Rappels divers

Référence :

_ éviter de se servir des ptr.

_ faire des contructeurs de copie.

```
int &u;
```
c'est faux, la référence ne référence rien.

```
int i;
int &u = i;
```
ok, u est une référence sur i.

```
int i=5;
int &u= i;
u = 7;
```
↳ maintenant, i = 7

Une référence sert à désigner directement qqch que l'on avait déjà.
Il n'est pas possible de modifier ce sur quoi pointe une référence

Const :

```
const double pi=....;
~~double &rpi=pi;~~
const double & rpi=pi;
```
Pas possible car le type est const double..

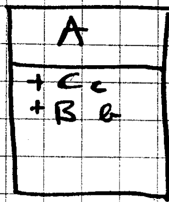
```
const int i = 3;
int *j = &i;
```
Pas possible : le type est (const int), pas int

```
const int *pi = &i;
const << (*pi);      OK
*pi = 5 ;            KO : left value ne peut ê modifié.
pi ++;               OK : on incremente le ptr.
```

```
const int const *pi= &t;
```
`const *` : ptr constant ; l'ensemble : sur un entier constant.

```
const A a;
a.foo(5);
```
KO : si le prototype est void A:: foo(int a);
OK : ——————————— void A:: foo(int a) const;

Constructeurs :

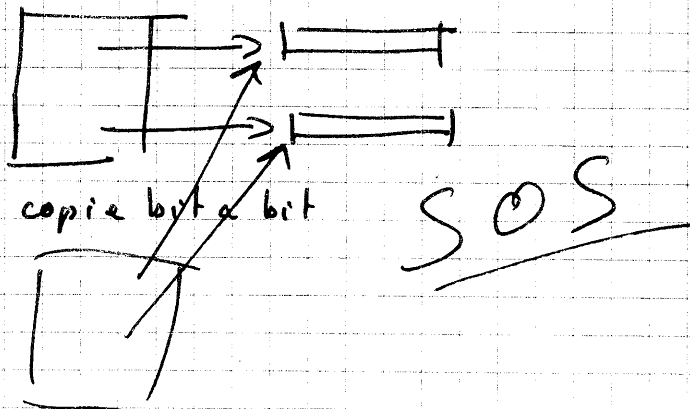


Avant de construire A, le compilo construit B automatiquement en récupérant le constructeur associé. Si on avait voulu construire B avec un param, il aurai falu faire :

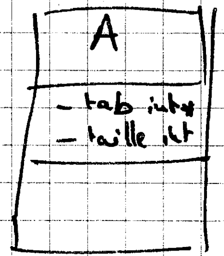
```
A::A(int i): b(5), c(i)
{
  :::
}
```

Constructeur de copie :

par défaut, le constructeur de copie est un constructeur bit a bit. C'est catastrophique quand on a des malloc par ex, car à la destruct° ... cata !



D'ou l'utilité du constructeur de copie

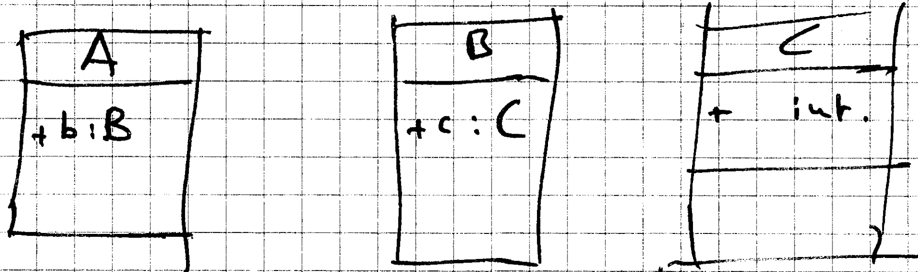



```
A::A(const A& a)
{ tab = new int[a.taille];
  for(int i=0; i < taille; i++)
     tab[i] = a.tab[i];
}
```

```
int main()
{
  A *pa;
  pa = new A[10];
  delete [] pa;
}
```

```
A::~A()
{
  delete [] tab;
}.
```

Destruction :



Destruct° de A ... le compila détruit automatiquem^t C puis B puis A

renvoyer une valeur.

int foo(void)
↑ la valeur de retour est placée ds la pile.

ex :

x = tab.crochet(1); OK
tab.crochet(1) = x; KO.

La solut° : les références :
si tab.crochet() renvoie une ref, ça marche !

Virtual : ds le .h

On met un virtual sur les méthodes pouvant ê redéfinies par les héritiers.

```
[A]  virtual m()          A  a, *pa;
 ↑                        B  b, *pb;
[B]  m()
                          a.m();
                          b.m();

                          pa = new A;
                          pa -> m()

                          pb = new B
                          pb -> m()

                          pb = new A;  NON
                          pa = new B;  OUI
                          pa -> m();   l'appel de la méthod de A
```

Au moment de la compil, l'affectat° de la f° est faite.
Si on a m() de la classe A qui est virtuelle, la décision se fait au moment de l'exécut° → la bonne méthode est appelée.

2

On met virtual qd la classe va ê dérivée.

Virtuel pur...

```
class Doc {
  public
    virtual void afficher() = 0;
    virtual void m() {....};
};
```

```
Doc *p;
p = new Doc;
```
KO classe abstraite car affiche non défini.

```
Doc *p;
p= new livre
p-> afficher
```
OK si afficher est déf ds livre.

Static :

l'atribut commun à ttes les instances:

ex : compter le nb d'instances d'une classe.

```
class cpt {
    public
        cpt();
static int getcpt();
    private
        static int nb;
}
```

```
cpt::cpt()
{ nb++;
}

int cpt::getcpt()
{ return nb;
}

int cpt::nb=0;
```

Surcharge d'opérateurs

operator ==(a, b) — permet de faire a == b.

exemple :

fonction — doit ê déclarée amie.

```
Cplx operator+(const Cplx &a, const Cplx &b)
{
    Cplx t;
    t.r = a.r + b.r ;
    t.i = a.i + b.i ;
    return t;
}
```

Méthode

```
Cplx   Cplx::operator+=(const Cplx &aj)
{
    i += aj.i ;
    r += aj.r ;
    return *this;
}
```

$t_1 += t_2$

Les flux :

cout << etud << endl

- cout → ostream&
- etud → Etudiant
- endl → char

```
ostream & operator << (ostream &os, const Etudiant &et)
{
    os << et.nom << ' '
       << et.prenom << endl ;
    return os;
}
```

const Etudiant &et → équivalent à Etudiant et mais évite la copie

flux d'entrée:

```
float f;
cin >> etud >> f;
istream& operator >> (istream &is, const Etudiant &et)
{ is >> et.nom;
  is >> et.prenom;
  return is
}
```

conversion de type) Dans A : B A::B() const

B A::operator B() const

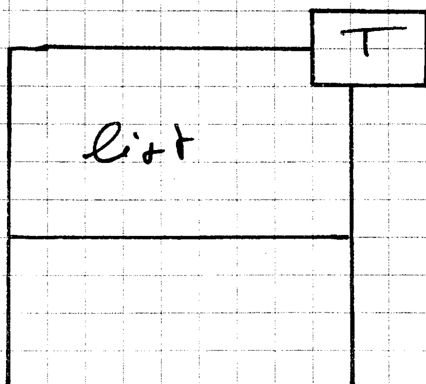
fichier de sortie:

```
{ ofstream out("toto", ios::out);

out << 5 << "bonjour";

} ← fermeture du fichier à la destruct°.
```

Les templates:




```
template <class T >
class List
{
    class cell
    { cell * next;
      T val
    }
    ...
  m();
}
List<T>::m()
{
}...
```

```
template <class T>

T max<T> (const T& e1, const T& e2)
{
    return (e1>e2)?e1:e2;
}
```

Déclaration:

```
int max <int>(int, int);
```

Equivalent de #define.

dans le .h : `extern const double pi;`

.c : `const double pi = 3.14;`

Les inline:

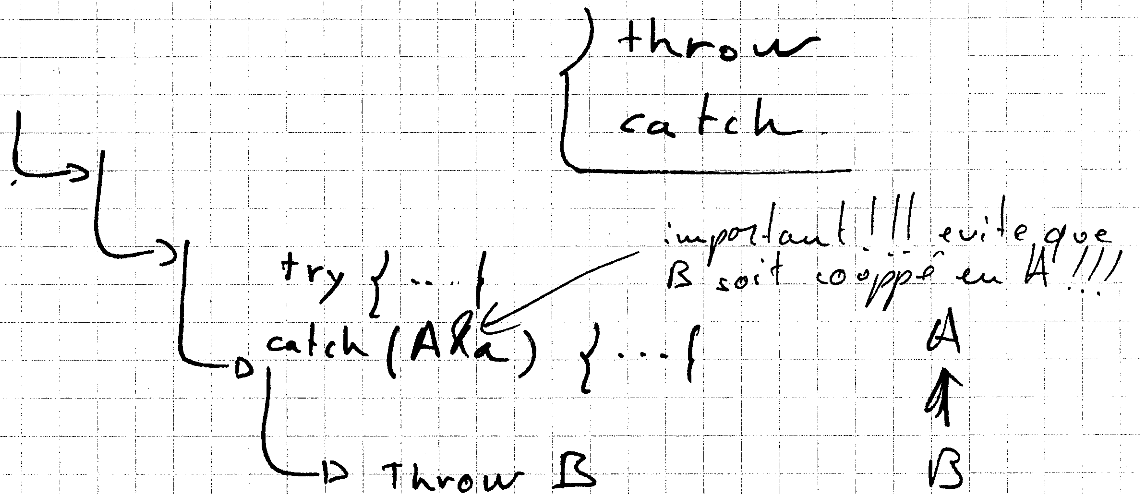
```
# define    MAX((a),(b)) {(a>b)? a : b}
```

En plus, ça bug ... `Max(i++, a);`

```
template<class T>
inline const T& max(const T&a, const T&b)
```

③

Exceptions :




```
catch(...) { //attrappe tout ! }.

catch (A) { throw; }
```

↑ relance l'exception.

```
for (flag = 1; flag ;)
{
    try { enregistrement = lit_enreg(); }
    catch (findufichier) { flag = 0; }
    ...
}

void foo() throws (A, B)
{
    ...
}
```

~~throws (A, B)~~ ← pas obligatoire mais plus joli...

```
A::A()
{
        tab = new int[50];
        if(...)  { delete [] tab;
        throw Erreur }
}
```

```
old.set_unexpected(myFonction); //q/except° lancée mais pas de throw ds prototype

old2.set_terminate(myFonction2); //avant de quitter.
```

```
char *s.

f(s) //nécessité d'un constructeur à partir d'un char     //f(String str){...}.

        String(char *s)
```

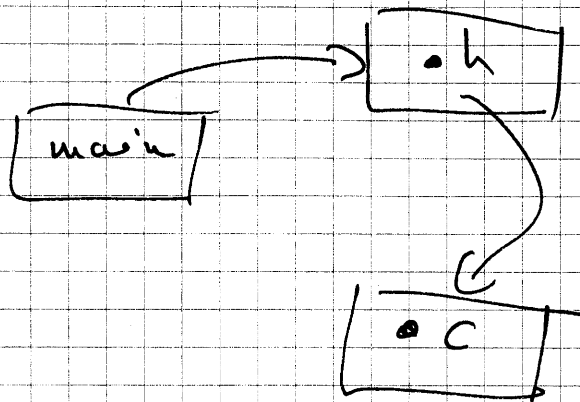
Autre solut°:

```
f :: operator B() const;
{...}.

appel par -> B(a);
          -> int(5.1);
```

Template 2 :

Pour le linkage, il faut



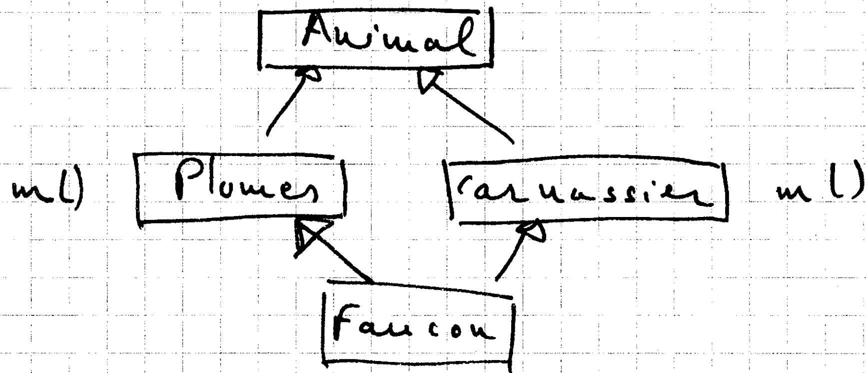
Autre solut° pour g++ :

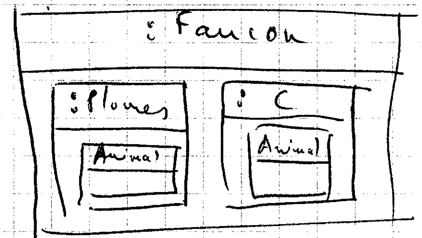
-f no_implicit_template

#pragma implementation

#template

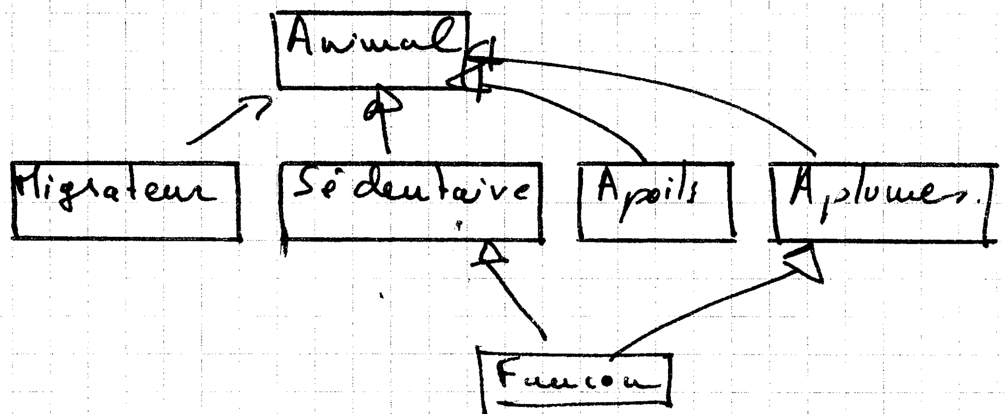
Héritage multiple.



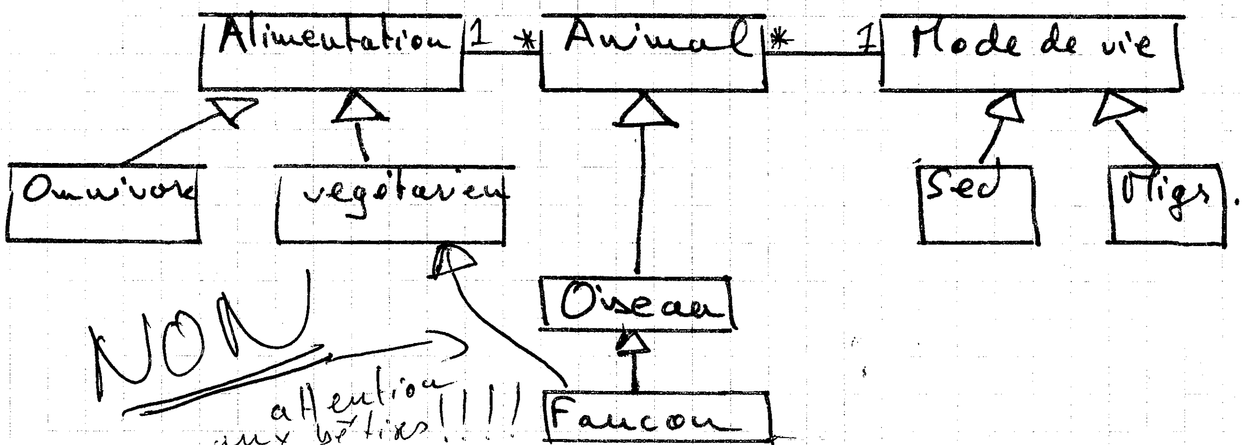


res 1 = plume :: m() ;
res 2 = carnassier :: m() ;
merge (res 1, res 2);

⚠ côté trop statique des choses :

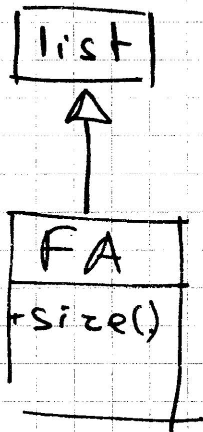


Autre représentation dynamique



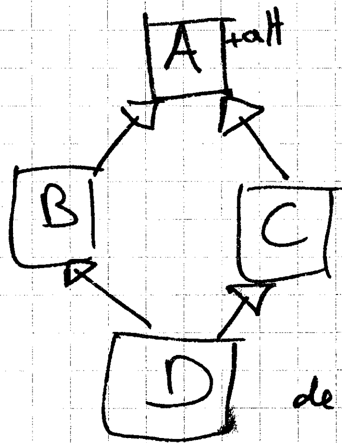
4

Lever des ambiguités en mettant le nom des classes



file attente a une méthode size, définie dans list.
pour y accéder, on fait
list :: size().

Revenons aux héritages multiples :

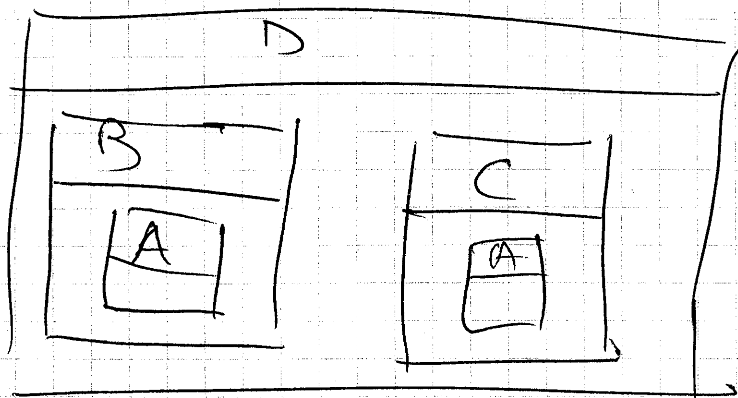


de D, accès à l'attribut de A :
B :: A :: att ;

Attention :

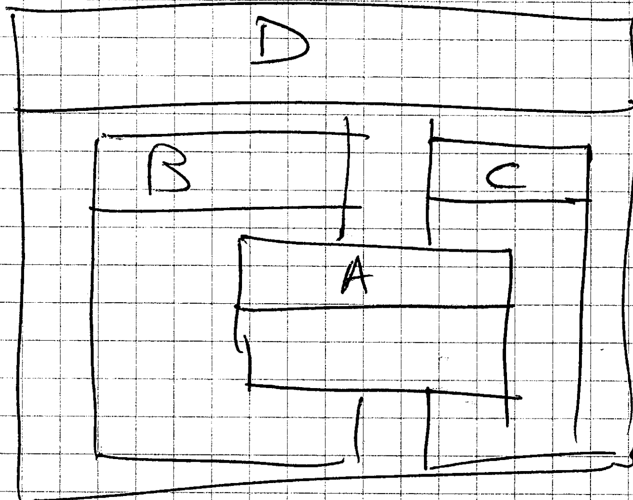
B :: A :: att ≠ C :: A :: att

car :



Pour feinter :

B : public virtual A { ... }
C : public virtual A { ... }



Les cast

Les cast en C sont brutaux et effectué à la compil.

Les static cast :

```
int i = static_cast <int> d
```

→ aussi brutal que C.

reinterpret_cast :

Convertir ptr vers ptr autre ou vers entier.

```
ArcQT *arc = reinterpret_cast<ArcQT*> arc2base;
```

const_cast :

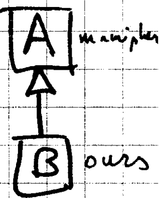
Pr ajouter ou supprimer un const.

cast dynamique : dynamic_cast :

```
T* pt = dynamique_cast <T*> (p)
```

Retourne 0 en cas de problème.

exemple :




```
A *a;
B *b;
a = new B;  // OK : un B est un A.
~~b = a;~~  // NON !!!
b = dynamic_cast <B*> a;
if (!b) throw ...
```

⚠ pr que le dynamic_cast marche, il faut un virtual dans classe.

Type id

```
#include <type_info>
```

J'ai deux ptr... de quels types st ils ?

```
int *a, *b;
if (typeid(a) == type id(b))
    (...);
```

∃ un classement. Utilisation de <u>before</u>.

type_info::name() ⟹ nommage des objets
⟹ ⚠ dépend du compilo.

Mutable.

Dans un objet constant, l'attribut mutable peut ē modifié.

```
class A {
    public:
        A();
        mutable int i;
};
```

Inline:

Parfois, un appel de fonction est plus cher que le corps de la fⁿ elle m̄.

↪ inline remplace l'appel par le corps.

Si le corps de la fonction est dans le point h, vue fⁿ est inline ⟹ ne pas le faire : c'est sale.

Sinon mettre le mot clé devant le nom de la fonction : inline int getA() const;

Register / Volatile.

register int i : affecte un registre à i.
volatile int i : l'inverse ex recept° série.

Optimisat°:

gcc -O2 optimise. ∃ aussi -O1, -O2, -O3 ...

⚠ ça va pas forcément + vite !!