C++ Express

CEMRACS 2012
1er Août

Pascal Havé
<pascal.have@ifpen.fr>

Schedule (1/2)

- Lecture 1
  - Les principes de la programmation orienté objet
  - Courte histoire des langages orienté objet
  - Le C++, du C avec…
  - Le C++, du C avec des classes
  - Héritage, Interface, Polymorphisme
  - Espace de noms, exception

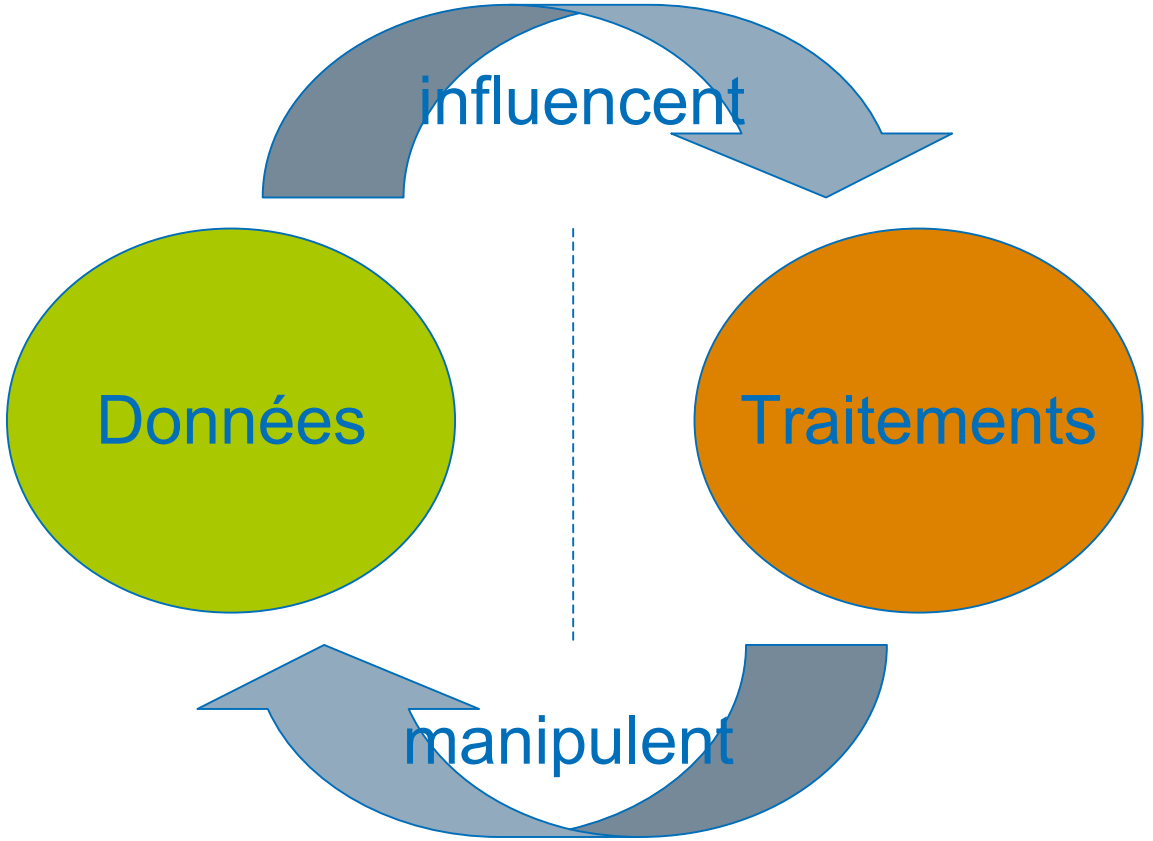
Schedule (2/2)

- Lecture 2
  - Template
    - polymorphisme statique
    - meta-programming
  - La STL
    - Conteneurs
    - Algorithmes
  - Boost
- Lecture 3
  - C++11
  - « Best practices »

Principes de l'Orienté Objet

La programmation procédurale

- Séparation des notions de données et traitements



La programmation procédurale

- Séparation des notions de données et traitements

Ex en C:

```
int main(int argc, char ** argv)
{
  int n, i;
  double * data;

  scanf("%d", n);

  data = (double*)malloc(n*sizeof(double));

  for(i=0;i<n;++i)
  {
    data[i] = 0.;
  }
}
```

La programmation procédurale

- Séparation des notions de données et traitements

Ex en C:

```
int main(int argc, char ** argv)
{
  int n, i;
  double * data;

  scanf("%d",&n);

  data = (double*)malloc(n*sizeof(double));

  for(i=0;i<n;++i)
  {
    data[i] = 0.;
  }
}
```

- mélange données et temporaires
- aucune protection avant utilisation

Allocation

Initialisation

Principes de la POO

- A partir de réflexion sur la vie d'un logiciel
  - Assurer une qualité de réalisation (règles)
  - Mieux maîtriser les coûts liés à la maintenance (corrective / évolutive) : environ 70% du coût total d'un logiciel.
  - La conception objet : méthode pour atteindre une certaine qualité logicielle.

Principes de la POO

- Critères de qualité logicielles
  - correction ou validité : fait qu'un logiciel effectue exactement les tâches pour lesquelles il a été conçu
  - extensibilité : capacité à intégrer facilement de nouvelles spécifications, qu'elles soient demandées par les utilisateurs ou imposées par un événement extérieur
  - réutilisabilité : possibilité d'utiliser certaines parties du code pour résoudre un autre problème.
    Ceci impose lors de la conception une attention particulière à l'organisation du logiciel et à la définition de ses composantes

Principes de la POO

- Critères de qualité logicielles
  - robustesse : aptitude d'un logiciel à fonctionner même dans des conditions anormales. Bien que ce critère soit plus difficile à respecter, les conditions anormales étant par définition non spécifiées lors de la conception d'un logiciel, il peut être atteint si le logiciel est capable de détecter qu'il se trouve dans une situation anormale
  - portabilité : facilité avec laquelle on peut exploiter un même logiciel sous différents environnements
  - efficacité : la rapidité d'exécution, la taille mémoire...

Paradigmes de la POO

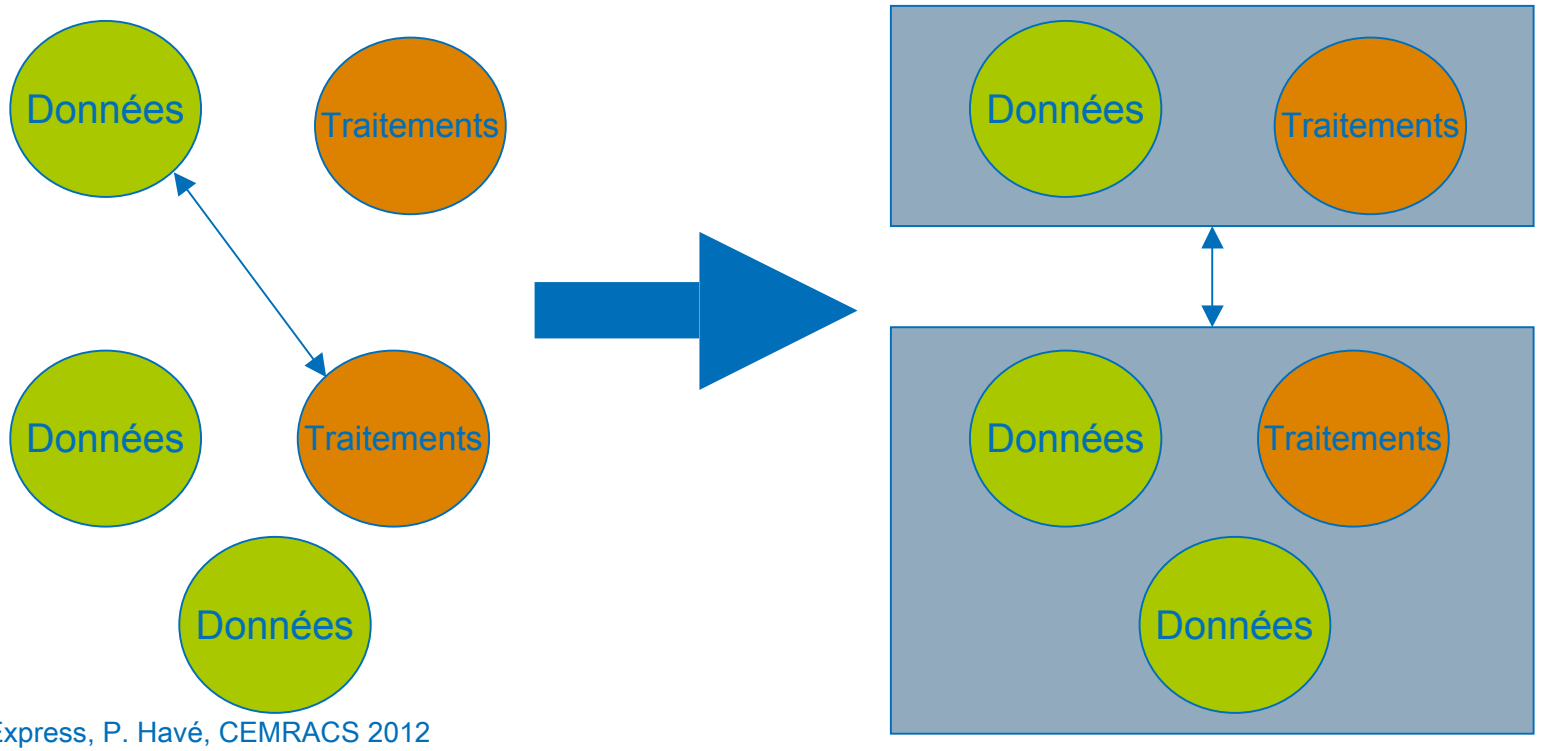
L'abstraction : capacité à ignorer les détails d'un élément pour en avoir une vision globale.

- L'abstraction de code (ou abstraction procédurale) correspond donc à ignorer les détails d'une portion de code pour la voir comme un tout : il s'agit des procédures et des fonctions !
- L'abstraction de données consiste à regrouper des données, ce qui aboutit aux types complexes de données (les structures en C).

- Le modèle objet est une synthèse de ces deux types d'abstraction : un objet est à la fois une abstraction de données et une abstraction de code.

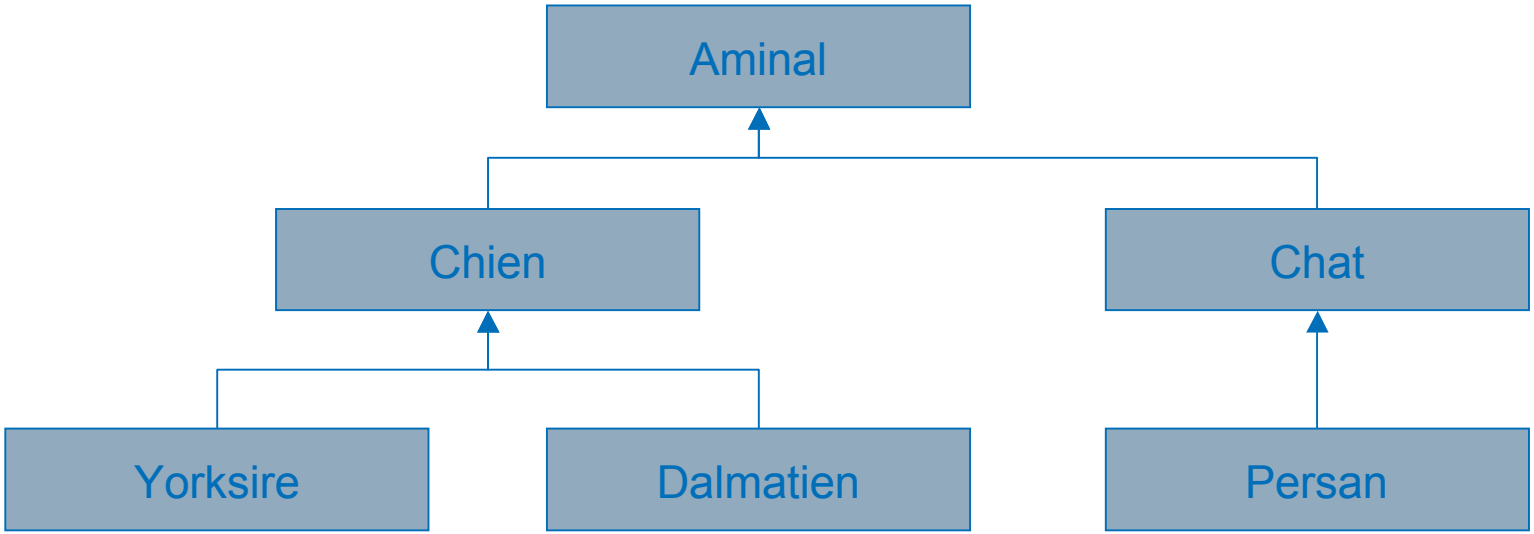
Paradigmes de la POO

- L'encapsulation
  - L'encapsulation, c'est le masquage de certains éléments au sein d'une entité (objet).
    Ils ne sont plus visibles (ou accessibles) de l'extérieur de l'entité : ceci permet de garantir une intégrité
    La mise en oeuvre rigoureuse de ce concept lors de la conception favorise la réutilisabilité et la maintenance.

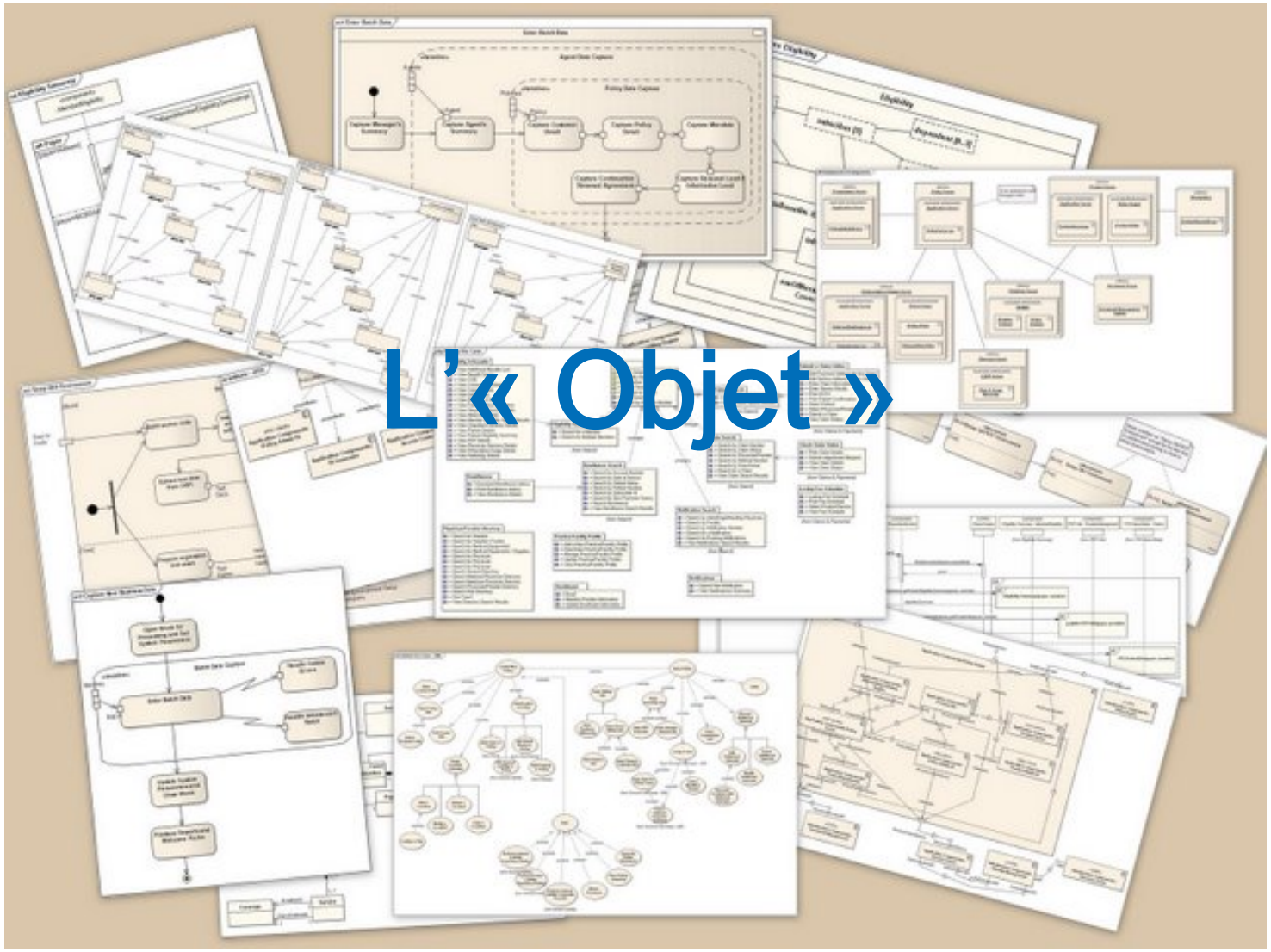


Paradigmes de la POO

- Héritage et polymorphisme
  - L'héritage c'est la capacité qu'une entité a à transmettre patrimoine (données et code).
  - Le polymorphisme traite de la capacité d'une entité à possèder plusieurs formes.
    Le polymorphisme par inclusion s'appuie sur l'héritage et permet de manipuler une entité enfant en la voyant sous la forme d'un de ses parents : principe de substitution de Lyskov*

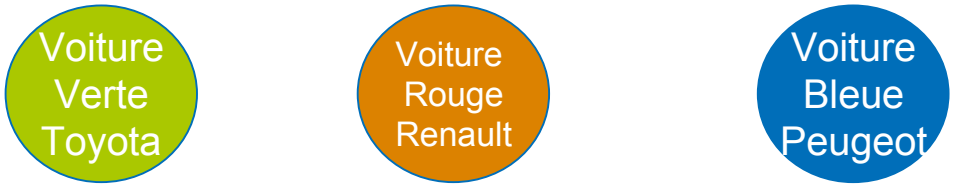


* : *Si q(x) est une propriété démontrable pour tout objet x de type T, alors q(y) est vraie pour tout objet y de type S tel que S est un sous-type de T.*



Objet et classes

- 3 objets

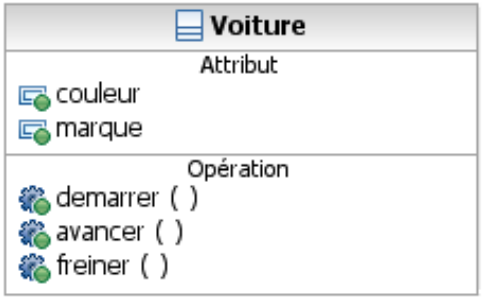


- Des voitures
- Elles font partie de la classe des voitures

  - La classe Voiture est leur *description*
  - Se sont des instances de Voiture

Objet et attributs

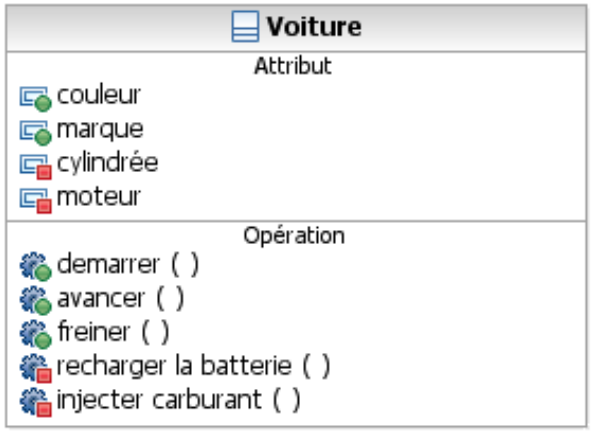
- La classe Voiture



- La Voiture a des données : les membres ou attributs
  - Couleur
  - Marque
- La Voiture a des actions : les méthodes ou opérations
  - démarrer
  - avancer
  - freiner

Objet et protection

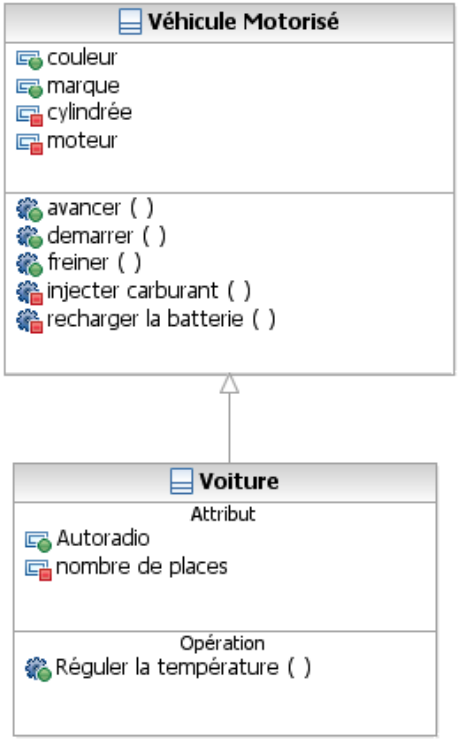
- La Voiture ne montre pas tout à l'extérieur



- Elle garde certains attributs ou méthodes privés
  - Attributs privés
    - Moteur
    - Cylindrée
  - Méthodes privées
    - recharger la batterie
    - injecter essence

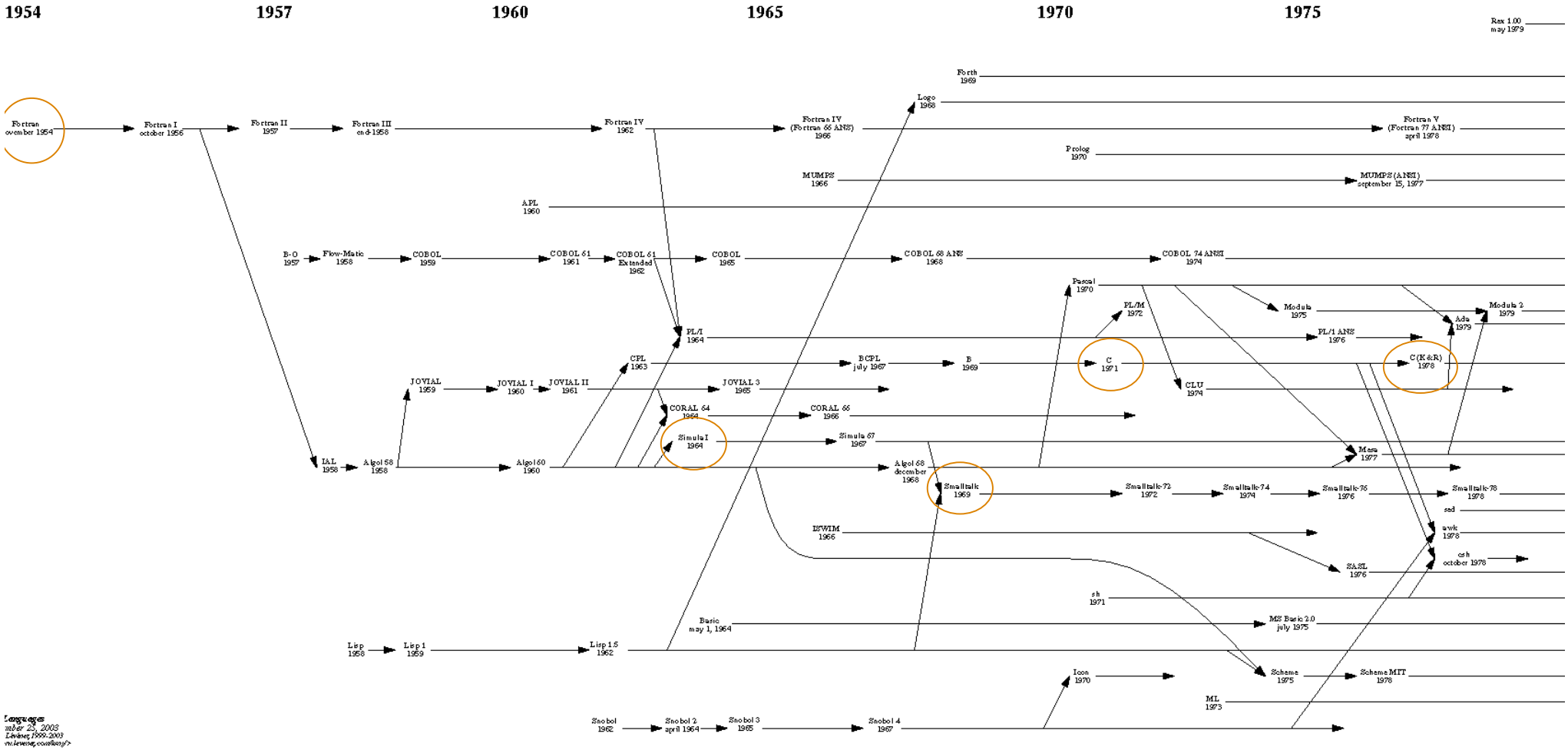
Objet et héritage

- La Voiture est une spécialisation de Véhicule



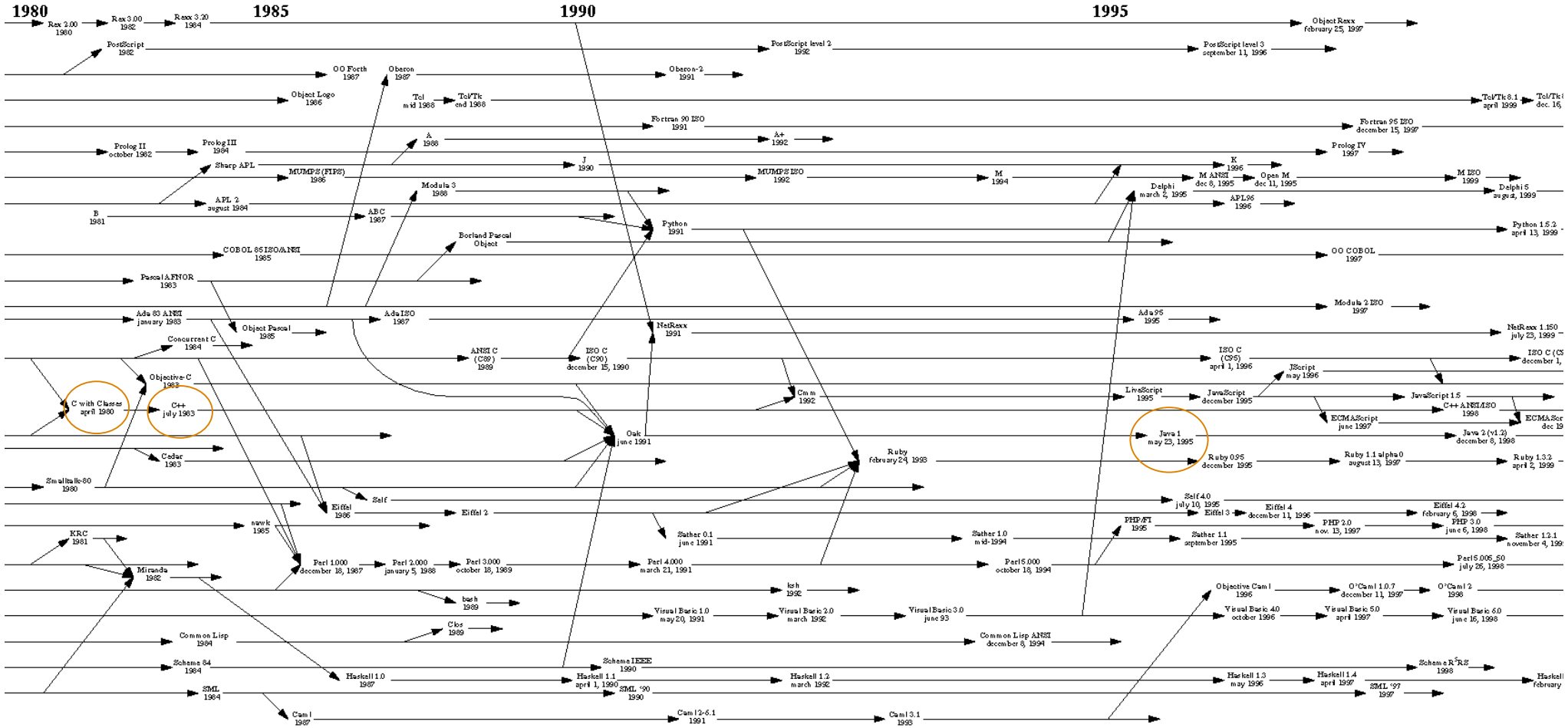
- La Voiture est un Véhicule Motorisé
- Le Véhicule Motorisé généralise la Voiture
- Ce qui n'empêche pas la Voiture d'avoir des attributs/opérations propres.

Courte histoire des langages orientés objets



- Langage Simula ; introduction de la notion d'objet (1967) et de classe (1973).
- SmallTalk (Alan Kay - Xerox PARC), un langage tout objet introduit la notion d'envoi de message (communication entre les objets; 1972) puis d'héritage (1976).
- L'ADA (1983) met en place le principe d'encapsulation.

Courte histoire des langages orientés objets



- Naissance du C++ : "C avec Classes" (1980), C++ (B. Stroustrup - AT&T Laboratory 1982, 1986): une évolution du C vers les concepts objets
- Le C++ a été standardisé (norme ANSI / ISO) en 1997.

...

- Eiffel (1986), Python (1991), Java (1995), Ruby (1995), D (1999), C# (2001), C++11…

Courte histoire des langages orientés objets

TIOBE Programming Community Index

Position Jul 2012	Position Jul 2011	Delta in Position	Programming Language	Ratings Jul 2012	Delta Jul 2011	Status
1	2	↑	C	18.331%	+1.05%	A
2	1	↓	Java	16.087%	-3.16%	A
3	6	↑↑↑	Objective-C	9.335%	+4.15%	A
4	3	↓	C++	9.118%	+0.10%	A
5	4	↓	C#	6.668%	+0.45%	A
6	7	↑	(Visual) Basic	5.695%	+0.59%	A
7	5	↓↓	PHP	5.012%	-1.17%	A
8	8	=	Python	4.000%	+0.42%	A
9	9	=	Perl	2.053%	-0.28%	A
10	12	↑↑	Ruby	1.768%	+0.44%	A
11	10	↓	JavaScript	1.454%	-0.79%	A
12	14	↑↑	Delphi/Object Pascal	1.157%	+0.27%	A
13	13	=	Lisp	0.997%	+0.09%	A
14	15	↑	Transact-SQL	0.954%	+0.15%	A
15	25	↑↑↑↑↑↑↑↑↑↑	Visual Basic .NET	0.917%	+0.43%	A
16	16	=	Pascal	0.837%	+0.17%	A
17	19	↑↑	Ada	0.689%	+0.14%	B
18	11	↓↓↓↓↓↓	Lua	0.684%	-0.89%	B
19	21	↑↑	PL/SQL	0.645%	+0.10%	A–
20	26	↑↑↑↑↑↑	MATLAB	0.639%	+0.19%	B

Evolution à long terme de l'indice

Programming Language	Position Nov 2011	Position Nov 2006	Position Nov 1996	Position Nov 1986
Java	1	1	5	-
C	2	2	1	1
C++	3	3	2	5
C#	4	8	-	-
PHP	5	5	-	-
Objective-C	6	43	-	-
(Visual) Basic	7	4	3	7
Python	8	7	26	-
JavaScript	9	9	25	-
Perl	10	6	6	-
Lisp	13	16	16	3
Ada	19	18	12	2

Référence:
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Le C++, du C avec…

- Le C a des avantages
  - C'est un langage très rapide car assez bas niveau.
  - C'est un langage *portable*. Le même code source peut être compilé aussi bien sous Windows, Linux, Mac OS et ne dépend d'aucun type de processeur particulier.
  - Le langage est libre; grande diversité des compilateurs : GCC, mingw, MS Visual C++, Borland, Intel…

- Mais le C a aussi des *défauts*
  - Tout langage a des défauts (contexte)
  - Absence de certains concepts récents
    - Ex: les références (*alias* plus *simple* que les pointeurs), les exceptions (puissante technique puissante pour gérer les erreurs)

    (mais c99 et + comblent le *retard*)
  - Mais le vrai problème du langage C est qu'il n'est pas prévu pour faire de la OO, technique de programmation reconnue particulièrement efficace (même si cela n'est pas *impossible*)

Le C++, du C avec…

- Programmes triviaux

```
int main() { return 0; }
```

```
int main(int argc, char ** argv)
{
  return 0;
}
```

```
#include <cstdio>
int main(int argc, char ** argv)
{
  fprintf(stdout, " Hello World! " );
  return 0;
}
```

Le C++, du C avec…

- Forte compatibilité avec le C
  - Structures de code et structures de contrôle: { … }, for, do, while, if, else, ?: (if inline), switch, case, break, continue
  - Déclaration et affectation de variables *simples*;
    - <initialisation> ::= <identifiant> [« = » <valeur>]
    - <liste_initialisation> ::= <identifiant> [« , » <liste_initialisation>]
    - <déclaration> ::= [<qualificateur>] <type> <liste_initialisation>
    - <affectation> ::= <initialisation>

```
int a;
long b = 3l;
double d = 3.0, e;
float f[3] = { 3., 1., 4. };
static char g[] = "chaîne de caractères";
struct H * h;
a = 0xFF;
e; /* commentaire: valeur de e ? */
```

  - **Pas d'initialisation par défaut sur les types de base**

Le C++, du C avec…

- Forte compatibilité avec le C
  - Déclaration et affectation de variables structurées;
    - <type_struct> ::=
      « struct » [<identifiant>] « { »
      <liste_declaration>;
      « } »
    - S'utilise ensuite comme un <type>

```
struct Point3D
{
  double x;
  double y, z;
};

struct Point3D p;
p.x = p.y = p.z = 0;
```

Le C++, du C avec…

- Forte compatibilité avec le C
  - Mêmes opérateurs:
    - Opérateurs arithmétiques: +, -, *, /, %
    - Opérateurs binaires: &, |, ^, <<, >>
    - Opérateur logiques: &&, ||, ==, !=
    - Opérateur avec affectation: =, &=, +=, >>= …

```
int a = 1;
a += a;
return ( a & 3 != 0 ); // valeur retournée ?
```

  - Alias de type avec **typedef**
    « typedef » <type_original> <nouveau_type>

```
typedef unsigned char byte;
byte code[] = { 0x3, 0xEF, 0x92 };
```

  - enum
    « enum » [<identifiant>] « { » <liste_identifiant> « } »

```
enum MonEnum { ValeurA, ValeurB };
```

Le C++, du C avec…

- Déjà quelques variations
  - Nouveau type simple: **bool** (type « logique » de valeurs true/false)
  - Les variables ne sont pas obligatoirement déclarées en tête de bloc
  - Nouveau qualificateur: **const**
    Le contenu est *protégé* contre la modification.
    (protégé uniquement par le compilateur)
  - Ajout de la notion de référence: **&**
    Une référence est un alias mémoire,
    mais sans pointeur apparent.
  - Le sens des opérateurs peut être redéfini !
    (sauf sur les types de base: *cf* compatibilité)

```
int i, n = 10;
/* … */
const int p = 3;
bool flag = (4 == i);
if (flag == true)
{
  const int x; // illégal, non défini
  p = 4; // illégal
  int & p_alias = p; // violation const
  int & n_alias = n;
  n_alias = 20;
  const int * pointeur1;
  const int * const pointer2 = &i;
}

cout << 123 << n;
```

Le C++, du C avec…

- Déjà quelques variations
  - Les enum et struct sont des types; ces qualifications sont superflus.

```
enum Flag
{
  Vrai,
  False
};

struct Point3D
{
  double x, y, z;
};

typedef struct
{
  double x, y, z;
} AutrePoint3D;
```

```
Flag reponse = Vrai;
Point3D p;
AutrePoint3D q;
const Point3D & r = p; // référence constante
AutrePoint3D & s = p; // illégal, type différent!
const int i; // illegal: type simple non initialisé
const Point3D t; // Ok, mais pas très utile
```

Le C++, du C avec…

- Prototypage et pré-processing
  - Le prototype déclare avant l'utilisation

Fichier proto.h

```
#ifndef PROTO_H
#define PROTO_H

int ma_fonction(int * p);
extern int global_data;

#endif /* PROTO_H */
```

  - Le pré-processing complète le langage via des *#commandes*

Non intégré au langage

Fichier proto.c

```
#include "proto.h"
#include <cassert>
int ma_fonction(int * p)
{
  assert(p != 0);
  return *p;
}
```

Le C++, du C avec…

- La portée et visibilité

```
// zone de portée globale
int global_data;

int une_fonction()
{
  // portée locale (bloc)
  int une_variable_locale = 1;
  {
    int une_variable_locale;
    /* masquage de la précédente variable_locale */
   une_variable_locale = 2;
    double une_autre_variable_locale;
  }
  /* … */
  return une_variable_locale; /* valeur retournée ? */
}
```

Le C++, du C avec…

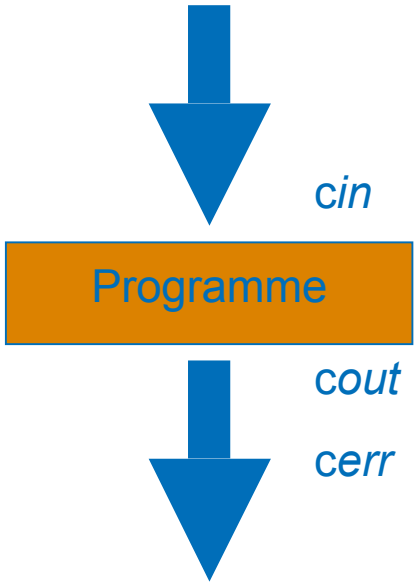
- Des nouveautés simplificatrices:
  - Les I/O (entrées/sorties) par flux

    Via les opérateurs << et >> redéfinis pour l'occasion.

```
#include <iostream>

int main()
{
  int i;
  float f;

  std::cin >> i >> f;
  std::cout << "Valeur de i = " << i << std::endl;
  std::cout << "Valeur de f = " << f << "\n";
  return 0;
}
```



A comparer avec:

```
scanf("%d %f",&i,&f);
printf("Valeur de i = %d\n",i);
printf("Valeur de f = %f\n",f);
```

Le C++, du C avec…

- Des nouveautés simplificatrices:
  - Allocation mémoire
    - allocation isolée: new, delete
    - allocation de tableau: new[], delete[]

```
int * a = new int;
*a = 10;

double * d = new double[10];
for(int i=0;i<=10;++i) d[i] = 0;

Point3D * p = new Point3D;

delete p;   // désallocation d'une structure
delete[] d; // désallocation d'une allocation tableau
delete a;   // désallocation simple
```

  - A comparer avec

```
double * d = (double*)malloc(10*sizeof(double));
free(d);
```

Les entités ici ainsi allouées ne sont pas initialisées!

Le C++, du C avec…

- Des nouveautés simplificatrices:
  - Allocation mémoire
    - allocation isolée: new, delete
    - allocation de tableau: new[], delete[]

```
int * a = new int;
*a = 10;

double * d = new double[10];
for(int i=0;i<=10;++i) d[i] = 0;

Point3D * p = new Point3D;

delete p;   // désallocation d'une structure
delete[] d; // désallocation d'une allocation tableau
delete a;   // désallocation simple
```

  - A comparer avec

```
double * d = (double*)malloc(10*sizeof(double));
free(d);
```

Les entités ici ainsi allouées ne sont pas initialisées!

Le C++, du C avec…

- Des nouveautés simplificatrices:
  - Allocation mémoire: rappel technique
    - La valeur **0** est la valeur de référence du pointeur nul (nommée NULL en C).
    - Tout pointeur devrait être initialisé par cette valeur, ainsi qu'après un désallocation.
    - Le déréférencement d'un pointeur nul ou non initialisé provoque une erreur de segmentation.
    - Il est légal de désallouer un pointeur nul.

```
int * a = 0;
delete[] a;
delete a;
```

    - La double désallocation est un bug! (*double free*)

  - Le C++ ne vérifie pas les débordements de ses tableaux *natifs*.
    - Risque de corruption mémoire (SegFault si vous avez de la chance)
  - Le C++ ne fournit pas de ramasse-miettes (garbage collector)
    - Désallocation *oubliée* = fuite mémoire

Le C++, du C avec…

- Polymorphisme ad hoc ou surcharge
  - un nom
  - plusieurs signature
  - adaptation au contexte d'appel

```
double somme(double a, double b) { return a+b; }
int somme(int a, int b) { return a+b; }
```

  - Est-ce possible sur les opérateurs *standards* ?
    - Oui, ils ont tous une expression sous forme de fonction: operator***X***

```
c = a + b;
c = operator+(a,b);
```

    - Pour des raisons de compatibilité, ces opérateurs ne sont pas surchargeables sur les types simples.

```
double operator+(double a, double b) { return a*b; }
```

    - C'est ainsi que operator<< et operator>> ont pu acquérir un sens I/O

Le C++, du C avec…

Priorité et associativité des opérateurs (surchargés ou non)

Tous surchargeables

- Sauf:
  - .
  - ?:
  - sizeof
- Mais aussi:
  - new
  - new[]
  - delete
  - delete[]
  - operator *type*

Opérateurs (par priorité descendante)	Associativité
() [] -> ->* .	→
! ~ ++ -- + - * & (*int*) sizeof	←
* / %	→
+ -	→
<< >>	→
< <= > >=	→
== !=	→
&	→
^	→
|	→
&&	→
||	→
?:	←
= += -= *= /= %= &= ^= |= <<= >>=	←
,	→

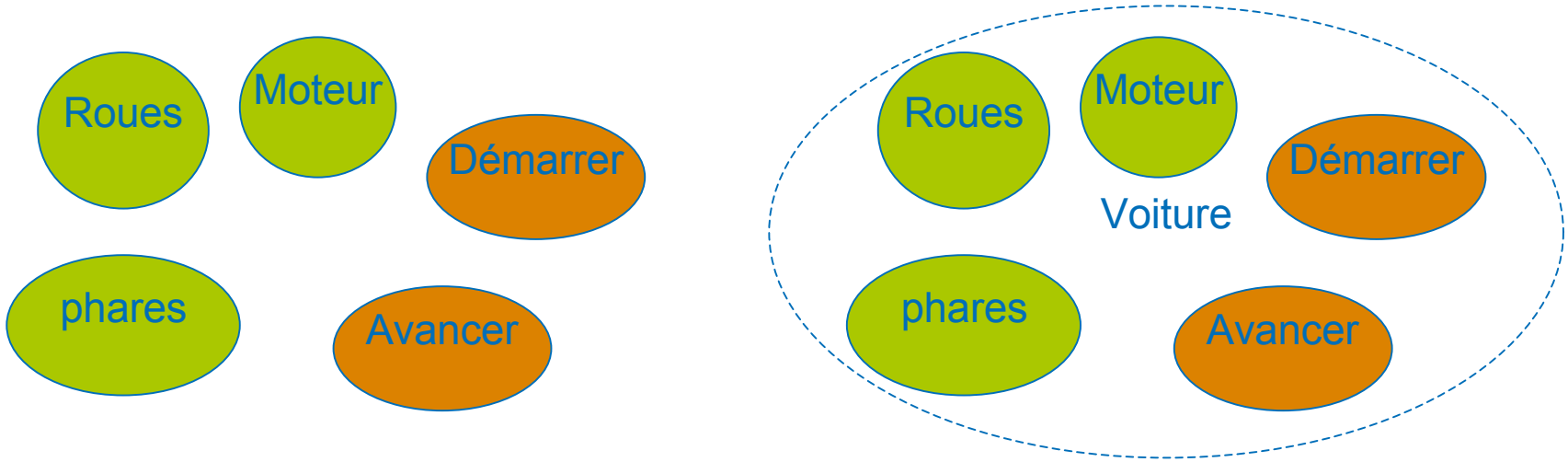
Le C++, du C avec...

la classe

Le C++: les objets

- Qu'est qu'un objet:
  - Des données : les *membres*
  - Des actions : les *méthodes*

- Objectif
  - Réunion en une entité des données et des actions formant un concept : l'encapsulation



Le C++ : la classe

- La fusion en struct et des *fonctions*

```
struct Voiture
{
  int nombre_de_roues;
  Roues * roues;
  Moteur moteur;
  Phare * phare;
};

void demarrer(Voiture * v, int clef);
void avancer(Voiture * v, double vitesse);
```

Fichier Voiture.h

```
class Voiture
{
public:
  int nombre_de_roues;
  Roues * roues;
  Moteur moteur;
  Phare * phare;

  void demarrer(int clef);
  void avancer(double vitesse);
};
```

Le C++ : la classe

- Tout ce qui est public est accessible, modifiable
  - Risque de corruption

- comme struct la protection en plus

Fichier Voiture.h

```
class Voiture
{
private:
  int nombre_de_roues;
  Roues * roues;
  Moteur moteur;
  Phare * phare;
public:
  void demarrer(int clef);
  void avancer(double vitesse);
};
```

- NB: le struct C++ est comme une classe publique par défaut

Le C++ : la classe

- De la déclaration au corps

Fichier Voiture.cpp

```
#include "Voiture.h"

void
Voiture::
demarrer(int clef)
{
  if (clef == 314)
    moteur.lancer();  accès aux membres et méthodes directs
}

void
Voiture::
avancer(double vitesse)
{
  for(int i=0;i< nombre_de_roues;++i)  accès direct à un membre
    roues[i].tourner(vitesse);
  phare->allumer();  accès indirect à une méthode
}
```

Le C++ : la classe

- Construction et …
  - Le constructeur est appelé au moment de la *construction*
    - Voiture v;
    - Voiture * pv = new Voiture();

Fichier Voiture.h

```
class Voiture
{
public:
  Voiture();   // constructeur
  ~Voiture(); // destructeur
  /* … */
};
```

Fichier Voiture.cpp

```
Voiture::
Voiture()
{
  phare = new Phare();
}
```

  - NB: Pas d'argument de retour (pas même *void*)

Le C++ : la classe

- … et destruction
  - Le destructeur est appelé au moment de la *destruction*
    - Passage en hors portée
    - delete p;

Fichier Voiture.h

```
class Voiture
{
public:
  Voiture();   // constructeur
  ~Voiture(); // destructeur
  /* … */
};
```

Fichier Voiture.cpp

```
Voiture::
~Voiture()
{
  delete phare;
}
```

Le C++ : la classe

- Le constructeur peut avoir plusieurs formes…
  - par surcharge du constructeur

```
Voiture::Voiture(int identifiant);
Voiture::Voiture(int identifiant, const char * nom);
```

  - *par copie* avec une version par défaut

    - Copie champs à champs
    - N'est pas identique à operator=
    - Il est plus sûr de toujours garder synchroniser le comportement du constructeur par copie et de operator=.

- … et être appelé de différentes manières
  - Construction avec ou sans argument
  - en retour de fonction par copie

```
Voiture v1;
Voiture v2(12340092);
Voiture v3 = v2;
```

Le C++ : la classe

- La notion de classe intègre l'héritage (version simple)

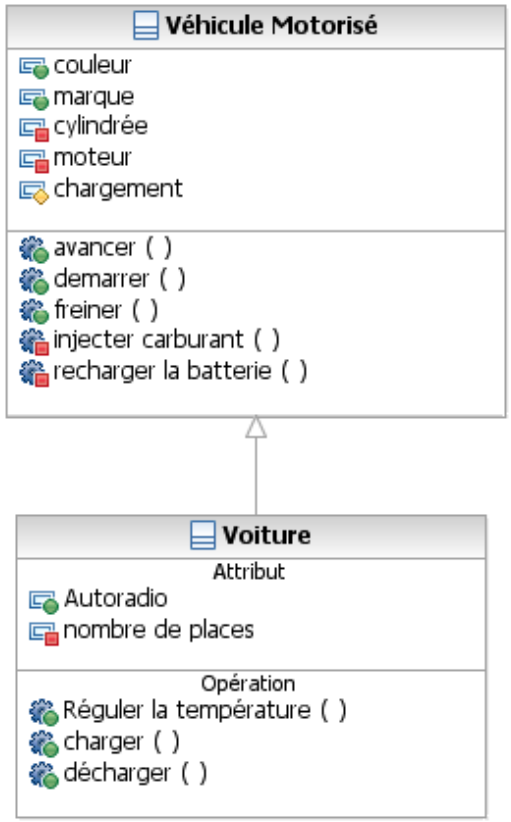
```
class VehiculeMotorise
{
public:
  Couleur couleur;
  Marque marque;
private:
  Cylindree cylindree;
  Moteur moteur;
protected:
  Chargement chargement;

public:
  void avancer();
  void demarrer();
  void freiner();
private:
  void injecterCarburant();
  void rechargerLaBatterie();
};
```

```
class Voiture
  : public VehiculeMotorise
{
public:
  Voiture() : VehiculeMotorise() { }

public:
  Autoradio autoradio;
private:
  int nombreDePlace;

public:
  void regulerLaTemperature();
  void charger();
  void decharger();
};
```



Le C++ : la classe

- Accès et héritage
  - Tout le monde accède aux champs/méthodes publiques (public)
  - Uniquement les instances de la classe accèdent à ses champs/méthodes privées (private)
  - Seuls les instances de la classe parente ou de la classe enfant accèdent aux champs/méthodes protégés de la classe parente
  - L'héritage *par défaut* est private.

Droit hérité résultant de l'héritage		Type d'héritage		
		public	protected	private
Type de protection dans la classe parente	public	public	protected	private
	protected	protected	protected	private
	private	private	private	private

Le C++ : la classe

- A quoi sert la protection ?
  - Contrôler les accès aux objets
  - Transmettre un patrimoine de manière contrôlée (protected)
  - Masquer les détails d'implémentation

- Conseil:
  - Eviter de mettre des données publiques, elles seront modifiables sans aucun contrôle de validation par la classe; préférer des accesseurs;

```
void charger(Chargement chargement);
Chargement decharger();
                                    ou
void setChargement(Chargement chargement);
Chargement chargement(); // attn collision de nom avec champs
Chargement getChargement();
```

Le C++ : la classe

- Zoom sur la construction/destruction des objets

```
int main()
{
  Voiture v;
}
```

Construction

- Voitrure::Voiture
  - VehiculeMotorise::VehiculeMotorise
    - Moteur::Moteur
      - *Code constructeur Moteur*
    - Chargement::Chargement
      - *Code constructeur Chargement*
    - *Code constructeur VehiculeMotorise*
  - Autoradio::Autoradio
    - *Code constructeur Autoradio*
  - *Code constructeur Voiture*

Destruction

- Voitrure::~Voiture
  - *Code destructeur Voiture*
  - Autoradio::~Autoradio
    - *Code destructeur Autoradio*
  - VehiculeMotorise::~VehiculeMotorise
    - *Code destructeur VehiculeMotorise*
    - Chargement::~Chargement
      - *Code destructeur Chargement*
    - Moteur::Moteur
      - *Code destructeur Moteur*

Le C++ : la classe

- NB sur la construction/destruction des objets
  - Les types de base ne sont pas initialisés (ex: nombreDePlace)
  - Les pointeurs ne sont:
    - ni initialisés
    - ni alloués
    - ni désalloués

Les initialiser avec 0

  - La copie d'objet active toute la mécanique
    - Ex avec:

```
void charger(Chargement chargement);
Chargement decharger();
```

    - Préférer:

```
void charger(const Chargement & chargement);
const Chargement & decharger();
```

Le C++ : la classe

- Accès plus fin aux classes
  - *this*, pointeur sur l'instance courante
    - Permet de lever des ambiguïtés
    - Permet s'envoyer (adresse de l'instance en cours)

```
void charger(const Chargement & chargement)
{
  this->chargement = chargement; // sans this : ambiguïté
}
```

  - static, qualificateur d'attributs/méthodes de classe
    - Indépendant de toute instance
    - Equivalent à une donnée/fonction globale 
    - Aucune garantie sur l'ordre d'initalisation

```
class Voiture {
/* ... */
private:
  static int m_compteur;
public:
  static int compteur() { return m_compteur; }
};
```

```
int Voiture::m_compteur = 0; // initialisation

Voiture::Voiture()
{
  ++m_compteur;
};
```

```
std::cout << Voiture::compteur() << std::endl;
```

Le C++ : la classe

- Accès plus fin aux classes
  - Méthodes *const*
    - Indique au compilateur que la méthode *const* ne peut pas modifier l'instance en cours

```
void charger(const Chargement & chargement) const
{
  this->chargement = chargement; // interdit
  // l'accès aux méthodes constantes de l'instance sont autorisées
}

const Chargement & decharger() const
{
  // sur l'instance en cours, uniquement des méthodes constantes sont autorisées
  return chargement;
}
```

    - Non compatible avec *static*
    - Le qualificateur *const* fait partie de la signature des méthodes
      - Surcharges possibles : le contexte définira le bon candidat

```
Data & Vector::operator[](int i);
const Data & Vector::operator[](int i) const;
```

Le C++ : la classe

- Accès plus fin aux classes
  - Données *const*
    - Indique au compilateur que la donnée ne peut être modifiée
    - Quelle utilité ?
      - Permet d'éviter toute corruption d'un concept après son initialisation
    - Comment l'initialiser ?
      - Via le constructeur de la classe uniquement

```
class Voiture
{
public:
  Voiture() : nombre_de_roues(4)
  {
    nombre_de_roues  = 4; // illégal, trop tard :/
  }

private:
  const int nombre_de_roues;
};
```

Ne pas confondre:

- const Type; Type const
- const Type * ; Type const *
- Type * const

Le C++ : la classe

- Héritage et polymorphisme
  - Perte des spécificités des enfants par héritage simple

```
Voiture v;
VehiculeMotorise & vm = v;
// Depuis vm plus rien n'est accessible de Voiture
```

  - Alors comment manipuler un enfant sans tout perdre ?

    Les interfaces
    - définir un contrat pour les enfants
    - ne pas l'implémenter (interface pure)
    - laisser aux enfants l'implémentation
    - pouvoir appeler l'implémentation enfant depuis le parent

    - Possibilité de définir un comportement par défaut

      Avec modération: risque de bugs en cas de changement d'interface

Le C++ : la classe

Héritage et polymorphisme

Mot clef : virtual

- Permet de propager un appel du père vers l'enfant (le plus profond)

```
VehiculeMotorise vm;   // non instanciable par
                       // contient du virtuel pur
Voiture v;
VehiculeMotorise * pv = &v; // On ne connaît plus la
                            // Voiture
pv->avancer(); // mais on peut la manipuler
pv->freiner();
```

Avec virtual : propagera l'appel à l'enfant

Sans virtual : ne propagera pas l'appel à l'enfant (même si l'enfant l'indiquera virtual)

```
class VehiculeMotorise
{
public:
  VehiculeMotorise();
  virtual ~VehiculeMotorise() { /* … */ }
public:
  virtual void avancer() = 0;
  virtual void freiner() { /* implémentation par défaut */ }
          void tourner() { /* implémentation */ }
};
```

```
class Voiture : public VehiculeMotorise
{
public:
  Voiture() : VehiculeMotorise() { }
  virtual ~Voiture() { /* … */ }
public:
  virtual void avancer() { /* implémentation */ }
  virtual void freiner() { /* autre implémentation */ }
  virtual void tourner() { /* implémentation par défaut */ }
};
```

Dans l'enfant, le qualificatif virtual sur une méthode héritée virtual du parent est facultatif mais permet d'indiquer l'intention de propager l'appel.

Le C++ : la classe

- Héritage et polymorphisme
  - Mais qui suis-je ?
    - Le typage dynamique

```
Voiture v;
VehiculeMotorise * pv = &v; // On ne connaît plus la
                            // Voiture
Voiture * w = dynamic_cast<Voiture*>(pv);
if (w != 0)
{ /* je suis une voiture */ }
else
{ /* je ne suis pas une voiture */ }
```

  - Avantages
    - On peut se retrouver après s'être perdu
    - Reconnaître à l'exécution qui est qui
  - Inconvénients
    - Fort risque de violation du principe de Liskov !

Le C++ : la classe

- Héritage et polymorphisme
  - Je sais qui je suis
    - Le typage statique

```
Voiture v;
VehiculeMotorise * pv = &v; // On ne connaît plus la
                            // Voiture
Voiture * w = static_cast<Voiture*>(pv);
{ /* je suis une voiture */ }
```

    - La validité du transtypage statique est à la charge du développeur.

Bannir le transtypage *à la C* qui est trop imprécis : risque de comportement au delà des besoins

~~`Voiture * w = (Voiture*)(pv);`~~

NB: La qualification *const* ne peut être qu'égal ou croissante à travers un transtypage dynamique ou statique.

- L'utilisation des interfaces permet de se passer d'une grande partie des pointeurs de fonctions du C
  - Les foncteurs: les classes fonction.

C++ : les exceptions

- Problèmatique : la propagation d'erreur
  - Les codes de retours (succès / erreur)
    - Faciles à *oublier*
    - Propagation à la charge du développeur
    - Occupe une sortie utile
- Solution : les exceptions

```
/* fonctionnement sans protection */

try {
  /* Code avec possibilité de disfonctionnement
   * en zone protégée
   * Le code du bloc s'arrêtera immédiatement
   * après la génération d'une exception via throw
   */
}
catch (…)
{
 /*gestion éventuelle du disfonctionnement */
}

/* Retour à un fonctionnement sans protection */
```