler le constructeur de la classe mère dans notre liste d'initialisation et lui passer les paramètres nécessaires :

```
Auto::Auto(float p_vitesseMax, std::string p_modele, int p_portes) :
      Vehicule(4, p_vitesseMax, p_modele), m_portes(p_portes)
{
}

Moto::Moto(float p_vitesseMax, std::string p_modele) : Vehicule(2, p_vitesseMax, p_modele)
{
}
```

6.1.3 Accès à la classe mère

Ci-dessous vous pouvez voir un tableau permettant de voir le type de protection final d'un membre (variable ou fonction) d'une classe mère selon la portée de l'héritage utilisée. De façon générale, on utilise l'héritage *Public*. Ainsi, les agents extérieurs peuvent accéder aux méthodes publiques de la classe mère et la classe fille peut accéder aux méthodes et variables protégées de la classe mère.

Protection des membres dans la classe de base	Portée de l'héritage		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Private	Private	Private

TABLE 1: Accès aux membres de la classe mère selon le type d'héritage

6.1.4 Masquage

Le masquage d'une fonction de la classe mère survient lors d'une redéfinition de cette fonction par la classe fille. Autrement dit, si la classe fille possède une méthode ayant exactement le même nom, paramètres et type de retour que celle de la classe mère, la méthode qui sera appelée sera celle de la classe fille. Ainsi, on dit que la méthode de la classe mère est masquée. Par exemple, avec notre exemple précédent de véhicules :

```
1  //Header
2  class Vehicule
3  {
4    public:
5      std::string modele() { return m_modele; }
6      (...)
7  }
8
9  class Moto : public Vehicule
10 {
11   public:
12     std::string modele() { return m_modele + " sport"; }
13     (...)
14 }
15
16 class Auto : public Vehicule
17 {
18   public:
19     std::string modele() { return m_modele + " de luxe"; }
20     (...)
21 }
22
23 //Main
24 Auto monAuto(200, "Audi", 4);
25 Moto maMoto(400, "BMW");
26
27 cout << monAuto.modele() << endl
28      << maMoto.modele() << endl;
```

Tel qu'attendu, la console nous indique *Audi de luxe* et *BMW sport*. On voit donc clairement que la fonction utilisée n'est pas celle de la classe mère, mais bien celle de la classe fille. Par contre, on peut appeler la fonction mère si on le désire dans la fonction fille :

```
1  //Header
2  class Vehicule
3  {
4    public:
5      std::string modele() { return m_modele; }
6      (...)
7  }
8
9  class Moto : public Vehicule
10 {
11   public:
12     std::string modele() { return Vehicule::modele() + " sport"; }
13     (...)
14 }
```

Remarquez l'appel à la fonction *Vehicule : :modele()*. Comme on spécifie explicitement que l'on veut la fonction mère, le compilateur se fait une joie de l'exécuter pour nous !

6.2 Polymorphisme

6.2.1 Principe

Continuons avec l'exemple des véhicules que nous avons développé dans les sections précédentes pour exposer l'utilité du polymorphisme. Considerez le code suivant :

```
1  //Main cpp
2  void afficherModele(Vehicule monVehicule)
3  {
4    std::cout << monVehicule.modele();
5  }
6
7  int main()
8  {
9    Auto monAuto(200, "Audi", 4);
10   Moto maMoto(400, "BMW");
11
12   afficherModele(monAuto);
13   afficherModele(maMoto);
14
15   return 0;
16 }
```

L'héritage nous permet d'écrire le code précédent qui compilera sans problème. En effet, on peut toujours utiliser une classe fille (*Auto* et *Moto* ici) là où on attend une classe mère (*Vehicule* ici). Notre problème se trouve plutôt au niveau de la sortie vers la console. On s'attendrait en effet à obtenir *Audi de luxe* et *BMW sport*. Pourtant, la console nous indique seulement *Audi* et *BMW*. En effet, le compilateur ne sait pas qu'il a affaire à la classe Auto et à la classe Moto et appele donc la fonction de la classe mère... Heureusement, nous avons le polymorphisme !

6.2.2 Mise en œuvre

Pour pouvoir utiliser la puissance du polymorphisme, il faut d'abord respecter la syntaxe. Revenons à la planche à dessin dans la définition des classes *Auto*, *Moto* et *Vehicule* :

```cpp
//Header
class Vehicule
{
  public:
    virtual std::string modele() { return m_modele; }
    (...)
}

class Moto : public Vehicule
{
  public:
    virtual std::string modele() { return Vehicule::modele() + " sport"; }
    (...)
}

class Auto : public Vehicule
{
  public:
    virtual std::string modele() { return m_modele + " de luxe"; }
    (...)
}
```

La simple et unique modification que nous avons apportée est l'ajout du mot-clé virtual devant la fonction mère et devant les fonctions filles. Cela nous permettra d'utiliser le polymorphisme dans notre programme. Notons que le polymorphisme ne fonctionne qu'avec des références et des pointeurs. Ainsi la fonction *afficherModele(Vehicule monVehicule)* ne fonctionnerait pas plus. Il faudrait plutôt l'écrire d'une des deux façons suivantes :

```cpp
//Main cpp
void afficherModele1(Vehicule* monVehicule)
{
  std::cout << monVehicule->modele();
}

void afficherModele2(Vehicule &monVehicule)
{
  std::cout << monVehicule.modele();
}

int main()
{
  Auto monAuto(200, "Audi", 4);
  Moto maMoto(400, "BMW");

  afficherModele1(&monAuto);
  afficherModele1(&maMoto);

  afficherModele2(monAuto);
  afficherModele2(maMoto);

  return 0;
}
```

Dans les deux cas, la console nous montre *Audi de luxe* et *BMW sport* !!! Maintenant, vous vous demandez peut-être à quoi cela sert-il vraiment. En fait, la force du polymorphisme est que nous pouvons créer et utiliser des interfaces (comme des fonctions) communes peu importe le type dérivé que l'on passe en paramètre.