ArtWorks

Advanced C++

Généricité

Microsoft Certified Professional Solution Developer Trainer

CompTIA. Certified Technical Trainer+

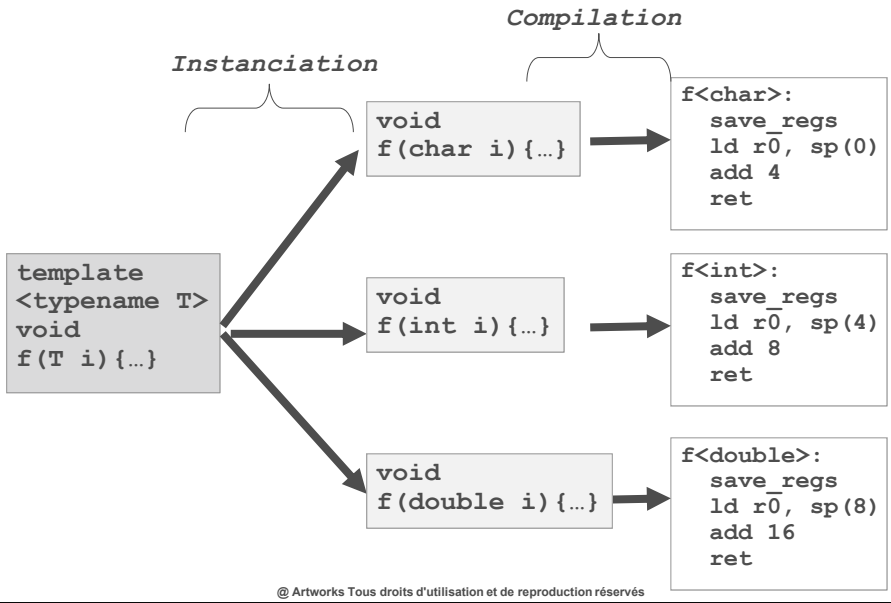
Michel André
michel@artworks.fr

Généricité avancée

- **Fonctions génériques**
- Classes génériques
- Variadic templates
- Métaprogrammation
- Les concepts
- Techniques de conception
- Workshop

2

Instanciation/compilation de code



3

Les fonctions template

```
template <typename T>                    peut être déclarée constexpr
T max (T one, T two){
   return one < two ? two : one;
}

   facultatif
template<>                               spécialisation
inline const char* max(const char* one, const char* two){
   return strcmp(one,two) < 0 ? two : one;
}

int main(){
  cout << ::max("adam", "zorro") << '\n';
  cout << ::max(321, 123) << '\n';
  cout << ::max(321.3, 123.8) << '\n';
  cout << ::max<int>(321.3, 123.8) << '\n';
}
```

```
zorro
321
321.3
321
```


4

Typage multiple

C++ 14

```
template <class ONE, typename TWO>
ONE add(ONE one, TWO two){return one + two;}

int main(){
 string two {"23"};
 cout << add(12, 23)  << "\n";
 cout << add(12, 2.8) << "\n";
 cout << add(2.8, 12) << "\n";
 cout << add<int,int>(2.8, 12) << "\n";
 cout << add(12, two) << "\n";
}
```

troncature

troncature

échec

```
35
14
14.8
14
```

```
template <class ONE, typename TWO>
auto add(ONE one, TWO two) -> decltype(one + two){
  return one + two;
}
```

C++ 11

```
template <class ONE, typename TWO>
auto add(ONE one, TWO two){
  return one + two;
}
```

C++ 14

5

Inférence du typage de fonctions

```
template <typename FUNC >
const string& alphaConvert(string& str, FUNC convert){
  for (size_t i {}; i < str.size(); ++i)
    if (isalpha(str[i]))
      str[i] = convert(str[i]);
  return str;
}

int main(){
  string str {"CamelCaseWord"};
  cout << alphaConvert(str, ::toupper) << "\n";
  cout << alphaConvert(str, ::tolower) << "\n";
}
```

```
CAMELCASEWORD
camelcaseword
```


6

Inférence des références

```
template <typename T>
void f(T t) { t = 99; }

int main() {
  int a{};
  int& ra{ a };

  f(ra);
  cout << a << endl;

  f<int&>(ra);
  cout << a << endl;
}
```

?

?

7

Inférence des tableaux

```
template <class T>
  void myFunc1(T* type){cout << typeid (type).name() << endl;
}
template <class T>
  void myFunc2(T& type){cout << typeid (type).name() << endl;
}
template <class T>
  void myFunc3(T type){cout << typeid (type).name() << endl;
}
```

parameter decay

```
int tab[] { 23, 78, 134 };
cout << typeid (tab).name();
myFunc1(tab);
myFunc2(tab);
myFunc3(tab);
```

```
int [3]
int *
int [3]
int *
```

8

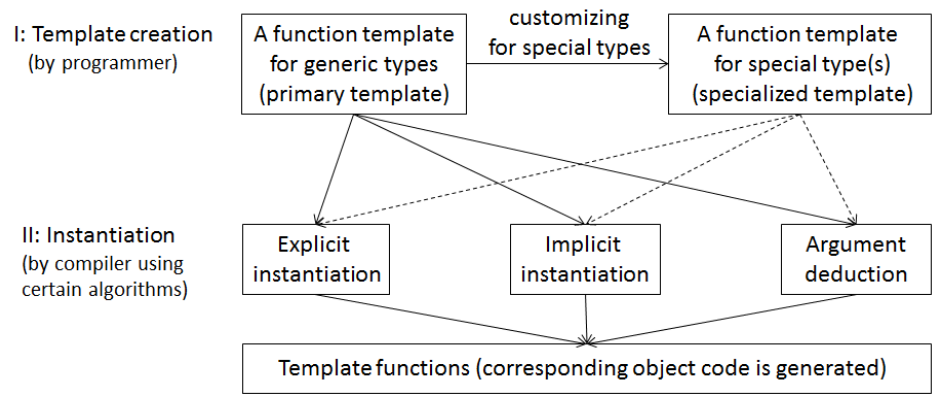
Spécialisation partielle et matching

```
template <typename T>
void f (T t){cout << "un: " << t << "\n";}

template <typename T>
void f (T* t){cout << "deux: " << *t << "\n";}


int main(){
  int n{99};

  f(n);
  f(&n);
}
```

compile ?
exécution ?

10

Processus de création et d'instanciation



11

Substitution

```
template <typename T>
void foo(const T& t) {cout << "template " << t << "\n";}
void foo(int i) {cout << "unsigned " << i << "\n";}
```

```
foo(123.67);
```

compile ?
exécution ?

Les variables templates

C++ 14

```
template<typename T>
const T PI{ static_cast<T>(3.1415926535897932385)};

template<typename T>
T getCircleArea(T radius) { return PI<T> * radius * radius; }
```

```
cout << getCircleArea<int>(123) << "\n" <<
  getCircleArea<double>(123.5) << "\n";
```

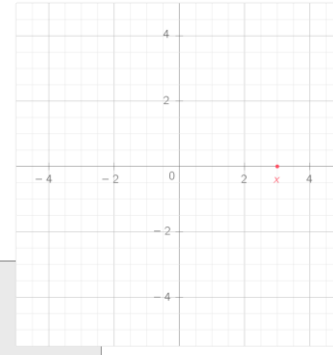
```
45387
47916.4
```

Généricité avancée

- Fonctions génériques
- **Classes génériques**
- Variadic templates
- Métaprogrammation
- Les concepts
- Techniques de conception
- Workshop

Classes non génériques

```
class PointInt {
  int x;
  int y;
public:
  PointInt(int x, int y) :
    x{ x },
    y{ y }
  {
  }
  double getDistance() const;
};
```




```
class PointDouble {
  double x;
  double y;
public:
  PointDouble(double x,double y) :
    x{ x },
    y{ y }
  {
  }
  double getDistance() const;
};
```

Classe générique

```
template <typename T>
class Point {
  T x;
  T y;
public:
  Point(T x, T y) :
    x{ x },
    y{ y }
  {
  }
  double getDistance() const;
};

template <class T>
double Point<T>::getDistance() const{
  return sqrt (x*x + y*y);
}
```

Point.h

```
#include <Point.h>

int main(){
  Point<int> pt1 {12, 78};
  Point<double> pt2 {0, 65.1};

  cout << pt1.getDistance() << "\n";
}
```


16

Spécialisation d'une classe générique

```
template <typename T>
struct A {
  T doThis(T t) { return t; }
};

template <>
struct A <string> {
  string doThis(const string& str);
};

inline string A<string>::doThis(const string& str) {
  return str + str;
}
```

```
cout << A<float>{}.doThis(3) << endl;
cout << A<string>{}.doThis("hello") << endl;
```

```
3
hellohello
```


18

Déduction de classe via le constructeur

C++ 17

before C++17

```
pair<int, double> p(123, 23.56);
```

constructeur paramétré

```
pair p(123, 23.56);
```

C++17

19

Paramètres par défaut

```
template <typename T, typename Container = deque<T>>
class stack {
  // …
};
```

implicite

```
stack<int> ms;
stack<int, list<int>> ms2;
```

explicite

20

Paramétrage / spécialisation de méthode

```
class A {
 public:
  void doThis(){}
  void doThat(){}

  template <typename T>
  T* create(){
    return new T;
  }
};
```

classe non paramétrée

méthode paramétrée

```
A a;
a.doThis();
int* n { a.create<int>()};
// …
delete n;
```


21

Constructeur paramétré et delete

```
class A {
  long value;
 public:
  A(long value) : value{ value }{}

  template<typename T>
  A(T) = delete;
};
```

```
A b {12L};
A b2 {3};
```

ok

échec

22

Paramètres non typés

- entiers, énumérations, pointeurs vers fonctions / objets / membres

```
template <typename T, size_t SIZE>
class array {
  //…
};
```

```
array<int, 4> myArray {12, 89; 4, 90};
cout << myArray.size() << "\n";
```


Paramètres non typés et auto

C++ 17

```
template <typename T, T value>
struct MyStruct {
  // ...
};

MyStruct <long long, 1'000'000'000> ms1;
MyStruct <int, 123> ms2;
```

avant C++17

```
template <auto value>
struct MyStruct {
  // ...
};

MyStruct <1'000'000'000> ms1;
MyStruct <123> ms2;
```

après

Extended friend declaration

```
template <typename T> class A {
  // …
  friend class T;
};
```

problème si T n'est pas une classe

```
template <typename T> class A {
  // …
  friend T;
};

class MyClass;

A <MyClass> a;
A <int> a2;
```

C++ 11

OK

25

Alias et types génériques

```
template <typename T>
class stack {
public:
  class iterator {
    // ...
  };
};
```

```
template<typename T>
using MyIterator = typename stack<T>::iterator;
```

```
stack<float> s;
MyIterator<float> itor2 {begin(s)};
```

```
template <class MOTEUR, class ROUE>
class Voiture {
  // …
};

template <class ROUE>
using VoitureElectrique = Voiture<MoteurElectrique, ROUE>;
```

typename restant à préciser

26

Généricité avancée

- Fonctions génériques
- Classes génériques
- **Variadic templates**
- Métaprogrammation
- Les concepts
- Techniques de conception
- Workshop

Exemple n°1: Création générique

template parameter pack

```
template <typename T, typename... Types>
T* createObject(Types... params) {
  return new T { params... };
  // …
}
```

function parameter pack

```
struct A {
  A(int n, bool b) {}
};

A* pa {createObject <A>(123, false)};
string* pS {createObject <string>("bonjour")};
```

- sizeof…(Types) ou sizeof… (args) retournent le nombre d'arguments

Exemple n°2: Surcharge générique

critère d'arrêt

```
void print(){cout << endl;}

template <typename T, typename... Types>
void print(const T& firstArg, Types... args){
  cout << firstArg << " ";
  print(args...);
}
```

```
print(2, 34.78, "bonjour");
cout << "\n";
print("hello");
```

```
2 34.78 bonjour

hello
```

autre critère d'arrêt possible

```
template <typename T>
void print(T t){cout << t << endl;}
```


Généralisation multiple

```
struct A { void a(){ cout << "a - "; } };
struct B { void b(){ cout << "b - "; } };

template <typename... BaseClasses>
class MyClass : public BaseClasses ...{
public:
  void f(){ cout << "f\n"; }
};
```

```
MyClass<A, B> mc;
mc.a();
mc.b();
mc.f();

MyClass<A> mc2;
mc2.a();
mc2.f();
```

```
a - b - f
a - f
```

Pattern matching (1 / 2)

```
template<typename T>
bool pair_comparer(const T& a, const T& b) {
  return a == b;
}

template<typename T, typename... Args>
bool pair_comparer(const T& a, const T& b,
  const Args&... args)
{
  return a == b && pair_comparer(args...);
}

int main() {
  cout << boolalpha << pair_comparer (1,1,2) << endl;
  cout << boolalpha << pair_comparer (1,1.3) << endl;
}
```

compile ?

compile ?

31

Pattern matching (2 / 2)

```
template<typename T>
bool pair_comparer(const T& t) {
  return false;
}

int main() {
  cout << boolalpha << pair_comparer (1, 2, 3) << endl;
}
```

compile

false

32

Fold expressions …

C++ 17

- application d'un opérateur binaire sur les éléments d'un pack
  - left fold: depuis la gauche vers la droite
  - right fold: depuis la droite vers la gauche

```
template<typename... Args>
auto sum(Args... args){
  return (args + ... + 0);
}

template<typename... Args>
auto avg(Args... args){
  return (args + ... + 0) / sizeof… (args);
}

int main(){
  cout << sum(1, 2, 3.2, 4, 5, 6, 7) << "\n";
  cout << avg(1, 2, 3, 4, 5, 6, 7) << "\n";
}
```

left fold

valeur initiale (facultative)

```
28.2
4
```


33

Fold expressions, schéma général

C++ 17

EXPRESSION	EXPANSION
(... op pack)	((pack1 op pack2) op ...) op packN
(init op ... op pack)	(((init op pack1) op pack2) op ...) op packN
(pack op ...)	pack1 op (... op (packN-1 op packN))
(pack op ... op init)	pack1 op (... op (packN-1 op (packN op init)))

OPERATOR	DEF. VALUE
&& (and)	true
|| (or)	false
,	void()

```
template<typename T, typename... Args>
void push_back(vector<T>& v, Args&&... args) {
  (v.push_back(args), ...);
}
```

comma operator

34

Autre exemple

C++ 17

cout est l'init

application de forward chacun des éléments du pack

```
template<typename ...Args>
void print(Args&&... args){
  (cout << ... << std::forward<Args>(args)) << '\n';
}
```

```
print("hello", ", ", 10, ", ", 90.0);
```

```
hello, 10, 90.0
```


35

Généricité avancée

- Fonctions génériques
- Classes génériques
- Variadic templates
- **Métaprogrammation**
- Les concepts
- Techniques de conception
- Workshop

36

Méta programmation

- Des problèmes initialement résolus à l'exécution peuvent l'être à la compilation

```
template <unsigned N>
struct facto {
  static constexpr unsigned value{
    N * facto<N - 1>::value };
};

template<>
struct facto<0> {
  static constexpr unsigned value{ 1 };
};
```

```
cout << facto<5>::value << endl;
```

```
cout << power<5, 3>::value << endl;
cout << binary<10010110>::value << endl;
```

TODO

Spécialisation de classe

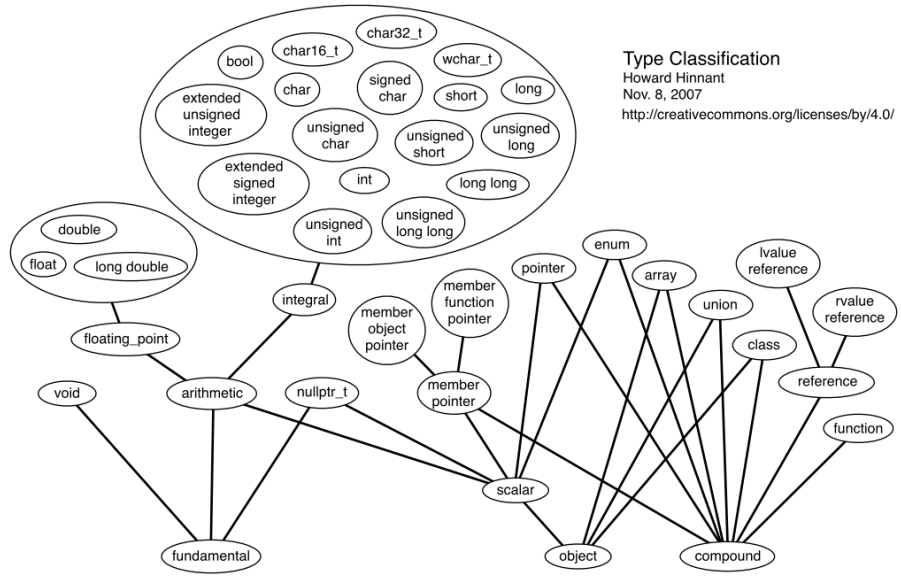
trait

```
template <typename T>
struct is_int {
  static const bool value {false};
};

template <>
struct is_int<int> {
  static const bool value {true};
};
```

```
cout << boolalpha << is_int<float>::value << endl;
cout << is_int<int>::value << endl;
```

```
false
true
```

Typologie C++



Assertions statiques et traits

- Une assertion statique permet de vérifier à la compilation qu'une valeur est vraie

```
#include <type_traits>

static_assert(is_integral<T1>::value,
  "T1 must be integral");

static_assert(is_integral_v<T2>);
```

message d'erreur à la compilation

C++ 17

C++17, le message est facultatif

```
static_assert(NB_PORTIERES > 1,
  "le nombre de portieres est incorrect");
```

<type_traits> - C++ Reference (cplusplus.com)

Tester un type

S'agit-il du même type ?

```
static_assert (is_same_v<Moteur, MOTEUR>);
```

Y a t'il une spécialisation ?

```
static_assert (is_base_of_v<Moteur, MOTEUR>);
```

super-type

sous-type

41

Simplification des type traits

C++ 17

```
std::is_pointer<T>::value
std::add_const_t<T>::type
```

C++ 11

```
std::is_pointer_v<T>
std::add_const_t<T>
```

C++ 17

```
namespace std {
  template<typename T>
  using add_const_t<T> = typename add_const<T>::type;
}
```


42

Traits liés à la typologie

- ils sont très nombreux: is_pod, is_scalar, is_trivial, is_trivial_constructible, is_litteral_type, is_object, is_signed, ….
- .. et sont très souvent composés entre eux …

```
struct A { int i; };
struct B : A { int j; };



static_assert (!is_copy_constructible<A>::value);
static_assert (is_assignable<A, A>::value);
static_assert (is_copy_assignable<A>::value);
static_assert (is_copy_assignable<B>::value);
static_assert (is_final<B>::value);
```

erreurs à la compilation
si les assertions ne sont pas vérifiées

constexpr if

C++ 17

- diminue l'intérêt de SFINAE et de la compilation conditionnelle

```
if constexpr(cond)
  statement1;
else
  statement2;
```

supprimé si cond est faux

supprimé si cond est vrai

```
template <typename T>
auto get_value(T t) {
  if constexpr (is_pointer_v<T>)
    return *t;
  else
    return t;
}
```

Quelle version est la plus claire ?

C++ 17

avant

```
template<int N>
constexpr int fibonacci() {
  return fibonacci<N-1>() + fibonacci<N-2>();
}
template<>
constexpr int fibonacci<1>() { return 1; }
template<>
constexpr int fibonacci<0>() { return 0; }
```

après

```
template<int N>
constexpr int fibonacci() {
  if constexpr (N >= 2)
    return fibonacci<N - 1>() + fibonacci<N - 2>();
  else
    return N;
}
```

```
cout << fibonacci<3>() << endl;
```


45

Généricité avancée

C++ 20

- Fonctions génériques
- Classes génériques
- Variadic templates
- Métaprogrammation
- **Les concepts**
- Techniques de conception
- Workshop

46

Pros of concepts

C++ 20

- Empower programmers to directly express their requirements as part of the interface.
- Support the overloading of functions and the specialization of class templates based on the requirements of the template parameters.
- Produce drastically improved error messages by comparing the requirements of the template parameter with the applied template arguments.
- Can be used as placeholders for generic programming.
- Empower you to define your concepts.

Polymorphisme statique

C++ 20

contrat non intrusif

```
template <typename T>
concept Shape = requires (T t) {
  {t.draw()}-> void;
};

template <typename T>
requires Shape<T>
void display(const T& t) {
  t.draw();
}
```

polymorphisme statique

```
struct Circle {void draw() const {cout << "circle\n"; }};
struct Rectangle {void draw() const {cout << "rectangle\n";}};

int main() {
  display(Circle{});
  display(Rectangle{});
}
```

Généricité avancée

C++ 20

- Fonctions génériques
- Classes génériques
- Variadic templates
- Métaprogrammation
- Les concepts
- **Techniques de conception**
- Workshop

49

Héritage / containment et généricité

- un type générique peut être dérivé ou embarqué

```
template <class T1, class T2>
class A : public T1 {
  T2 t2;
};
struct B{/* */};
struct C{/* */};
```

```
int main(){
  A<B,C> a;
  /* */
}
```

- Une "policy" est un fragment de comportement pluggable
  - n'est pas instancié directement
    - superclasse ou membre

50

Le pattern Command

```
template <class T, class... Args>
class Command {
  T& object;
  void (T::*method)(Args... args);
 public:
  Command(T& object, void (T::*method)(Args...)) :
    object{ object }, method{ method }{}
  void operator()(Args... args) const {
    (object.*method)(args...);
  }
};
```

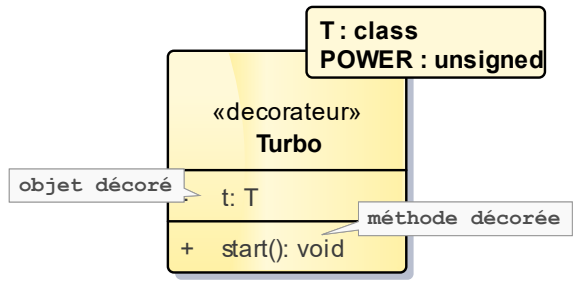
```
struct Player {
  void play(int volume){ cout << "playing " <<
    volume << " dB" << endl; }
  void stop(){ cout << "stop" << endl; }
};

Player p;
Command<Player, int> cmd1{ p, &Player::play };
Command<Player> cmd2{ p, &Player::stop };
cmd1(100); cmd2();
```

paramètre

51

Le pattern Décorateur

- un décorateur enrichit la compétence métier de l'objet décoré
  - la méthode décorée est présente dans les deux classes




```
Voiture <Turbo<Moteur, 3>> v;
```

décoré

décorateur

52

CRTP: Curiously Recurring Template Pattern

- Template Method Pattern

```
template <class T>
struct Modèle {
  void doIt(){
    static_cast<T*>(this)->hook1();
    // …
    static_cast<T*>(this)->hook2();
  }
};

struct A : public Modèle<A> {
  void hook1(){ cout << "A::hook1\n";}
  void hook2(){ cout << "A::hook2\n";}
};

struct B : public Modèle<B> {
  void hook1(){ cout << "B::hook1\n";}
  void hook2(){ cout << "B::hook2\n";}
};
```

```
int main(){
  A{}.doIt();
  B{}.doIt();
}
```


Généricité avancée

- Fonctions génériques
- Classes génériques
- Variadic templates
- Métaprogrammation
- Les concepts
- Techniques de conception
- **Workshop**

Taxinomie animale

- Herbivore
  - mange de l'herbe
- Quadrupède
  - se déplace en trottant
- Bipède
  - se déplace en marchant
- Omnivore
  - mange de tout
- Carnivore
  - mange de la viande
- Prédateur
  - carnivore
  - tue sa proie
- Oiseau (Bird)
  - vole
  - omnivore par défaut
- Rapace (Raptor)
  - oiseau prédateur

Taxons

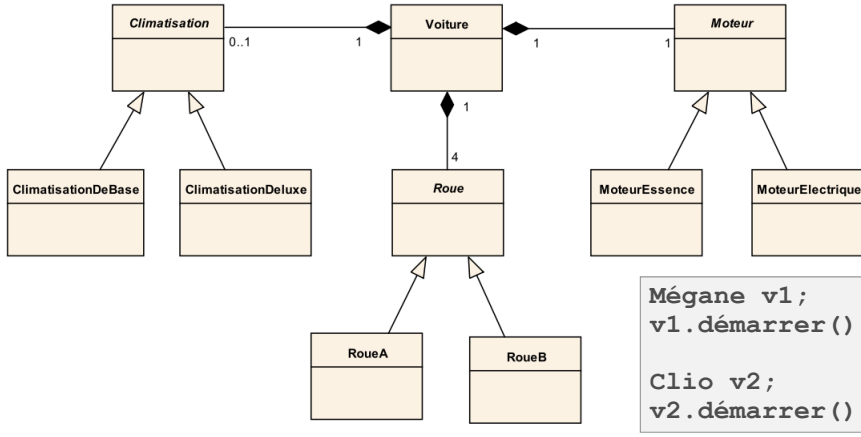
Animaux

```
Gazelle
Moineau
Singe
Aigle
Guépard
```

55

Séries automobiles

- Une Mégane possède un moteur électrique, des roues A et pas de climatisation
- Une Clio possède un moteur à essence des roues B et une climatisation de luxe
- Une TurboClio possède un moteur 3 fois plus puissant




```
Mégane v1;
v1.démarrer();

Clio v2;
v2.démarrer();
```


56

Généricité avancée

- Fonctions génériques
- Classes génériques
- Variadic templates
- Métaprogrammation
- Les concepts
- Techniques de conception
- Workshop