Méthodes informatiques pour physiciens

introduction à C++ et
résolution de problèmes de physique par ordinateur

Leçon # 9 : Les classes

Alessandro Bravar Alessandro.Bravar@unige.ch
tél.: 96210 bureau: EP 206

assistants

Johanna Gramling Johanna.Gramling@unige.ch
tél.: 96368 bureau: EP 202A

Mark Rayner Mark.Rayner@unige.ch
tél.: 96263 bureau: EP 219

http://dpnc.unige.ch/~bravar/C++2015/L9
pour les notes du cours, les exemples, les corrigés, …

Plan du jour #9

Récapitulatif et corrigé de la leçon #8

Programmation orientée objets – idées de base

texte Micheloud et Rieder
chap. 16 et 17

Introduction aux classes

Les classes

Données membres

Constructeurs

La surcharge des operateurs

Programmation orientée objets

C++ = C + typage fort + classes

Ce que nous avons étudié jusqu' ici (programmation structurée) se retrouve dans la plupart des langages de programmation, notamment le C et Java.

La notion de classe ajoute au C++ les caractéristiques objets.

La notion d'objet est basée sur l'idée de ne pas séparer les données et les fonctions agissant sur les données, mais de traiter ces données et ces fonctions de la même manière.

Une classe est un nouveau type de données qui contient les données et les méthodes (fonctions et operateurs) permettant de travailler sur ces données.

La programmation orientée objet (POO) implique donc des programmes qui utilisent des classes.

Programmation orientée objets

Dans la programmation structurée,
les fonctions ont un rôle privilégié

Avec la POO et les classes,
les données et les fonctions sont traitées de la même façon.

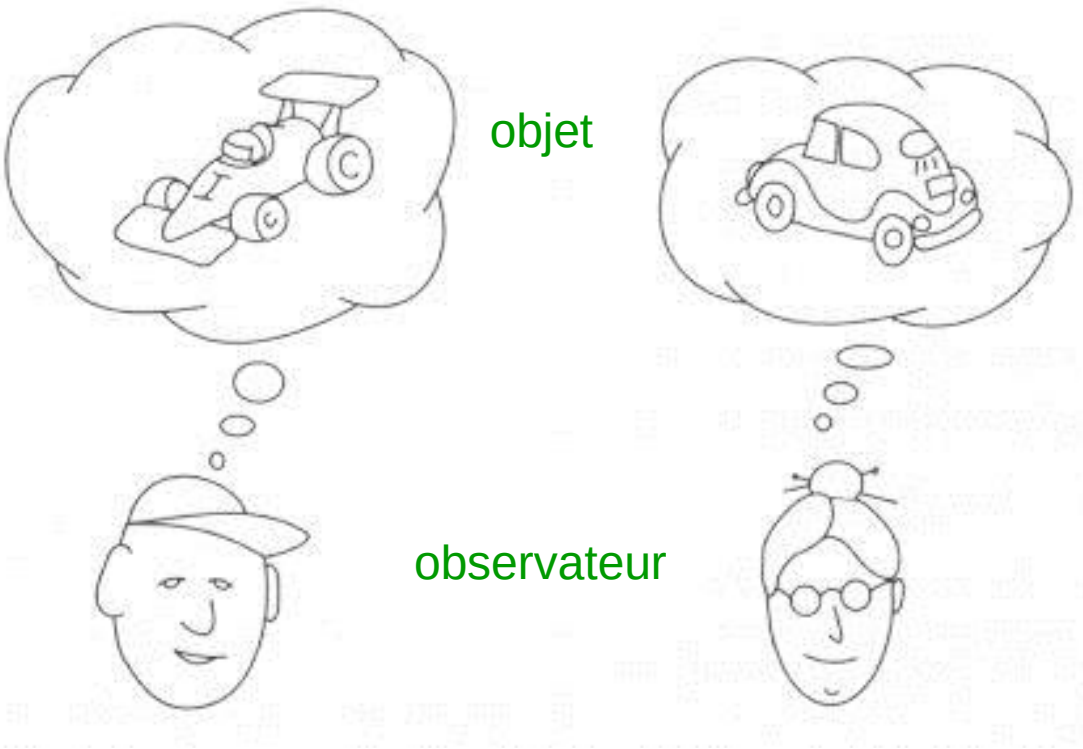
Nous nous focalisons sur les données et ce que nous pouvons faire (fonctions) avec ces données.

La POO apporte une amélioration à la fiabilité du code et une réutilisation possible des éléments d'un développement.

C'est très important dans le développement de logiciels complexes (codes très longs) avec plusieurs programmeurs qui travaillent sur le développement du même programme :

- fiabilité (validité et robustesse)
- extensibilité
- réutilisabilité
- compatibilité

Abstraction

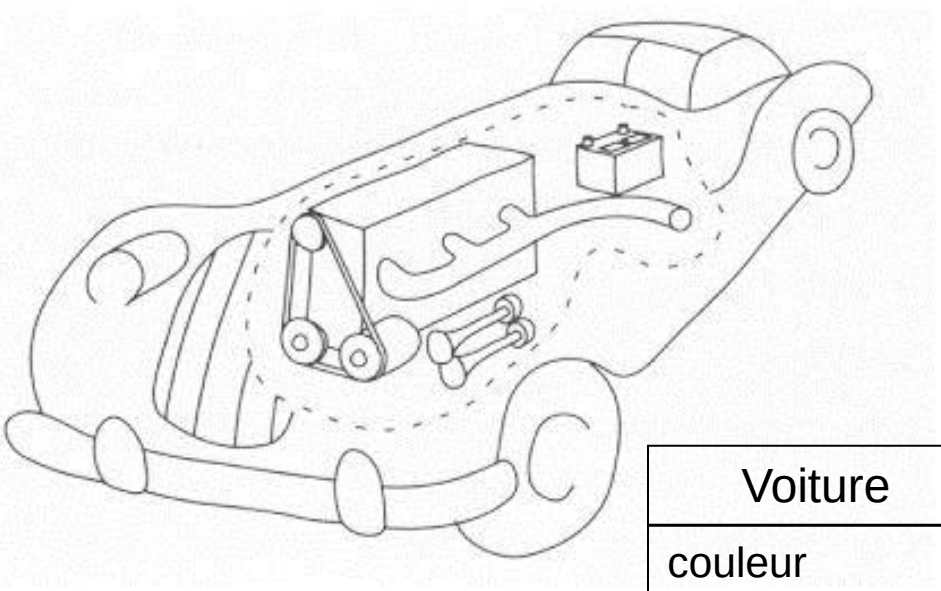


L'abstraction se concentre sur les caractéristiques importantes d'un objet selon le point de vue de l'observateur.

Toutes les voitures sont similaires: i.e. elles ont 4 roues, 1 moteur, elles peuvent rouler, etc.

Elles diffèrent par le model, la couleur, etc.

Encapsulation



Voiture
couleur marque modèle cylindré
démarrer changerVitesse freiner

attributs →

méthodes →

Par l'intermédiaire de l'encapsulation, un objet peut être utilisé sans connaître son fonctionnement interne (masquage de l'information).

Par ex. pour conduire une voiture on n'a pas besoin de connaitre le fonctionnement du moteur.

Cette modélisation nous amène à construire une classe `Voiture` qui contiendra ces différentes définitions

Même chose pour les logiciels: on n'a pas besoin de connaître les informations contenues dans les objets (classes) et leur fonctionnement interne.
Mais pour utiliser une classe quelqu'un a dû écrire cette partie du logiciel.

Ces notions peuvent apparaître très abstraits, mais avec des exemples la notion de une classe, comment la construire et comment l'utiliser deviendra plus clair.

Programmation orientée objets

La POO est basée sur quatre concepts majeurs :

- l'abstraction de données
  caractéristiques importantes d'un objet

- l'encapsulation
  masquage de l'information

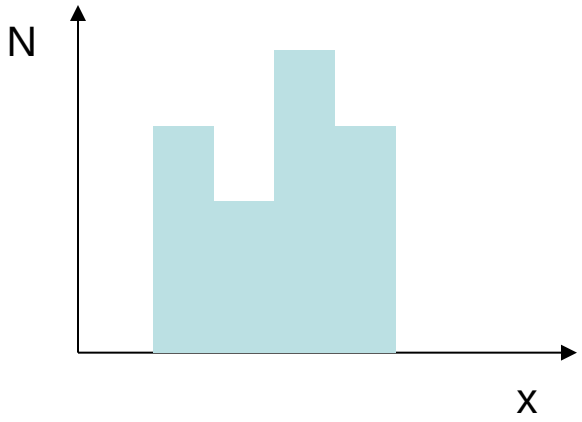
- l'héritage ou dérivation
  création de sous-classes à partir d'une autre classe (hiérarchie d'abstraction)

- le polymorphisme
  construction d'une classe descendante qui remplace des classes ancêtres,
  e.g. rectangles, triangles, … → polygones

Objets et abstraction

L'abstraction se concentre sur les caractéristiques importantes d'un objet selon le point de vue de l'observateur.

Considérons p.ex. un histogramme:



caractéristiques de l'histogramme:

- x_{min}
- x_{max}
- nombres des canaux
- . . .

méthodes:

- valeur moyenne
- écart quadratique moyenne
- . . .
- affichage de l'histogramme

Pour présenter un ensemble de mesures en forme d'histogramme je dois spécifier la valeur x_{min} et x_{max} des mes données, et le nombre de canaux de l'histogramme. En plus, je veux connaître la valeur moyenne de mes mesures et afficher l'histo. Pour un autre ensemble de mesures je obtiendrai un autre histogramme.

Voir p.ex. http://root.cern.ch , un de logiciel d'analyse de données basé sur un ensemble de classes développées au CERN.

Classes

Une classe est un type dérivé. Les éléments d'une classe peuvent être des fonctions, et notamment des opérateurs. Les classes sont la base de la programmation orientée objet. En fait, un objet est une entité autonome qui stocke ses propres données et possède ses fonctions. Avec celles-ci on va manipuler des données et des fonctions qui vont apporter des fonctionnalités au programme.

La déclaration commence par le mot-clé **`class`**, suivi du nom de la classe, et se termine par un point-virgule, comme dans l'exemple suivant:

variables

fonctions

```
class Rational {
  public:
    int num, den;
    double convert();
    void invert();
    void print();
};
```

par défaut tous les membres sont `private` (privés)

(prototypes des fonctions)

Les variables sont appelées *données membres* et les fonctions `convert`, … *méthodes*. Dans cette classe toutes les variables et méthodes sont désignées comme `public`. Les membres `public` sont accessibles depuis l'extérieur de la classe, les membres `private` le sont uniquement depuis l'intérieur de la classe. Les données membres sont toujours accessibles aux méthodes: il n'est pas nécessaire de les passer aux fonctions.

Données membres

Une classe contient des données, appelées données membres et des traitements appelés traitements membres (ou méthodes).

Les données membres sont des variables qui sont rangées à l'intérieur d'une classe; elles doivent être précédées de leur type et posséder un nom. On regroupe ces variables selon l'accès que le programme aura vis-à-vis de cette donnée (qualificatifs d'accès):

- **`private`**: accès aux membres seulement depuis l'intérieur de la classe; le masquage des données permet d'empêcher l'accès depuis l'extérieur (et donc de modifier ces données). Par défaut tous les membres sont `private` (privés).

- **`public`**: accès aux membres depuis l'extérieur de la classe

- **`protected`**: accès aux membres seulement aux *classes dérivées* ou *amies* (**`friend`**)

Comme exemple de classe nous construirons un nouveau type de variable, les nombres rationnels. Il s'agit d'un couple de nombres entiers ordonnés. Ils ne sont pas définis dans le C++ parce qu'on peut faire tous les calculs avec les entiers et les virgules flottantes.

Nous définirons:
- le couple de nombres entiers
- des fonctions pour manipuler ces nombres (ce couple)
- des operateurs pour effectuer des calculs typiques (addition, relations, …)

```
#fichiers en tête

class Rational {
  public:
    int num, den;
    double convert();
    void invert();
    void print();
};

double Rational::convert() {
  return double(num) / double(den); }
void Rational::invert() {   // 1/q
  int temp = num; num = den; den = temp; }
void Rational::print() {
  cout << num << ''/'' << den; }

int main() {
  Rational q;
  q.num = 22; q.den = 7;
  cout << '' q ='' ; q.print();
  cout << '' = '' << q.convert();
  q.invert();
  cout << ''1/q ='' ; q.print();
  return 0; }
```

variables

(prototypes des) fonctions

définition des fonctions de la classe `Rational`;

l'opérateur de résolution de portée ::
permet de relier la définition de fonction à la classe `Rational`

Les données membres sont toujours accessibles aux méthodes: il n'est pas nécessaire de les passer aux fonctions.

`q` est déclaré comme un type `Rational`
(objet de la class `Rational`)

initialisation

l'accès aux membres d'une classe (données ou fonctions)
se fait au moyen de l'opérateur de sélection de membre direct . (point)

voir Rational1.cpp

Du point de vue de la sécurité il serait mieux de restreindre l'accès aux données et déclarer toutes les données d'une classe comme privées (`private`), c'est-à-dire cacher les données.
Une fois initialisées elles ne peuvent être modifiées que par des fonctions d'accès.
Ce mécanisme, qui empêche l'accès direct à des membres privés, s'appelle l' encapsulation. Pour initialiser l'objet, on a besoin des fonctions d'accès déclarées `public`.

Dans l'exemple précèdent :

voir Rational2.cpp

```
class Rational {
  public:
     .....
     void assign (int n, int d);
private:
     int num, den;
};
Rational::assign(int n, int d) {
  num = n;    den = d; }
.....
// dans main
Rational q;
q.assign(22 , 7);
```

ici on a construit la fonction `assign`
pour initialiser l'élément `q` de la classe `Rational`

Dans cet exemple toutes les méthodes (fonctions) sont désignées comme `public` et toutes les données membres comme `private`. Les membres publics sont accessibles depuis l'extérieur de la classe tandis que les membres `private` le sont uniquement depuis l'intérieur de la classe, c'est à dire par les méthodes de la classe.

Constructeurs

Pour initialiser les objets au moment de la déclaration on utilise des constructeurs.
Un constructeur est une méthode appelée automatiquement dès qu'un objet est déclaré.
Le rôle des constructeurs est d' allouer la mémoire nécessaire pour le stockage des objets et leur initialisation. Le constructeur porte le même non que la classe et est déclaré sans type de renvoi.

voir Rational3.cpp

```
class Rational {
  public:
    Rational(int n, int d);
  private:
    int num, den;
};
Rational::Rational(int n, int d) {
 num = n; den = d; }
.....
// dans main
Rational q(22,7);  // même que int i = 5; !
```

après la déclaration, `q` ne peut plus être modifié parce qu' il est déclaré `private` dans `Rational` sauf si on utilise des fonctions d'accès (méthodes de la classe)

Une classe peut avoir plusieurs constructeurs selon l'application. Comme pour les fonctions surchargées ils se distinguent par leurs différentes listes de paramètres :

```
Rational() {num = 0; den = 1;}
Rational(int n) {num = n; den = 1;}
Rational(int n, int d) {num = n; den = d;}
```

Les données membres d'une classe sont généralement déclarées comme `private`, et ses méthodes comme `public`. On peut aussi déclarer des méthodes comme `private`, comme dans cet exemple; ces méthodes ne peuvent être utilisées que dans la classe et sont qualifiées de `fonctions d'utilité` locales.

Ce mécanisme, qui empêche l'accès direct à des membres privés, s'appelle l'encapsulation.

on peut définir un constructeur à l'aide des listes d'initialisation

```
class Rational {
public:
  Rational(int n=0, int d=1) : num(n), den(d) {reduce();};
  void print() {cout << num << "/" << den << endl;}
private:
  int num, den;
  int dcg(int, int);
  void reduce() {int g = dcg(num,den); num/=g; den/=g;}
};

int Rational::dcg(int m, int n)
//renvoie le diviseur commun le plus grand de m et d n
  {if(m<n) swap(m,n);
   while (n>0) {int r=m%n; m = n; n = r;}}

int main() {
  Rational q(100, 360);
  q.print();
}
```

ici on appelle la fonction `reduce` dans le constructeur après l'initialisation de `num` et `den` (algorithme d'Euclide pour trouver le DCG

les fonctions `reduce` et `qgd` ne peuvent pas être appelées dans main

résultat: 5 / 18

voir Rational4.cpp

Fonctions d'accès

L'accès aux données membres d'une classe se fait

1. au moyen de l'opérateur de sélection de membre direct . (point) si les données sont `public`

```
class Rational {
  public:
    int num, den;
    . . . . };
```

```
int main() {
  Rational q(22, 7);
  cout << q.num;
  . . . . }
```

2. Avec une fonction d'accès si les données sont `private`

```
class Rational {
  public:
    int numerator() const {return num;}
  private:
    int num, den;
  . . . . };
```

```
int main() {
  Rational q(22, 7);
  cout << q.numerator();
  . . . . }
```

On utilise l'operateur de sélection de membre direct . aussi pour accéder aux méthodes de la classe (fonctions membres)

Opérateurs de surcharge

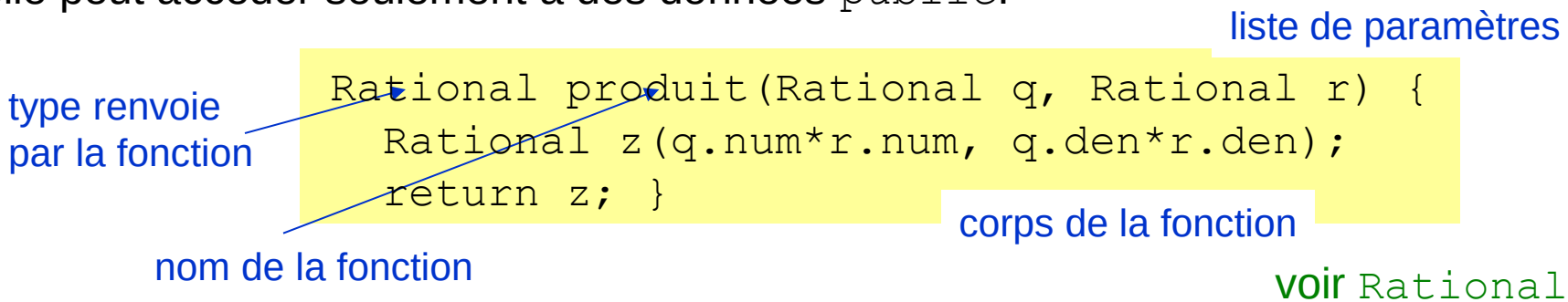
```
class Rational {
  public:
    Rational(int n=0, int d=1) : num(n), den(d) { };
    void print() {cout << num << " / " << den << endl;}
  private:
    int num, den;
};

int main() {
  Rational q(100, 360);
  Rational r;
  r = q;      ???     Est-ce qu' on peut écrire ça?
  Rational s(10, 36), t(15, 29);
  r = s + t;      ???       Est-ce qu' on peut faire ça?
  return 0;
}
```

On voudrait faire les mêmes opérations arithmétiques et logiques aussi avec les nombres rationnels: `+`, `-`, `*`, `:`, `<`, `>`, `==`, `&&`, etc.

Vous vous souvenez de la surcharge des fonctions ?
Même nom pour la fonction avec une liste de paramètres différents.

Le C++ inclut de nombreux opérateurs définis automatiquement pour les types fondamentaux (`int`, `double`, etc.). Lorsque vous créez un nouveau type (une classe) vous pouvez surcharger la plupart des opérateurs C++ pour les utiliser de la même façon qu'avec les types fondamentaux.

Pour multiplier q * r on peut définir une fonction `produit`.
Attention: la fonction produit n'est pas une fonction membre de la class `Rational`: elle peut accéder seulement à des données `public`.



voir `Rational5.cpp`

(pour accéder a données membres privées, on a besoind des fonctions amies (`friend`))
Alternativement on peut aussi surcharger l'opérateur * utilisant le mot clés `operator`
(l' opérateur doit être déclare comme méthode publique de la classe `Rational`):

```
Rational Rational::operator*(Rational & q) {
  Rational z(num*q.num, den*r.den);
  return z; }
```

voir `Rational6.cpp`

et pour les operateurs logiques :

```
bool Rational::operator==(Rational x, Rational y) {
  return (x.num*y.den == y.num*x.den); }
```

Multiplication par une fonction

```
#include <iostream>

using namespace std;

class Rational {
  public:
    Rational (int n = 0, int d = 1): num(n), den(d) {};
    int num, den;
    void print();
};

void Rational::print() {
  cout << num << '/' << den  << endl;}

Rational produit(Rational x, Rational y) {
  Rational z (x.num*y.num, x.den*y.den);
  return z; }

int main() {
    Rational q(11, 3), r(5,7);
    cout << "q = "; q.print();
    cout << "r = "; r.print();
    Rational s = produit(q,r);
    cout << "q * r = "; s.print();
    return 0;
}
```

définition de la fonction produit

multiplication par appel de la fonction produit

voir Rational5.cpp

Surcharge de l'opérateur de multiplication

```
#include <iostream>

using namespace std;

class Rational {
  public:
    Rational (int n = 0, int d = 1): num(n), den(d) {};
    //operateur surcharge
    Rational operator*(const Rational &) const;
  private:
    int num, den;
};

//operateur surcharge
Rational Rational::operator*(Rational &r) const {
  Rational q (num*r.num, den*r.den);
  return q;
}

int main() {
    Rational q(11, 3), r(5,7);
    cout << "q = ";   q.print();
    cout << "r = ";   r.print();
    Rational s = q * r;
    cout << "q * r = "; s.print();
    return 0;
}
```

surcharge de l'opérateur *

multiplication par l'operateur * surchargé

voir Rational6.cpp

Un exemple pratique

voir Spring.cpp

Résolution des équations différentielles avec des classes:

classe `Ressort` - données de ressort

- calcul de la trajectoire et dessin de la trajectoire

définition de la classe `Ressort`

```
class Ressort {
  public:
    Ressort();   //constructeur par defaut
    Ressort(float, float, float, float);   //const. avec param.
    ~Ressort();   //destructeur

    void definition();         //saisie
    void deplacer(float, float);          //mise a jour
    void periode();     //periode d'oscillation

  private:
    float young;
    float masse;
    float vitesse;
    float distance;
    float ini_vit;
    float ini_dist;

    void trace();  //cette fonction memorise les cond. initiales
};
```

constructeurs

destructeur

fonctions

données

fonction

constructeurs et destructeur

```
// constructeur par defaut
Ressort::Ressort() {
  vitesse = 0.;
  distance = 0.;
  trace();
  cout << "Ressort construit sans initialization !" << endl;
}

//constructeur avec parametres
Ressort::Ressort(float y, float m, float vit, float dist) :
  young(y), masse(m), vitesse(vit), distance(dist) {
  trace();
  cout << "Periode d'oscillation: " << young/masse << endl;
}

//destructeur
Ressort::~Ressort() {
  cout << "Le ressort a ete detruite !" << endl;
}
```

fonction membre
de la classe `Ressort`

```
//memorise les conditions initiales
void Ressort::trace() {
  ini_vit = vitesse;
  ini_dist = distance;
}
```

```
//saisie des donnees
void Ressort::definition() {
  cout << "Module de Young  : "; cin >> young;
  cout << "Masse            : "; cin >> masse;
  cout << "Position initiale: "; cin >> distance;
  cout << "Vitesse initiale : "; cin >> vitesse;
}

//periode d'oscillation
void Ressort::periode() {
  cout << "periode d'oscillation: " << young / masse << endl;
}
//calcul et dessin de la trajectorie
void Ressort::deplacer(float t, float dt) {
  trace();
  int steps = int(t/dt);
//allocation dynamique des vecteurs iX, iV, iT pour les fonctions dislin
  float *iX = new float[steps];
  float *iV = new float[steps];
  float *iT = new float[steps];
  iV[0] = ini_vit; iX[0] = ini_dist; iT[0] = 0.;
  for (int i = 1; i < steps; i++) {
    iX[i] = distance + vitesse * dt;
    iV[i] = vitesse - young / masse * distance * dt;
    iT[i] = dt * i; distance = iX[i]; vitesse = iV[i]; }
  metafl(« XWIN");
  qplot(iT, iV, steps);     //dislin
  delete [] iX, iV, iT; }
```

fonctions membres de la classe `Ressort`

méthode d'Euler

fonction main

```
int main() {
  Ressort R1(1.0, 1.0, 0.0, 1.0);   //Ressort 1 avec initialization
  R1.periode();
//dessin de la trajectoire
  R1.deplacer(10., 0.01);   //t et delta t

  Ressort R2;                       //Ressort 2 sans initialization
  R2.definition();                  //saisie de donnees
  R2.periode();
//dessin de la trajectoire
  R2.deplacer(100., 0.01);   //t et delta t

  return 0;
}
```

Directives de prétraitement

Le compilateur C/C++ contient un préprocesseur capable d'inclusion de fichiers, de compilation conditionnelle et de substitution de macros. Le directive de prétraitement commence par le symbole #. Ce sont des outils permettant d'avoir une meilleure vue d'ensemble du programme ou de grands projets.
Il ne s'agit pas des programmation C++, mais de manipulation de texte.

```
#include <nom du fichier>
```

Permet d'inclure le contenu entier du fichier spécifié; on utilise cette directive avec le fichier en-têtes (header file) contenant des déclaration de classes, fonctions, etc.

```
#define identificateur symbole
```

Cette directive provoque le remplacement par le préprocesseur de toutes les occurrences suivantes de l'`identificateur` par la séquence `symbole`.

Ces constantes sont globales par rapport au fichier.

```
#define LONGEUR 80
var = LONGEUR * 20 → var = 80 * 20
```

Il est conseillé d'utiliser `const` au lieu de `#define` pour déclarer les constantes du programme, car les noms ne sont pas typés et ne suivent pas les règles de portée.

```
#if expressionConstante
  . . .
#endif
```

ou

```
#if expression Constante1
  . . .
#elif expressionConstante2
  . . .
#else
  . . .
#endif
```

Si l'`expressionConstante` est vrai (donc différente de 0) les instructions ou directives placées dans cette structure seront compilés.

```
#ifdef identificateur
  . . .
#endif
```

ou

```
#ifndef idenficateur
   . . .
#endif
```

Si l'identificateur est vrai, il a bien été défini avec `#define` (cas `#ifdef`) ou n'a pas été défini (cas `#ifndef`) les instructions ou directives seront compilés.

En général toutes le fichers en-tête sont construits sur ce modèle pour éviter plusieurs inclusions d'un même fichier.

Décomposition en fichier

Un programme peut être composé d'un ou de plusieurs fichiers. Dès que la taille d'un programme augmente, il peut être utile de le décomposer en plusieurs parties (fichiers). Ceci permet une manipulation du programme plus aisée. Chaque fichier est constitué d'une suite de déclarations et d'instructions et peut être compilé séparément. Les fichiers compilés sont ensuite assemblés par l'éditeur de liens (linker) pour obtenir un programme exécutable. Cette méthode nécessite une structuration rigoureuse des fichiers qui composent un programme ainsi que l'utilisation des types de variables.

Dans la pratique, on sépare la déclaration d'une classe et sa définition (implémentation des fonctions) en deux fichiers. Même pour les fonctions : déclaration ou prototype de la fonction et définition de la fonction.

Ainsi pour chaque classe on créera deux fichiers :

- un fichier en-tête contenant la déclaration de la classe :
  toutes les données membres et les prototypes des fonctions membres (méthodes)
  le fichier en-tête est inclus dans le programme par l'instruction `#include`

- un fichier contenant les définitions (corps) des fonctions membres (méthodes)
  en général, on reçoit le fichier objet résultant de la compilation séparé de ce fichier de définition de la classe

Voir Leçon #7 - projet

Fichier contenant la déclaration de la classe `Ressort.h`

voir spring, cercle, …

```
#ifndef RESSORT_H
#define RESSORT_H

classe Ressort {
  données membre
  méthodes
};

#endif
```

cela permet de contrôler et éviter plusieurs inclusions d'une même fichier

Fichier de définition de la classe contenant les définitions de les méthodes (corps de les fonctions de la classe) `Ressort.cpp`

```
#include "Ressort.h"

Ressort::Ressort() {   // constructeur par défaut
}
autres méthodes
```

Fichier d'utilisation de la classe (programme principal) `main.cpp`

```
#include Ressort.h

int main() {
  . . .
}
```

La Bibliothèque STL de C++

La Standard Template Library (STL) fait partie de la bibliothèque standard ANSI C++.
Il s'agit d'une collection de types de données et d'algorithmes offrant des solutions pour
Une variété de problèmes. Le recours à des bibliothèque de classes facilite et rand
plus efficace le développement d'applications.

La STL est basée sur un concept appelé « programmation générique » qui est
implémenté avec l'utilisations de patrons (templates) ou modèles des classes.
Le compilateur utilise les modèles afin de générer le code de différentes fonctions
ou classes selon le type des objets utilisés.

La STL contienne des modèles des classes fournissant

les conteneurs
des objet contenants d'autres objets, comme `<vector>`

les itérateurs
des « fonctions » pour accéder au éléments d'un conteneur

les algorithmes
qui permettent de manipuler les données à l'intérieurs des conteneurs

les nombres complexes

Un exemple

```
#include <iostream>

#include <complex>

using namespace std;

int main() {

  complex<double> z1(1., 0.);
  complex<double> z2(3., 4.);
  cout << z1 << endl;
  cout << z1 + z2 << endl;

  return 0;
}
```

voir Complex.cpp

nome du patron
(modèle de la classe)

type donnée

Mots-clés du langage C++

asm	do	if	return	try
auto	double	inline	short	typedef
bool	dynamic_cast	int	signed	typeid
break	else	long	sizeof	typename
case	enum	mutable	static	union
catch	explicit	namespace	static_cast	unsigned
char	export	new	struct	using
class	extern	operator	switch	virtual
const	false	private	template	void
const_cast	float	protected	this	volatile
continue	for	public	throw	wchar_t
default	friend	register	true	while
delete	goto	reinterpret_cast		

mots-clés déjà vus

mots-clés rencontrés aujourd'hui

Résumé

Dans la programmation structurée, les fonctions ont un rôle privilégié ; avec la PPO et les classes, les données et les fonctions sont traitées de la même façon. Nous nous focalisons sur les données et ce que nous pouvons faire (fonctions) avec ces données.

Construction d'une classe

```
class Nomclasse {
  public:
    type1 element1;
    type2 element2;
    fonct1();
    fonct2();
       . . .
  private:
    typea elementa;
    typeb elementb;
    fonct a();
    fonct b();
       . . .
};
```

Déclaration d'un objet (dans `main`)

```
Nomclasse Objet1;
Nomclasse *Objet2 = new Nomclasse;
```

Constructeurs et Destructeurs

Pour initialiser les objets au moment de la déclaration on utilise des fonctions particulières appelées constructeurs:

```
class Nomclasse {
  public:
    Nomclasse(type1 x, ...);
};
```

constructeur

```
Nomclasse::Nomclasse(type1 x, …) {
  element1 = x;
  …
}
```

corps de la "fonction" constructeur

ou

```
Nomclasse::Nomclasse(type1 x, …) : element1(x), … {
}
```

liste des paramètres

(si le constructeur n'est pas défini, on utilise le constructeur par default)

Pour libérer de l'espace mémoire après l'utilisation d'un objet (détruire l'objet), on utilise une fonction appelée destructeur.

Méthodes (fonctions membres de la classe)

En général la méthode est déclarée dans la classe (prototype) et définie de hors de la classe.

```
class Nomclasse {
  public:
    type nomfonction( ... );
};

type Nomclasse::nomfonction( ... ) {
  ...
  return ...;
}
```

prototype de la méthode

corps de la fonction

opérateur de résolution de portée

Fonctions d'accès

L'accès directe aux données membres de type `private` est interdit.
Pour accéder aux données membres de type `private` (lecture / écriture) on utilise des méthodes (fonctions membre) de type `public` de la classe.

```
class Rational {
  public:
    int numerator() const {return num;}
  private:
   int num, den;
};
```

Accès au membre d'une classe

Pour accéder aux membres d'une classe (données et méthodes) on utilise l'opérateur de sélection de membre direct . (point)

```
objet1.element1;
```
(donné)

```
objet1.fonction1();
```
(méthode)

et l'opérateur de sélection de membre indirect -> si déclare par pointeur

```
objet1->element1;
```
(donné)

```
objet1->fonction1();
```
(méthode)

Surcharge des opérateurs

Les opérateurs (arithmétique, logique) sont définis pour les types fondamentaux. La surcharge consiste à ajouter à un certain opérateur des fonctionnalités qui nous permettent d'utiliser aussi ces opérateur avec des types dérivés (e.g. un classe).

p.ex. surcharge de l'operateur de multiplication *

```
Rational operator*(const Rational &) const;

Rational::Rational operator*(const Rational &q) const {
  return Rational(num*q.num, den*q.den);
}
```

Exercices – série 9

Exercices

1. Implémentez une classe `Point` pour des points tridimensionnels `(x, y, z)`. Déclarez `(x, y, z)` comme `private`. Intégrez un constructeur par défaut, une fonction `norm()` qui renvoie la distance de son origine `(0, 0, 0)`, une fonction `negate()` pour rendre le point négatif, et une fonction `print()`.
Développez une fonction pour additionner deux vecteurs définis par les points définis ci-dessus.

2. Implémentez une class `Circle`. Chaque objet de cette classe représentera un cercle et stockera son rayon et les coordonnées `(x, y)` de son centre. Intégrez un constructeur, des fonctions d'accès, une fonction `area()` et une fonction `circumference()`.

3. La même technique peut être utilisée pour surcharger tous les opérateurs. Etudiez en détail le programme `RationalX.cpp` et complétez la class Rational avec `=, +, -, /, <, >, ==` …

Opérateurs logiques:

```
bool Rational operator == (Rational &q)
{return (num*q.den == den*q.num);}
```

4. Nombres Complexes

On peut utiliser la même procédure pour construire les nombres complexes.
Les nombres Complexes ne sont pas des types fondamentaux. Ils sont définis dans la class `Complex` dans `complex.h`.

Bien qu' ils existent en C++, la construction d'une classe de nombres complexes est un très bon exercice pour comprendre la construction et le fonctionnement des classes.

z = a + ib: (a,b) sont un couple de `float;` q = p/r: (p,r) était un couple de `int`

Essayez de construire une classe `Complexe` avec les principales opérations sur les Nombres complexes (+, -, *, :, z*, |z|, etc.).
Ajoutez une fonction pour utiliser les nombres complexes aussi en notation polaire.

```
class Complexe {      //trace de la class Complex
  public:
    Complexe(float re_z, float im_z);      //constructeur
    double getmod_z();      //fonction
    double getphase_z();
    void print();
  //Surchargez les operateurs + - * /
  private:
    float re_z, im_z;   //partie reele et partie imaginaire
    float mod_z, phase_z      //notation polaire
};
```