ArtWorks

Advanced C++

Héritage avancé

Lookup rule 1 : héritage et overloading

```
struct A {
  void display(double d) const { cout << d << endl; }
  void display(float f) const { cout << f << endl; }
};

struct B : A {
  void display(int n) const { cout << n << endl; }
};
```

```
int main(){
  B b;
  b.display(4.2f);
}
```

?

Overloading et ambiguïté

méthode hors de portée pour un client de B

```
struct A {
  void foo(int*){}
};

struct B : A {
  void foo(const string&){}
};

int main(){
  B b;
  b.foo("hello");

  int n;
  b.foo(&n);
}
```

ne compile pas

4

Résolution avec la clause using

```
struct A {
  void f(int){}
};

struct B : A {
  void f(double*){}
  using A::f;
};

struct C : B {
  void f(const char*){}
  using B::f;
};
```

```
double d {2.9};

C c;
c.f(12);
c.f(&d);
c.f("bonjour");
```

compile

5

Lookup rule 2 : héritage et abstraction

```
struct A {
  virtual void f() = 0;
  virtual void g() = 0;
};

struct B : A {
  void f() override {}
};

struct C : A {
  void g() override {}
};

struct D: B, C {
};
```

```
int main(){
  D d;
  d.f();
  d.g();
}
```

compile ?

6

Lookup rule 3 : héritage et valeur par défaut

```
struct Base {
  virtual void fun(int x = 0) {
    cout << "Base x = " << x << endl;
  }
};

struct Derived : public Base {
  void fun(int x) override {
    cout << "Derived x = " << x << endl;
  }
};
```

```
int main() {
  Derived derived;
  derived.fun();
  Base& base{ derived };
  base.fun();
}
```

?

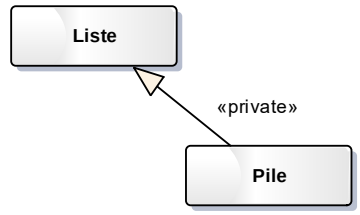
?

7

Héritage privé

```
class Dérivée : private Base {
 public :
   using Base::methodeDeBase;
};
```

optionnel

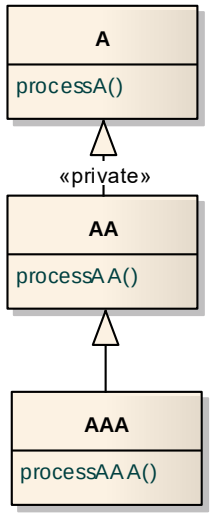


- L'implémentation seule est héritée, pas l'interface
  - la classe dérivée peut décider des méthodes de la superclasse à exposer

```
int main(){
  Base * p {new Dérivée};
}
```

échec

Héritage privé et accès




```
class A {
public:
  void processA(){}
};
class AA : private A {
public:
  void processAA(){this->processA();}
};
class AAA : public AA {
public:
  void processAAA(){this->processA();}
};
```

compile

ne compile pas

Application: classe non copiable

```
class noncopyable {
protected:
  constexpr noncopyable () = default;

  noncopyable(const noncopyable &) = delete;
  noncopyable& operator=(const noncopyable &) = delete;
};
```

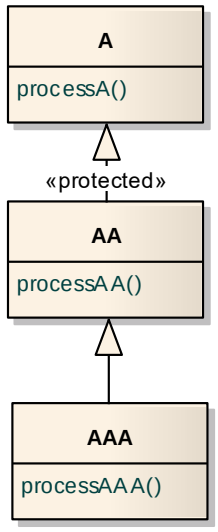
classe abstraite …

… et non copiable

```
class MyClass : noncopyable {
// …
};
```

MyClass devient non copiable par héritage

Héritage protégé

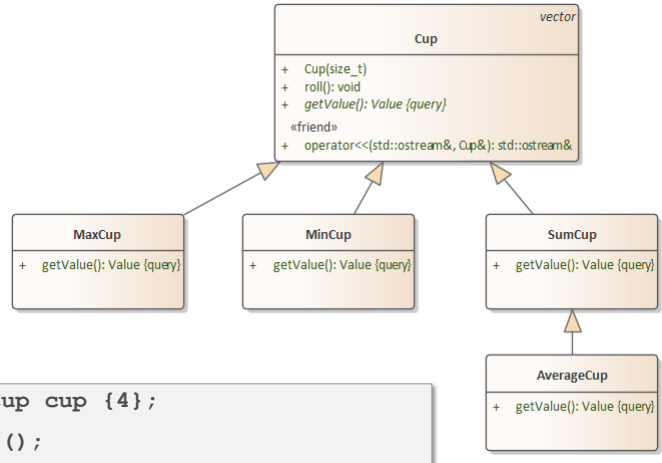



```
class A {
public:
  void processA(){}
};
class AA : protected A {
public:
  void processAA(){this->processA();}
};
class AAA : public AA {
public:
  void processAAA(){this->processA();}
};
```

compile

compile

Atelier



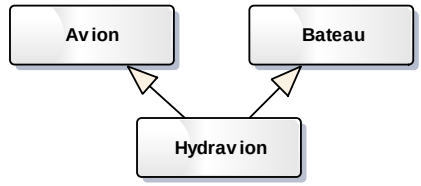

```
AverageCup cup {4};
cup.roll();
cout << cup << "\n";  // ( 3 2 5 4 )
cout << cup.getValue() << "\n";
```


15

Héritage multiple

- Fournit l'union des interfaces des classes de base

```
class Hydravion: public Avion, public Bateau {
 public :
   Hydravion () : Avion{}, Bateau{}{}
   // …
};
```



16

Héritage multiple et ambigüité

```
struct Avion {
  void avancer(){ cout << "Avion\n"; }
};

struct Bateau {
  void avancer(){ cout << "Bateau\n"; }
};

struct Hydravion : Avion, Bateau {};

void démarrer(Avion& a, Bateau& b, Hydravion& h){
  a.avancer();
  b.avancer();
  //h.avancer();
  h.Avion::avancer();
  h.Bateau::avancer();
}
```

le compilateur ne détecte pas l'ambigüité

ne compile pas

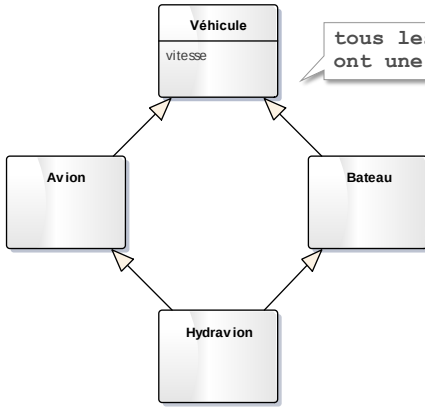
```
int main(){
  Hydravion h;
  démarrer(h, h, h);
}
```

```
struct Hydravion : Avion, Bateau {
 void avancer(){
  // …
 }
};
```

résolution

17

Héritage en diamant



tous les véhicules ont une vitesse

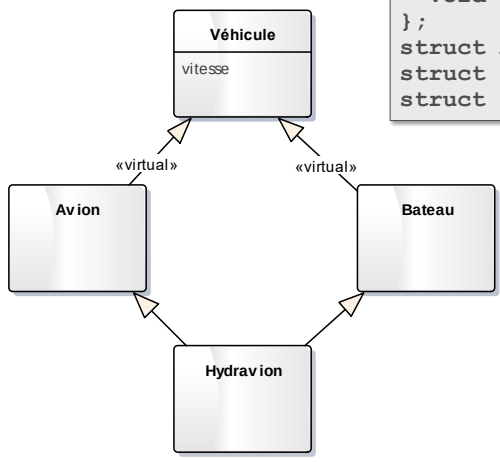
```
struct Véhicule {
  int vitesse;
  void avancer(){/*... */}
};
struct Avion : Véhicule{};
struct Bateau : Véhicule{};
struct Hydravion:Avion,Bateau{};
```

```
Hydravion h;
cout << sizeof h << "\n";
h.avancer();
```

?

18

Héritage virtuel




```
struct Véhicule {
  int vitesse;
  void avancer(){/*... */}
};
struct Avion : virtual Véhicule{};
struct Bateau : virtual Véhicule{};
struct Hydravion : Avion, Bateau{};
```

```
Hydravion h;
cout << sizeof h << "\n";
h.avancer();
```

?

19

Héritage virtuel et construction

```
struct A {
  int n;
  A(int n) : n{n}{cout << "A::ctor" << endl;}
  int getValue() const { return n;}
};

struct B : virtual A{
  B() : A{5}{ cout << "B::ctor" << endl; }
};

struct C : virtual A{
  int valeur;
  C(int a) : A{10}, valeur{a}{ cout << "C::ctor" << endl; }
};

struct D : B, C {
  D(int n, int p) : A{n}, C(p){ cout << "D::ctor" << endl; }
};
```

```
D d {3, 4};
cout << d.getValue() << endl;
B b;
cout << b.getValue() << endl;
```

?

20

Dominance et accès aux membres

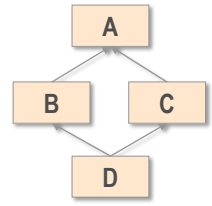
```
struct A {
  int value {1};
};

struct B : virtual A {
  int value {2};
};

struct C : virtual A {
};

struct D : B, C {};
```

compile ?



compile ?

```
D d;
cout << d.value << "\n";
cout << d.C::value << "\n";
```

?

Dominance et abstraction

```
struct A {
  virtual void f() = 0;
  virtual void g() = 0;
};

struct B : virtual A {
  void f() override {};
};

struct C : virtual A {
  void g() override {}
};

struct D: B, C {
};
```

```
int main(){
  D d;
}
```

compile
D est concrète

Downcasting et héritage virtuel

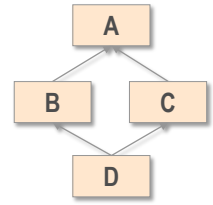
```
class A {public: virtual ~A(){}};
class B : public virtual A {};
class C : public virtual A {};
class D : public B, public C {};

int main(){
 D d;
 A* pBase {&d};
 D* pDerive {static_cast<D*>(pBase)};

 D* pDerive {dynamic_cast<D*>(pBase)};
}
```

pas de conversion de 'A*' vers 'D*';
car une classe de base virtuelle
est impliquée

OK



23