Cours n° 2
Introduction au C++
Module Info3/M3105C

Sebastien.Kramm@univ-rouen.fr

IUT GEII Rouen

2018-2019

Sommaire

UNIVERSITÉ DE ROUEN

Prérequis & Objectifs

- Prérequis :
  - Langage C,
  - Notions de base de P.O.O.
- Objectifs :
  - Premiers éléments concrets sur le C++
  - Être capable :
    - d'écrire la déclaration et le code d'une classe,
    - de l'utiliser dans un programme,
    - de réaliser des entrées/sorties texte "façon C++".

Certains mécanismes ou particularités seront vus plus tard.

UNIVERSITÉ DE ROUEN

Historique

- Bjarne Stroustrup, 1982, A&T Lab

Historique

- Bjarne Stroustrup, 1982, A&T Lab
- Idées de base :

Historique

- Bjarne Stroustrup, 1982, A&T Lab
- Idées de base :
  - Permettre la POO

Historique

- Bjarne Stroustrup, 1982, A&T Lab
- Idées de base :
  - Permettre la POO
  - Améliorer certains points du C

Historique

- Bjarne Stroustrup, 1982, A&T Lab
- Idées de base :
  - Permettre la POO
  - Améliorer certains points du C
  - Garder la compatibilité (un compilateur C++ doit compiler du C)

Historique

- Bjarne Stroustrup, 1982, A&T Lab
- Idées de base :
  - Permettre la POO
  - Améliorer certains points du C
  - Garder la compatibilité (un compilateur C++ doit compiler du C)
- Un langage en évolution :

Historique

- Bjarne Stroustrup, 1982, A&T Lab
- Idées de base :
  - Permettre la POO
  - Améliorer certains points du C
  - Garder la compatibilité (un compilateur C++ doit compiler du C)
- Un langage en évolution :
  - version 2.0 : 1989

UNIVERSITÉ DE ROUEN

Historique

- Bjarne Stroustrup, 1982, A&T Lab
- Idées de base :
  - Permettre la POO
  - Améliorer certains points du C
  - Garder la compatibilité (un compilateur C++ doit compiler du C)
- Un langage en évolution :
  - version 2.0 : 1989
  - 1ère normalisation ANSI/ISO : 1998, puis 2003 (*C++03*)

UNIVERSITÉ DE ROUEN

Historique

- Bjarne Stroustrup, 1982, A&T Lab
- Idées de base :
  - Permettre la POO
  - Améliorer certains points du C
  - Garder la compatibilité (un compilateur C++ doit compiler du C)
- Un langage en évolution :
  - version 2.0 : 1989
  - 1ère normalisation ANSI/ISO : 1998, puis 2003 (*C++03*)
  - Dernières normalisations : 2011 (*C++11*) et 2014 (*C++14*)

UNIVERSITÉ DE ROUEN

Historique

- Bjarne Stroustrup, 1982, A&T Lab
- Idées de base :
  - Permettre la POO
  - Améliorer certains points du C
  - Garder la compatibilité (un compilateur C++ doit compiler du C)
- Un langage en évolution :
  - version 2.0 : 1989
  - 1ère normalisation ANSI/ISO : 1998, puis 2003 (*C++03*)
  - Dernières normalisations : 2011 (*C++11*) et 2014 (*C++14*)
  - La prochaine : 2017

UNIVERSITÉ DE ROUEN

Historique

- Bjarne Stroustrup, 1982, A&T Lab
- Idées de base :
  - Permettre la POO
  - Améliorer certains points du C
  - Garder la compatibilité (un compilateur C++ doit compiler du C)
- Un langage en évolution :
  - version 2.0 : 1989
  - 1ère normalisation ANSI/ISO : 1998, puis 2003 (*C++03*)
  - Dernières normalisations : 2011 (*C++11*) et 2014 (*C++14*)
  - La prochaine : 2017
- Aujourd'hui : C et C++ sont très différents !

UNIVERSITÉ DE ROUEN

Sommaire

UNIVERSITÉ DE ROUEN

C++ par rapport au C : points clés

Ce qui ne change pas :

- Structures de contrôle (`if`, `for`, `while`, ...)
- Types de base (int, float, char)
- Syntaxe générale : `,` `-` `;` `-` `-` `[]` ...
- La bibliothèque standard C reste utilisable (fonctions mathématiques et autres).

Ce qui est "supprimé"

- codes de formats pour les E/S (`"%d"`, `"%f"`, etc)
- les chaines de caractères via des tableaux.
- L'utilisation de fonction bas niveau pour les E/S (`printf()`, `scanf()`, etc.)

C++ par rapport au C : points clés

Ce qui est ajouté :

- commentaires : `//` en début de ligne
- emplacement libre des déclarations de variables
- "flots" pour les E/S texte, sans "codes de format"
- type "chaînes de caractères" (`string`)
- type booléen (`bool`)
- arguments de fonctions par défaut
- passage d'arguments "par référence" : allège les lourdeurs et les risques liés au pointeurs
- surcharge de fonction (même nom pour différentes signatures)
- surdéfinition d'opérateurs (+, -, *, ...)
- "Espaces de noms" (*namespace*), utile pour gros projets.
- et d'autres...

Déclaration de variables "au plus près"

Supprime des possibilités d'erreur :

En C :

```
int a, i, j = 5;
 ...
for( i=0; i<max; i++ )
{
    ...
}
 ...
a = i * 3; // 'j' en fait !
```

⇒ Le compilateur ne détecte pas d'erreur !

Déclaration de variables "au plus près"

Supprime des possibilités d'erreur :

En C :

```
int a, i, j = 5;
 ...
for( i=0; i<max; i++ )
{
    ...
}
 ...
a = i * 3; // 'j' en fait !
```

⇒ Le compilateur ne détecte pas d'erreur !

En C++ :

```
int j = 5;
 ...
for( int i=0; i<max; i++ )
{
    ...      // 'i' n'existe que dans ce
         bloc
}
int a = i * 3 ; //'j' en fait !
```

⇒ L'erreur est détectée à la compilation, 'i' n'existe pas ici

E/S texte

- En C : printf()/scanf() souvent délicat :
  - Erreurs de sens non détectées par le compilateur (par ex., `%d` pour afficher un flottant),
  - Besoin de mémoriser les codes de format.
- En C++ : notion de **flot** ("*stream*").
  - La librairie standard C++ fournit les flots :
    - cin : flot standard d'entrée (équivalents à stdin en C),
    - cout : flot standard de sortie (équivalents à stdout en C).
  - opérateur << pour **injecter** quelque chose dans un flot.
  - opérateur >> pour **extraire** des données d'un flot.
  - Formatage automatique, mais modifiable.

E/S texte : exemple

- Exemple :

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

int main()
{
   int n;
   cout << "taper un nombre : ";
   cin >> n;
   cout << "vous avez tapé : " << n << endl;
}
```

- Remarque : pas d'extension aux fichiers d'en-tête de la bibliothèque standard !

Chaînes de caractères

- En C, les chaînes sont gérées par des tableaux de `char` :

```
char ch1[] = "Bonjour";
char ch2[] = ", Monsieur";
strcat( ch1, ch2 ); // fonction de concaténation
```

$\Rightarrow$ Erreur à l'exécution ! ! !

Chaînes de caractères

- En C, les chaînes sont gérées par des tableaux de `char` :

```
char ch1[] = "Bonjour";
char ch2[] = ", Monsieur";
strcat( ch1, ch2 ); // fonction de concaténation
```

  $\Rightarrow$ Erreur à l'exécution ! ! !

- Inconvénients :
  - Manipulation délicate (utilisation de fonctions : strcpy(), strcat(), ...),
  - Risque élevé de corruption de mémoire,
  - Allocation mémoire pas très optimale, ou compliquée.

Chaînes de caractères

- En C, les chaînes sont gérées par des tableaux de `char` :

```
char ch1[] = "Bonjour";
char ch2[] = ", Monsieur";
strcat( ch1, ch2 ); // fonction de concaténation
```

  $\Rightarrow$ Erreur à l'exécution ! ! !

- Inconvénients :
  - Manipulation délicate (utilisation de fonctions : strcpy(), strcat(), ...),
  - Risque élevé de corruption de mémoire,
  - Allocation mémoire pas très optimale, ou compliquée.
- La bibliothèque standard C++ propose la classe `string`
  - Manipulation aisée : les opérateurs sont surdéfinis,
  - Allocation mémoire automatisée,
  - Plus de risque de corruption mémoire,
  - Conversion du type `char*` au type `string` très facile.

UNIVERSITÉ DE ROUEN

Chaînes de caractères : la classe string

Exemple complet :

```
#include <string>
int main()
{
  std::string s;
  std::cout << "tapez votre nom : ";
  std::cin >> s;
  std::cout << "Bonjour, " << s << "\n";
}
```

Concaténation :

```
string s1( "abc" );    string s2( "def" );
string s = s1 + s2;
cout << s << " : taille=" << s.size() << endl;
```

va afficher :

```
abcdef : taille=
```

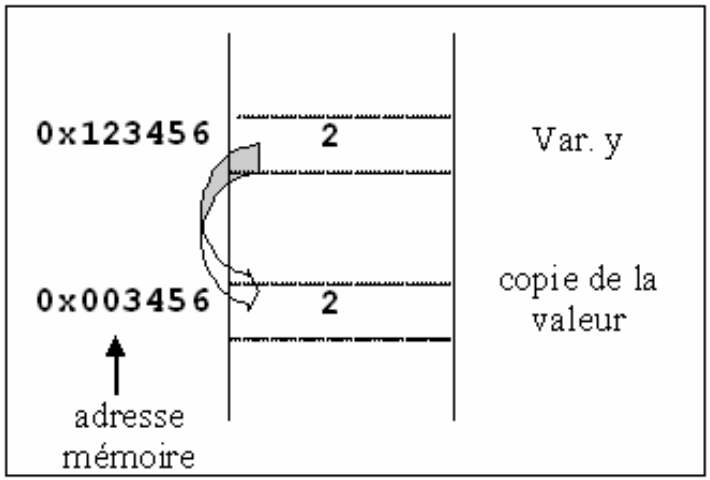
Et bien d'autres facilités encore (recherche, remplacement, insertion, ...)

Rappel : transmission d'arguments par valeur

- En C/C++, lors d'un appel de fonction avec passage d'arguments, la fonction travaille sur une **copie** des arguments (copie sur la **pile**).

```
int main()
{
  int x, y = 2;
  x = carre( y );
}

int carre( int a )
  { return a*a; }
```

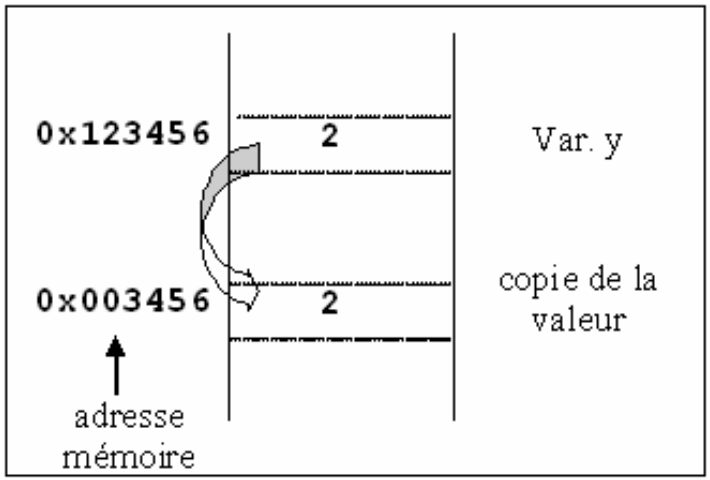


Rappel : transmission d'arguments par valeur

- En C/C++, lors d'un appel de fonction avec passage d'arguments, la fonction travaille sur une **copie** des arguments (copie sur la **pile**).

```
int main()
{
  int x, y = 2;
  x = carre( y );
}

int carre( int a )
  { return a*a; }
```



- Conséquence : il est **impossible** à la fonction de modifier la valeur de la variable originelle
  $\Rightarrow$ elle ne sait pas où celle-ci se trouve dans la mémoire !

Transmission d'arguments par adresse

- Pour écrire une fonction qui doit modifier les arguments transmis, on transmet l'**adresse** de la variable, avec l'opérateur "adresse de" (`&`).

Transmission d'arguments par adresse

- Pour écrire une fonction qui doit modifier les arguments transmis, on transmet l'**adresse** de la variable, avec l'opérateur "adresse de" (`&`).
- Exemple : utilisation de la fonction `scanf()` :

```
int a;
printf( "Tapez un nombre :" );
scanf( "%d", &a );
```

Transmission d'arguments par adresse

- Pour écrire une fonction qui doit modifier les arguments transmis, on transmet l'**adresse** de la variable, avec l'opérateur "adresse de" (&).
- Exemple : utilisation de la fonction `scanf()` :

```
int a;
printf( "Tapez un nombre :" );
scanf( "%d", &a );
```

- La fonction doit travailler sur des pointeurs :

```
int* a; // se lit : "a est une variable qui contient une adresse, à
     laquelle se trouve un 'int'"

int z = *a; // se lit :" copier dans 'z' la valeur de
            // l'entier se trouvant à l'adresse 'a'"
```

Transmission d'arguments par adresse : exemple

- Exemple : fonction de permutation de ses deux arguments :
- Utilisation de la fonction

```
int main()
{
   int x = 3
   int y = 5;
   ...
   swap( &x, &y );
   ...
}
```

⇒ On passe à la fonction **l'adresse** des variables.

Transmission d'arguments par adresse : exemple

- Exemple : fonction de permutation de ses deux arguments :

- Utilisation de la fonction

```
int main()
{
   int x = 3
   int y = 5;
   ...
   swap( &x, &y );
   ...
}
```

⇒ On passe à la fonction **l'adresse** des variables.

- Definition de la fonction

```
void swap( int* a, int* b )
{
   int temp = *a;
   *a = *b;
   *b = temp;
}
```

⇒ La fonction reçoit des **pointeurs** en argument.

C++ : passage d'arguments par référence

- C++ introduit un nouveau mode de transmission des arguments : le **passage par référence**.
- Correspond aussi à une adresse, sans les inconvénients des pointeurs.
- Une référence est un **alias** sur la variable originelle :
  `int& a = b;` se lit *"a est un alias sur la variable b"*.
- La fonction de swap peut être écrite de la façon suivante :
- Utilisation de la fonction

```
int main()
{
   int x = 3
   int y = 5;
   ...
   swap( x, y );
   ...
}
```

C++ : passage d'arguments par référence

- C++ introduit un nouveau mode de transmission des arguments : le **passage par référence**.
- Correspond aussi à une adresse, sans les inconvénients des pointeurs.
- Une référence est un **alias** sur la variable originelle :
  `int& a = b;` se lit *"a est un alias sur la variable b"*.
- La fonction de swap peut être écrite de la façon suivante :

- Utilisation de la fonction

```
int main()
{
   int x = 3
   int y = 5;
   ...
   swap( x, y );
   ...
}
```

- Definition de la fonction

```
void swap( int& a, int& b )
{
   int temp = a;
   a = b;
   b = temp;
}
```

⇒ La fonction reçoit des **références** sur les variables transmises.

Sommaire

UNIVERSITÉ DE ROUEN

Sommaire

En C : Le mot clé struct

- En C, les structures permettent déjà de créer des types de données composés :

```
typedef struct ma_voiture {
    int marque;
    int couleur;
    char immat[8];
} Voiture;
```

En C : Le mot clé struct

- En C, les structures permettent déjà de créer des types de données composés :

```
typedef struct ma_voiture {
   int marque;
   int couleur;
   char immat[8];
} Voiture;
```

- On peut créer en mémoire des objets ("instancier") de ce type comme n'importe quel autre type de variable :

```
int     i;
float   f;
Voiture v;
```

En C : Le mot clé struct

- En C, les structures permettent déjà de créer des types de données composés :

```
typedef struct ma_voiture {
   int marque;
   int couleur;
   char immat[8];
} Voiture;
```

- On peut créer en mémoire des objets (”instancier”) de ce type comme n'importe quel autre type de variable :

```
int     i;
float   f;
Voiture v;
```

- On accède aux données (champs) avec l'opérateur ”.”

```
v.marque = PEUGEOT;
```

UNIVERSITÉ DE ROUEN

Limitations des `struct` en C

Mais :

- On ne peut pas associer des fonctions aux structures ;
- Pas d'initialisation par défaut : les valeurs sont indéterminées ;
- Pas de protection des données (pas d'encapsulation) ;
- Pas d'héritage possible (mais agrégation possible).

UNIVERSITÉ DE ROUEN

Limitations des struct en C

Mais :

- On ne peut pas associer des fonctions aux structures ;
- Pas d'initialisation par défaut : les valeurs sont indéterminées ;
- Pas de protection des données (pas d'encapsulation) ;
- Pas d'héritage possible (mais agrégation possible).

Exemple d'agregation :

```
#include "roue.h"
#include "moteur.h"
struct Voiture
{
   int marque;
   int couleur;
   char immat[8];
   Roue av_g, av_d, ar_g, ar_d;
   Moteur mot;
};
```

Sommaire

P.O.O en C++ : classe d'objet

Le C++ introduit le mot clé **class**, qui permet de déclarer une classe :

- liste des attributs (= "champs" de données),
- liste des méthodes (= fonctions associées à l'objet).

```
// voiture.h

class Voiture
{
  private:
    int marque;
    int couleur;
    bool en_marche;

  public:
    void demarre();
    void arrete();
};
```

- Cette **déclaration** se fait dans un **fichier d'en-tête** (*header file*), avec l'extension .h ou .hpp
- Ce fichier devra être "inclus" (`#include`) dans tous les programmes qui utilisent cette classe.

Partie publique & privée

- On précise quelles sont les attributs/méthodes accessibles (visibles) de l'extérieur avec les mots-clés `private` et `public`.

Partie "`public`"

Cette partie s'appelle l'**interface** : c'est ce qui est accessible de l'extérieur, ce qu'on peut faire avec les objets de ce type.

Partie publique & privée

- On précise quelles sont les attributs/méthodes accessibles (visibles) de l'extérieur avec les mots-clés `private` et `public`.

Partie "`public`"

Cette partie s'appelle l'**interface** : c'est ce qui est accessible de l'extérieur, ce qu'on peut faire avec les objets de ce type.

Partie "`private`"

Cette partie contient les **données** internes de l'objet, elles seront inaccessibles de l'extérieur (principe d'encapsulation).

UNIVERSITÉ DE ROUEN

Partie publique & privée

- On précise quelles sont les attributs/méthodes accessibles (visibles) de l'extérieur avec les mots-clés `private` et `public`.

Partie "`public`"

Cette partie s'appelle l'**interface** : c'est ce qui est accessible de l'extérieur, ce qu'on peut faire avec les objets de ce type.

Partie "`private`"

Cette partie contient les **données** internes de l'objet, elles seront inaccessibles de l'extérieur (principe d'encapsulation).

- L'ensemble des données détermine l'**empreinte mémoire** de la classe : nombre d'octets occupé par un objet de ce type.

UNIVERSITÉ DE ROUEN

Partie publique & privée

- On précise quelles sont les attributs/méthodes accessibles (visibles) de l'extérieur avec les mots-clés `private` et `public`.

Partie "`public`"

Cette partie s'appelle l'**interface** : c'est ce qui est accessible de l'extérieur, ce qu'on peut faire avec les objets de ce type.

Partie "`private`"

Cette partie contient les **données** internes de l'objet, elles seront inaccessibles de l'extérieur (principe d'encapsulation).

- L'ensemble des données détermine l'**empreinte mémoire** de la classe : nombre d'octets occupé par un objet de ce type.

(Remarque : on pourra aussi avoir des méthodes privées, qui ne pourront être appelées **que** par des méthodes publiques.)

UNIVERSITÉ DE ROUEN

Exercice : empreinte mémoire

- Donner l'empreinte mémoire de la classe `Voiture` :
- Donner l'empreinte mémoire de la classe suivante :

```
class ABCDEFG
{
   private:
      char code;
      int v1, v2;
      float resultat;
      char nom[15];
};
```

Définition des méthodes

- La **définition** des méthodes de la classe est effectuée **séparément**, dans le fichier d'**implémentation** de la classe (.cpp) :

```
// voiture.cpp
#include "voiture.h"

void Voiture::demarre()
{
  en_marche = true;
}

void Voiture::arrete()
{
  en_marche = false;
}
```

- On doit utiliser l'**opérateur de résolution de portée** (**::**) pour indiquer l'appartenance à la classe de cette fonction.

Sommaire

UNIVERSITÉ DE ROUEN

Utilisation d'une classe

- Le programme principal (*main.cpp*) va **utiliser** des classes, en plus des types de base (*int*,*float*, . . .)

Utilisation d'une classe

- Le programme principal (*main.cpp*) va **utiliser** des classes, en plus des types de base (*int*,*float*, . . .)
- Ces classes pourront être issues d'un développement personnel, ou d'une bibliothèque tierce.

Utilisation d'une classe

- Le programme principal (*main.cpp*) va **utiliser** des classes, en plus des types de base (*int*,*float*, . . .)
- Ces classes pourront être issues d'un développement personnel, ou d'une bibliothèque tierce.
- Il est impératif d'inclure dans *main.cpp* les fichiers de déclaration des classes utilisées.

UNIVERSITÉ DE ROUEN

Utilisation d'une classe

- Le programme principal (*main.cpp*) va **utiliser** des classes, en plus des types de base (*int*,*float*, . . .)
- Ces classes pourront être issues d'un développement personnel, ou d'une bibliothèque tierce.
- Il est impératif d'inclure dans *main.cpp* les fichiers de déclaration des classes utilisées.
- Comme en C, on **instancie** un objet dans un programme avec une **déclaration**.
  (**objet** et **variable** sont des synonymes) :

UNIVERSITÉ DE ROUEN

Utilisation d'une classe

- Le programme principal (*main.cpp*) va **utiliser** des classes, en plus des types de base (*int*,*float*, . . .)
- Ces classes pourront être issues d'un développement personnel, ou d'une bibliothèque tierce.
- Il est impératif d'inclure dans *main.cpp* les fichiers de déclaration des classes utilisées.
- Comme en C, on **instancie** un objet dans un programme avec une **déclaration**.
  (**objet** et **variable** sont des synonymes) :

UNIVERSITÉ DE ROUEN

Utilisation d'une classe

- Le programme principal (*main.cpp*) va **utiliser** des classes, en plus des types de base (*int*,*float*, . . .)
- Ces classes pourront être issues d'un développement personnel, ou d'une bibliothèque tierce.
- Il est impératif d'inclure dans *main.cpp* les fichiers de déclaration des classes utilisées.
- Comme en C, on **instancie** un objet dans un programme avec une **déclaration**.
  (**objet** et **variable** sont des synonymes) :

```
// main.cpp
#include "voiture.h"
int main()
{
  int     a; // creation d'un objet 'a' de type 'int'
  Voiture b; // creation d'un objet 'b' de type 'Voiture'
   ...
}
```

Protection (encapsulation)

- Le principe d'encapsulation (mot-clé `private`) interdit au programme utilisateur de la classe la manipulation directe des attributs :

```
Voiture v1;
...
v1.marque = PEUGEOT; // écriture
cout << "marque = "<< v1.marque; // lecture
...
```

$\Rightarrow$ Erreur à la compilation !

Accesseurs / Mutateurs (1)

Les attributs sont **privés** (principe d'encapsulation).
On y accède à travers des méthodes (interface), qu'il faut écrire :

- mutateur : écriture (*Set...*),
- accesseur : lecture (*Get...*),.

```
class Voiture {
  private:
    int vitesse_max;
  public:
    int get_vmax(); // renvoie la  vit .  maxi
    void set_vmax( int max ); // modifie la  vit .  maxi
};
```

Accesseurs / Mutateurs (2)

- L'**accesseur** se contente en général de fournir la valeur cachée dans l'implémentation :

```
int Voiture::get_vmax()
{
   return vitesse_max;
}
```

Accesseurs / Mutateurs (2)

- L'**accesseur** se contente en général de fournir la valeur cachée dans l'implémentation :

```
int Voiture::get_vmax()
{
   return vitesse_max;
}
```

- Le **mutateur** vérifie que la nouvelle valeur est cohérente avant de la mémoriser :

```
void Voiture::set_vmax( int max )
{
   if( max < 0 )
      cout << "erreur : vit. < 0!\n";
   else
      vitesse_max = max;
}
```

Instanciation d'une classe

- L'utilisation de l'objet se fait via les **méthodes** de la classe.

```
#include "voiture.h"
int main()
{
  Voiture v1;

  v1.set_vmax( 200 );
  v1.demarre();
   ...
  v1.arrete();
  cout << "vitesse max =";
  cout << v1.get_vmax() << endl;
}
```

Agrégation d'objets

- La déclaration d'une classe peut inclure d'autres objets :

```
// voiture.h

#include "moteur.h"

class Voiture
{
   private:
      Moteur mot;
      int couleur;
       ...
   public:
      void demarre();
       ...
};
```

Agrégation d'objets

- La déclaration d'une classe peut inclure d'autres objets :

```
// voiture.h

#include "moteur.h"

class Voiture
{
   private:
      Moteur mot;
      int couleur;
       ...
   public:
      void demarre();
       ...
};
```

- Dans la définition des méthodes, on pourra accéder aux méthodes **publiques** de l'objet inclus :

```
// voiture.cpp

#include "voiture.h"

void Voiture::demarre()
{
   mot.verifier_huile();
   mot.demarrer();
    ...
}
```

Sommaire

Surdéfinition d'opérateurs

- Les classes permettent de créer de véritables nouveaux types de données, en modifiant le sens des opérateurs classiques
- Exemple : soit un type de donnée "nombre complexe", composé de deux nombres a et b (réel + imaginaire)

Surdéfinition d'opérateurs

- Les classes permettent de créer de véritables nouveaux types de données, en modifiant le sens des opérateurs classiques
- Exemple : soit un type de donnée "nombre complexe", composé de deux nombres a et b (réel + imaginaire)

- En C, on pourra définir une structure regroupant les deux valeurs :

```
typedef struct cplxe {
   float a, b;
} Complexe;
```

Surdéfinition d'opérateurs

- Les classes permettent de créer de véritables nouveaux types de données, en modifiant le sens des opérateurs classiques
- Exemple : soit un type de donnée "nombre complexe", composé de deux nombres a et b (réel + imaginaire)

- En C, on pourra définir une structure regroupant les deux valeurs :

```
typedef struct cplxe {
   float a, b;
} Complexe;
```

- Mais en C, pour additionner deux complexes, il faudra faire l'addition champ par champ :

```
Complexe n1, n2, n3;
 ...
n3.a = n1.a + n2.a;
n3.b = n1.b + n2.b;
```

Surdéfinition d'opérateurs

- Les classes permettent de créer de véritables nouveaux types de données, en modifiant le sens des opérateurs classiques
- Exemple : soit un type de donnée "nombre complexe", composé de deux nombres a et b (réel + imaginaire)

- En C, on pourra définir une structure regroupant les deux valeurs :

```
typedef struct cplxe {
   float a, b;
} Complexe;
```

- Mais en C, pour additionner deux complexes, il faudra faire l'addition champ par champ :

```
Complexe n1, n2, n3;
 ...
n3.a = n1.a + n2.a;
n3.b = n1.b + n2.b;
```

- Le C++ permettra de **définir** le sens de l'opérateur '+' (ou '-') au sein de la classe, et on pourra écrire de façon plus intuitive :

```
Complexe n1, n2, n3;
 ...
n3 = n1 + n2;
```

Initialisations automatique

- En C, la création d'une variable n'entraîne aucune exécution de code : la variable est laissée dans un état indéterminé.

Initialisations automatique

- En C, la création d'une variable n'entraîne aucune exécution de code : la variable est laissée dans un état indéterminé.
- En C++, on pourra avoir une initialisation automatique des variables, grace à une méthode particulière, appelée **constructeur**.
- Cette méthode est exécutée automatiquement (de façon transparente) **à chaque création d'un objet** de ce type.

```
Complexe n1;      // initialisé automatiquement à 0 + 0i
```

Initialisations automatique

- En C, la création d'une variable n'entraîne aucune exécution de code : la variable est laissée dans un état indéterminé.
- En C++, on pourra avoir une initialisation automatique des variables, grace à une méthode particulière, appelée **constructeur**.
- Cette méthode est exécutée automatiquement (de façon transparente) **à chaque création d'un objet** de ce type.

```
Complexe n1;      // initialisé automatiquement à 0 + 0i
```

- On pourra aussi passer des arguments au constructeur, pour initialiser l'objet à une valeur donnée :

```
Complexe n1( 3, 2 );    // n1 = 3 + 2i
```

(Voir cours suivant...)

UNIVERSITÉ DE ROUEN

Pour aller plus loin...

- fr.wikipedia.org/wiki/Programmation_orientée_objet
- fr.wikipedia.org/wiki/C++

Exercice : classe RECTANGLE

- Ecrire une classe RECTANGLE qui modélise un rectangle par sa hauteur et sa largeur, et qui offre les fonctions suivantes :
  - calcul du périmètre : méthode `float Perim()`
  - calcul de la surface : méthode `float Surface()`
  - affichage : méthode `void Affiche()`

  ainsi que les accesseurs et mutateurs triviaux (lecture et modification de la largeur et de la hauteur).