LANGAGE OBJ. AV.(C++)
MASTER 1

Yan Jurski

U.F.R. d’Informatique
Université de Paris Cité

RETOUR SUR LES CLASSES

(Un peu plus exhaustivement que les introductions pratiques qui ont été vues en cours 1 et 2)

La **déclaration** (dans Compte.hpp) du concept élémentaire de compte en banque pourrait être :

```
class Compte {
 int solde;
public:
 int getSolde();
 void deposer(int somme);
 void retirer(int somme);
};
```

Elle **suffit** à imaginer une utilisation :

```
#include "Compte.hpp"
...
Compte monCompte;
cout << “Reste “ << monCompte.getSolde() << endl;
monCompte.deposer(100);
monCompte.retirer(23);
```

La **définition** de la classe (dans Compte.cpp)

```
void Compte::deposer(int s) { solde += s; }
void Compte::retirer(int s) { solde -= s; }
int Compte::getSolde() { return solde; }
```

se fait avec l’opérateur de portée ::

Attention, si vous oubliez la portée et écrivez dans Compte.cpp

```
void deposer(int s) { ...qq chose ... }
```

Vous définirez alors une fonction globale, et vous ferez une erreur. Ici, comme {...qq chose...} utilisera l’attribut solde, le compilateur préviendra heureusement d'une erreur

Convention :

la **déclaration** d’une classe se fait dans un "fichier d’entête" MaClasse.hpp

la **définition** des méthodes de la classe se fait dans un fichier "MaClasse.cpp"

l’**utilisation** de la classe nécessite #include "MaClasse.hpp"

La **compilation** séparée se fait via le couple make/Makefile (voir cours 1)

l’encapsulation c’est avoir :

- des constituants agrégés

- des moyens d'actions regroupés

- un mécanisme permettant de cacher certains aspects

On distingue les domaines :
private, public, protected, default

La protection se fait durant la déclaration.

```
class A {
    déclarations
  domaine:
    déclarations
  domaine:
    déclarations
};
```

Un même domaine peut apparaître plusieurs fois

Le domaine `private` :

ses éléments ne sont visibles que par des fonctions membres de la classe.

```
class A {
    private:
      int attribut;
      void methode1();
    public:
      void methode2();
};
```

```
void A::methode1() {
    attribut = 1; // ok
}
void A::methode2() {
    methode1();      // ok
    attribut = 2; // ok
}
int main() {
    A a;
    a.attribut;    // non
    a.methode1(); // non
    a.methode2(); // oui
}
```

Le domaine `public` :

visible, accessible partout

Le domaine `protected` :

est un intermédiaire entre private et public.
La visibilité est conservée par héritage.

Le domaine par défaut :

c'est `private` (un peu différent de java donc)

Les **constructeurs** sont des méthodes qui :

- portent le nom de la classe,
- ne précisent pas de type retour
- sont appelées lors de la création de l'objet

```
class Compte {
 private:
  int solde;
 public:
  Compte(int x); // Un constructeur
};
```

```
Compte::Compte(int x):solde{x} {  cout << "bienvenue" << endl; }
```

Notez bien la séquence d'initialisation avant le bloc

```
int main() {
 Compte c1{1000};
}
```

Les **constructeurs** sont des méthodes qui :

- portent le nom de la classe,
- ne précisent pas de type retour
- sont appelées lors de la création de l'objet

```
class Compte {
 private:
  int solde;
 public:
  Compte(int x); // Un constructeur
  Compte();
};
```

```
Compte::Compte(int x):solde{x} {  cout << "bienvenue" << endl; }
Compte::Compte():Compte{0}{}
```

Notez comment un constructeur peut en appeler un autre

```
int main() {
 Compte c1{1000}, c2, c3{}; // mais ne pas écrire c4() !
}
```

Le **concepteur** doit adopter une démarche centrée sur l'utilisateur :

Qu’est ce que l’utilisateur peut raisonnablement fournir comme données pour initialiser un objet ?

et non pas imposer que lui soit fournies exhaustivement toutes les données internes à l'objet...

Avec :

```
class Compte {
private:
 int solde;
public:
 Compte(int x); // Un constructeur
};
```

comprenez bien la différence entre :

```
Compte::Compte(int x):solde{x} {
    cout << "bienvenue" << endl;
 }
```

et :

```
Compte::Compte(int x) {
    solde =x;
    cout << "bienvenue" << endl;
 }
```

L’ordre des opérations effectuées lors de la création d'un objet par la machine :

- allocation de la mémoire (pour l'objet et ses champs)

- puis initialisation de toutes les données membres dans l’ordre de leur déclaration :

soit explicitement grâce à la séquence d'initialisation, soit par un constructeur par défaut.

- puis exécution du code du bloc du constructeur

Avec :

```
class Compte {
private:
 int solde;
public:
 Compte(int x); // Un constructeur
};
```

comprenez bien la différence entre :

```
Compte::Compte(int x):solde{x} {
    cout << "bienvenue" << endl;
 }
```

et :

```
Compte::Compte(int x) {
    solde =x;
    cout << "bienvenue" << endl;
 }
```

avec cette seconde version il y a eu 2 écritures dans solde ...

Mise en situation ...
sans séquence d'initialisation ...

```
class Point {
    private:
      int abs, ord;
    public:
      Point(int x,int y);
};
```

```
class Segment{
    private:
      Point premier, second;
    public:
      Segment(int x1,int y1,
              int x2,int y2);
};
```

```
Point::Point(int x,int y) { abscisse = x; ordonnee = y; }
Segment::Segment(int x1,int y1,int x2,int y2){
  // les points premier et second sont censés être construits
  // mais comment ?
  // de plus leurs abs, ord seraient inaccessibles (private)
  // comment pourrait-on leur affecter x1,y1,x2,y2 ? ...
 }
```

Mise en situation ...
avec séquence d'initialisation
(les pbs sont réglés)

```
class Point {
   private:
     int abs, ord;
   public:
     Point(int x,int y);
};
```

```
class Segment{
   private:
     Point premier, second;
   public:
     Segment(int x1,int y1,
             int x2,int y2);
};
```

```
Segment::Segment(int x1,int y1,int x2,int y2)
          : premier{x1,y1}, second{x2,y2} {}
```

et bien entendu : `Point::Point(int x,int y): abs{x},ord{y} {}`

(même si l'écriture précédente n'était pas bloquante, elle n'était pas bien maîtrisée)

Retenez bien que vous serez "critiquables" si vous écrivez (sans liste d'initialisation) :

```
Compte::Compte(int x) {
    solde =x;  // Non !
 }
```

```
Compte::Compte(int x) : solde {x} { } // Oui !
```

Objets et pointeurs

```
class Point {
    private:
      int abs, ord;
    public:
      Point(int x,int y);
};
```

```
int main(){
   Point *p;
   return EXIT_SUCCESS;
};
```

Ici :

- aucun objet Point n'est construit
- la variable p manipule des adresses

Objets et pointeurs

```
class Point {
    private:
      int abs, ord;
    public:
      Point(int x,int y);
};
```

```
int main(){
   Point *p{nullptr}, q{1,2};
   return EXIT_SUCCESS;
};
```

p est initialisé à nullptr,
q est construit

Objets et pointeurs

```
class Point {
    private:
      int abs, ord;
    public:
      Point(int x,int y);
};
```

```
int main(){
   Point q{1,2}, *p{&q};
   return EXIT_SUCCESS;
};
```

q est construit
p est initialisé avec l'adresse de q

Objets et pointeurs

```
class Point {
    private:
      int abs, ord;
    public:
      Point(int x,int y);
};
```

```
int main(){
  Point q{1,2}, *p{&q};
  p=new Point{3,4};
  delete p;
  return EXIT_SUCCESS;
};
```

- new permet d'invoquer un constructeur
- l'adresse de l'objet créé est retournée à p
- *p est de type Point
- (*p).methode() est autorisé
- sémantiquement équivalent à : p-> methode()
  (syntaxiquement les opérateurs différents ...)

Un objet peut parler de lui-même en utilisant le mot-clé `this`

son type est "pointeur vers le type de l'objet"

`this` permet en particulier :

- de lever certaines *ambiguïtés*

- de transmettre l'objet actif en argument d'une autre méthode ou à un return

exemple de levée d’ambiguïté :

```
class A {
  private:
      int valeur;
    public:
      void setValeur(int valeur);
};
```

```
void A::setValeur(int valeur) {
   this->valeur = valeur;
}
```

exemple de besoin d'auto-nommage :

```
class A {
  private:
     int valeur;
    public:
      A min(A other);
};
```

```
A A::min(A other) {
   if (other.valeur < valeur) return other;
   else return ???  // l'objet courant
}
```

exemple de besoin d'auto-nommage :

```
class A {
  private:
     int valeur;
    public:
      A min(A other);
};
```

```
A A::min(A other) {
   if (other.valeur < valeur) return other;
   else return *this;  // pour respecter le type retour
}
```

exemple de besoin d'auto-nommage :

```
class A {
  private:
     int valeur;
    public:
      A min(A other);
};
```

```
A A::min(A other) {
   if (other.valeur < valeur) return other;
   else return *this;  // pour respecter le type retour
}
```

Spoiler : ici on a choisi A comme type retour pour min. On aurait pu choisir A* .
Nous en reparlerons lorsque nous aborderons le constructeur de copie...

Complément sur const ...

Déjà évoqué rapidement comme modificateur de type : il restreint les usages d’une variable afin de ne pas pouvoir la modifier.

const peut également être adossée à la spécification d'une méthode : la méthode s'engage alors à laisser l'objet qui l'exécute inchangé
(const s'applique alors aux attributs de l'objet : ce sont eux qui seront non modifiables).

```
class A {
private :
   int att;
public:
 A(int x);
 int get() const;
 void affiche() const;
 void set(int v);
};
```

Les méthodes get() et affiche() laisseront invariant l'objet qui les exécute

La contrainte const peut être jugée trop forte.

On veut pouvoir modifier certains attributs techniques alors même que du point de vue d'un observateur extérieur l'objet sera tout de même considéré globalement comme étant constant...

Exemple : un compte en banque dont on imagine que les transactions sont toutes tracées y compris les consultations...
Un utilisateur peut considérer qu'après un get() le compte est grosso-modo inchangé

```
class Compte {
private:
 int solde;
 int nbTransactions;
public:
 int getSolde() const {
    nbTransactions++;    // interdit par le const
    return solde;
 }
};
```

```
class Compte {
private:
 int solde;
 mutable int nbTransactions;
public:
 int getSolde() const {
    nbTransactions++;    // autorisé malgré tout !
    return solde;
 }
};
```

On a vu dans cette première partie essentiellement des choses auxquelles ceux qui viennent de notre licence s'attendaient :

- définition des classes (private etc ..)
- construction (à la c++)
- pointeurs et objets :
  x->m() ou (*x).m() ; new + constructeur ; delete
- this
- const pour les méthodes / objets
- exception à const : mutable

Place aux nouveautés maintenant :
destructions, autres copies, références

Destruction

Durée de vie des objets

C++ propose un moyen de réaliser des opérations à la fin de vie d'un objet :

- les méthodes **destructeur**

Chaque classe en possède une.

Le destructeur est l'alter-ego des constructeurs...

Un **destructeur** est une méthode :

- dont le nom est celui de la classe préfixée par le caractère ~
- elle ne renvoie rien,
- elle ne prend aucun paramètre,
- il n'y a qu'un seul destructeur par classe
- il faut le qualifier virtual (on verra plus tard pourquoi)
- il est appelé automatiquement à la destruction d’un objet.

Une **destruction** est :

- implicite/automatique lorsque le point de contrôle du programme quitte le bloc de la déclaration de l'objet
- explicite (via delete) lorsqu’on décide de détruire un objet alloué dynamiquement (un pointeur obtenu par new).

Le destructeur est appelé avant de libérer la mémoire.

Exemple 1 :

```
class A {
public:
 char name;
 A(char);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
   cout << "Naissance de "<< name;
}
A::~A() { cout << "Mort de " << name;}
```

```
void f(){
     cout << "dans f";
     A x{'x'};
}
int main() {
     A a{'a'},b{'b'};
     f();
     f();
     return EXIT_SUCCESS;
}
```

```
Naissance de a
Naissance de b
dans f
Naissance de x
Mort de x
dans f
Naissance de x
Mort de x
Mort de b
Mort de a
```

Rq : autant de mort que de naissance !

Exemple 2 :

```
class A {
public:
 char name;
 A(char);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
   cout << "Naissance de "<< name;
}
A::~A() { cout << "Mort de " << name;}
```

```
void g(A y) {
 cout << "dans g";
}

int main() {
    A a{'a'};
    g(a);
    return EXIT_SUCCESS;
}
```

```
Naissance de a
dans g
Mort de a
Mort de a
```

ici, on dirait que "a" meurt 2 fois ...

Exemple 2 :

```
class A {
public:
 char name;
 A(char);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
   cout << "Naissance de "<< name;
}
A::~A() { cout << "Mort de " << name;}
```

```
void g(A y) {
 cout << "dans g";
}

int main() {
    A a{'a'};
    g(a);
    return EXIT_SUCCESS;
}
```

```
Naissance de a
dans g
Mort de a
Mort de a
```

on comprend que c'est la trace de la mort de y
... c'est un peu plus clair ...
mais il manque quand même la trace d'une naissance ...

Explication :
visiblement la déclaration/initialisation des paramètres au moment de l'appel de fonction ne fait pas appel au constructeur que nous avons écrit (sinon on aurait une "naissance")

On comprend, en remarquant que la mort du paramètre est visible, que c++ a opéré une sorte de clonage en recopiant argument pour argument l'objet transmis.

Ce mécanisme est appelé **construction par copie** (d'un original).

Cette construction par copie existe par défaut, mais on peut aussi le redéfinir explicitement (et donc ici pouvoir témoigner de la "naissance" qu'on veut voir apparaître)

Présentation du constructeur de copie

```
class A {
public:
 char name;
 A(char);
 A (const A&);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
   cout << "Naissance de "<< name;}
A::A (const A & x):name{x.name} {
   cout << "Naissance d'une copie de "<<
name;}
A::~A() { cout << "Mort de " << name;}
```

```
void g(A y) {
 cout << "dans g";
}

int main() {
    A a{'a'};
    g(a);
    return EXIT_SUCCESS;
}
```

```
Naissance de a
Naissance d'une copie de a
dans g
Mort de a
Mort de a
```

ici tout devient cohérent

Focus sur le constructeur de copie qu'on vient d'écrire :

```
class A {
public:
 A (const A&);
...
};
```

```
...
A::A (const A & x):name{x.name} {
  cout << "Naissance d'une copie de
"<< name;}
```

remarques :

Rq 1 : à la copie on s’est chargé, dans la séquence d'initialisation, de faire se correspondre les attributs (on aurait pu faire autrement)

Rq 2 : on a déclaré le constructeur de copie public (on aurait pu faire autrement)

Rq 3 : la copie déclare laisser son argument constant (on aurait pu faire autrement)

Rq 4 : le symbole & est utilisé dans l'argument (à justifier ...)

Rq 1 : à la copie on s'est chargé, dans la séquence d'initialisation, de faire se correspondre les attributs
(on aurait pu faire autrement)

```
class A {
public:
 string name;
 A(char);
 A (const A&);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name;}
A::A (const A & x):name{x.name+"_copie"} {
  cout << "Naissance d'une copie de "<<
x.name;}
A::~A() { cout << "Mort de " << name;}
```

```
void g(A y) {
 cout << "dans g";
}

int main() {
    A a{'a'};
    g(a);
    return EXIT_SUCCESS;
}
```

```
Naissance de a
Naissance d'une copie de a
dans g
Mort de a_copie
Mort de a
```

Rq 2 : on a déclaré le constructeur de copie public
(on aurait pu faire autrement)
cela reviendrait à "interdire" de passer un objet en argument d'une fonction ...

```
class A {
public:
 char name;
 A(char);
 virtual ~A();
private :
 A (const A&);
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name;}
A::A (const A & x):name{x.name} {
  cout << "Naissance d'une copie de "<<
name;}
A::~A() { cout << "Mort de " << name;}
```

```
void g(A y) {
 cout << "dans g";
}

int main() {
    A a{'a'};
    g(a);
    return EXIT_SUCCESS;
}
```

```
main.cpp: error:
‘A::A(const A&)’ is private
within this context
     g(a);
         ^
```

Rq 2 : on a déclaré le constructeur de copie public
(on aurait pu faire autrement)
cela reviendrait à "interdire" de passer un objet en argument d'une fonction ...

```
class A {
public:
 char name;
 A(char);
 virtual ~A();
private :
 A (const A&);
 h (A x);
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name;}
A::A (const A & x):name{x.name} {
  cout << "Naissance d'une copie de "<<
name;}
A::~A() { cout << "Mort de " << name;}
```

"L'interdiction" n'est pas interne : le constructeur (private) peut être naturellement utilisé dans la classe A.

Ce serait le cas avec la méthode h (A x) ici.

Rq 3 : la copie déclare laisser son argument constant (on aurait pu faire autrement)

```
class A {
public:
 string name;
 A(char);
 A ( A&);// pas const
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
   cout << "Naissance de "<< name;}
A::A ( A & x):name{x.name+"_copie"} {
   cout << "Naissance d'une copie de "<<
x.name;
   x.name='*';
}
A::~A() { cout << "Mort de " << name;}
```

```
void g(A y) {
 cout << "dans g" << endl;
}

int main() {
    A a{'a'};
    g(a);
    return EXIT_SUCCESS;
}
```

```
Naissance de a
Naissance d'une copie de a
dans g
Mort de a_copie
Mort de *
```

Que se passe t'il lors du retour d'une fonction ? (1)

```
class A {
public:
 string name;
 A(char);
 A(const A & x);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name;
}
A::A ( const A & x):name{x.name+"_copie"} {
  cout << "Naissance d'une copie de "<< x.name;
}
A::~A() { cout << "Mort de " << name;}
```

```
A g(A x) {
   return x;
}

int main() {
    A a{'a'};
    cout << g(a).name;
    return EXIT_SUCCESS;
}
```

```
Naissance de a
Naissance d'une copie de a
Naissance d'une copie de a_copie
a_copie_copie
Mort de a_copie_copie
Mort de a_copie
Mort de a
```

L'objet retourné est une nouvelle construction.
Rq : la mort de a_copie intervient dans un ordre inattendu ... (explication un peu plus loin)

Que se passe t'il lors du retour d'une fonction ? (2)

```
class A {
public:
 string name;
 A(char);
 A(const A & x);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name;
}
A::A ( const A & x):name{x.name+"_copie"} {
  cout << "Naissance d'une copie de "<< x.name;
}
A::~A() { cout << "Mort de " << name;}
```

```
A g() {
   return {'x'};
}

int main() {
    cout << g().name;
    return EXIT_SUCCESS;
}
```

```
Naissance de x
x
Mort de x
```

Le type retourné par g() attend un A à construire.

Le return lui fourni un argument correspondant à un constructeur existant.

Il est invoqué.

Que se passe t'il lors du retour d'une fonction ? (3)

```
class A {
public:
 string name;
 A(char);
 A(const A & x);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name;
}
A::A ( const A & x):name{x.name+"_copie"} {
  cout << "Naissance d'une copie de "<< x.name;
}
A::~A() { cout << "Mort de " << name;}
```

```
A h() {
 cout << "dans h";
 A z{'z'};
 return z;
}

int main() {
    cout << "affichage en retour de
"           << h().name;
    return EXIT_SUCCESS;
}
```

```
dans h
Naissance de z
affichage en retour de
z
Mort de z
```

Voilà qui est étrange, car le retour "devrait" produire une copie

Que comprendre ?
soit le A local à h() n'est pas détruit (?)
soit la copie retournée n'est pas faite (?)

Réponse : attention aux optimisations du compilateur

"RVO Return Value Optimization"

dans certains cas le compilateur essaie de minimiser la construction d’objets...

Pour supprimer cette optimisation il faut utiliser (avec gcc) l’option `-fno-elide-constructors`

Avec l'option -fno-elide-constructors :

```
class A {
public:
 string name;
 A(char);
 A ( const A &
x);
 virtual ~A();
};
```

```
#include "A.hpp"
#include <iostream>
using namespace std;
A::A(char n) : name{n} {
  cout << "Naissance de "<< name;
}
A::A ( const A & x):name{x.name+"_copie"} {
  cout << "Naissance d'une copie de "<< x.name;
}
A::~A() { cout << "Mort de " << name;}
```

```
A h() {
 cout << "dans h";
 A z{'z'};
 return z;
}

int main() {
    cout << "affichage en retour de
"           << h().name;
    return EXIT_SUCCESS;
}
```

```
dans h
Naissance de z
Naissance d'une copie de z
Mort de z
z_copie
Mort de z_copie
```

plus cohérent ! Attention donc ...

A retenir :

- Le constructeur de copie est implicitement impliqué dans la transmission des objets en arguments, et dans le return
- Il a une définition par défaut qui copie membre à membre les attributs
- Sa signature habituellement retenue (pour une classe A) est :

```
public:
 A(const A&);
```

- Il est possible de le redéfinir largement.
(En conséquence la "copie" peut ne pas en être une !
Son utilisation "intuitive" peut s'en trouver totalement perturbée)

- Le compilateur court-circuite certaines constructions temporaires qu'il croit inutiles sur des objets anonymes (c. à d. non stockés dans une variable)

Le type référence :

Il nous reste à parler de ce symbole & dans

```
public:
 A(const A&);
```

Nous l’avons déjà rencontré au moment où nous avons parlé des pointeurs.

```
int a{3}, *pa{&a};
```

Il s'agissait d'obtenir l'adresse d'une variable

L'usage de & ici lors d’une déclaration de type est totalement différent :

nous allons parler de **transmission par référence**

(Commençons par justifier de son utilité)

Rappel :
À l’entrée d’une fonction recevant un argument :
- on créé une nouvelle variable du type considéré (ici A)
- son nom est celui du paramètre formel (ici y).
- elle est initialisée à l’aide de la valeur du paramètre effectif (ici **a** qui encapsule 'a') et nous venons de voir que cette initialisation est faite précisément par le constructeur de copie

```
void g(A y) {
 cout << "dans g";
}

int main() {
    A a{'a'};
    g(a);
    return EXIT_SUCCESS;
}
```

Si on imaginait une signature pour le constructeur de copie de la forme

```
class A {
public:
 A(A original); //incorrect
};
```

Sachant que le constructeur par copie est invoqué à chaque passage d'argument, et ... qu'il dispose lui même d'un argument ... il y aurait donc un pb récursif : à l'entrée du constructeur, "original" devrait être initialisé par copie, via le constructeur par copie etc ...

On comprend qu'on a besoin d'un autre mode de passage de paramètre, sans copie.

Le symbole & indique qu'on ne créé pas de nouvel objet pour cette variable. Il exprime simplement qu'on désigne le paramètre transmis à l’aide d’un nom local, c'est un alias.

L'argument transmis et cet alias désigneront le même objet.

```
class A {
public:
 A(A & original);
};
```

chapitre sur **le type référence** (&)

En C++, en plus des variables classiques, et des variables pointeurs (avec *), on peut définir des variables références (avec &) qui fonctionnent par aliasing.

cet exemple illustre que a et b se réfèrent au même objet.

On peut le vérifier en affichant leurs adresses

```
class A {
  public :
  int val;
  A(int x=0);
};

A::A(int x) : val{x} {}

int main() {
  A a;
  A &b {a};
  a.val++;
  cout << a.val; // 1
  cout << b.val; // 1
  b.val++;
  cout << a.val; // 2
  cout << b.val; // 2
  cout << &a; // 0xbfbee944
  cout << &b; // 0xbfbee944
  return EXIT_SUCCESS;
}
```

chapitre sur **le type référence** (&)

En toute logique, que devrait afficher ce code ?

```
class A {
  public :
  int val;
  A(int x=0);
};

A::A(int x) : val{x} {}

int main() {
   A a;
   A &b {a};  // une référence
   A c;       // un autre objet
   a.val++;
   b=c;  // quelle interprétation ?
   cout << c.val; // 0
   cout << a.val; // ?
   cout << &a; //
   cout << &c; //
   cout << &b; //
   return EXIT_SUCCESS;
}
```

chapitre sur **le type référence** (&)

En toute logique, que devrait afficher ce code ?

explication : b étant un alias de a, faire l'affection b=c ou a=c doit donner le même résultat.
C'est donc normal que 'a' prenne la valeur interne de 'c' qui est 0

Que dire des adresses ?

```
class A {
  public :
  int val;
  A(int x=0);
};

A::A(int x) : val{x} {}

int main() {
   A a;
   A &b {a};  // une référence
   A c;       // un autre objet
   a.val++;
   b=c;  // quelle interprétation ?
   cout << c.val; // 0
   cout << a.val; // 0
   cout << &a; //
   cout << &c; //
   cout << &b; //
   return EXIT_SUCCESS;
}
```

chapitre sur **le type référence** (&)

En toute logique, que devrait afficher ce code ?

explication : b étant un alias de a, faire l'affection b=c ou a=c doit donner le même résultat.
C'est donc normal que 'a' prenne la valeur interne de 'c' qui est 0

Que dire des adresses ?

'a' et 'c' sont des variables différentes

L'affectation b=c change t'elle l'aliasing ?

```
class A {
  public :
  int val;
  A(int x=0);
};

A::A(int x) : val{x} {}

int main() {
   A a;
   A &b {a};  // une référence
   A c;       // un autre objet
   a.val++;
   b=c;  // quelle interprétation ?
   cout << c.val; // 0
   cout << a.val; // 0
   cout << &a; // 0x...b0
   cout << &c; // 0x...b4
   cout << &b; //
   return EXIT_SUCCESS;
}
```

chapitre sur **le type référence** (&)

En toute logique, que devrait afficher ce code ?

```
class A {
  public :
  int val;
  A(int x=0);
};

A::A(int x) : val{x} {}

int main() {
   A a;
   A &b {a};  // une référence
   A c;       // un autre objet
   a.val++;
   b=c;  // quelle interprétation ?
   cout << c.val; // 0
   cout << a.val; // 0
   cout << &a; // 0x...b0
   cout << &c; // 0x...b4
   cout << &b; //0x...b0
   return EXIT_SUCCESS;
}
```

explication : b étant un alias de a, faire l'affection b=c ou a=c doit donner le même résultat.
C'est donc normal que 'a' prenne la valeur interne de 'c' qui est 0

Que dire des adresses ?

'a' et 'c' sont des variables différentes

L'affectation b=c change t'elle l'aliasing ?

non : l'aliasing est une chose définitive

chapitre sur **le type référence** (&)

```
int main() {
   A a;
   A *p {&a};  // pointeur
   A c;
   a.val++;
   *b=c;
   b= &c;
   return EXIT_SUCCESS;
}
```

```
class A {
  public :
  int val;
  A(int x=0);
};

A::A(int x) : val{x} {}

int main() {
   A a;
   A &b {a};  // une référence
   A c;       // un autre objet
   a.val++;
   b=c;  // quelle interprétation ?
   cout << c.val; // 0
   cout << a.val; // 0
   cout << &a; // 0x...b0
   cout << &c; // 0x...b4
   cout << &b; //0x...b0
   return EXIT_SUCCESS;
}
```

Si on avait souhaité autre chose, il aurait fallu utiliser des pointeurs

chapitre sur **le type référence** (&)

Un type référence est-il détruit comme les autres variables ?

chapitre sur **le type référence** (&)

Un type référence est-il détruit comme les autres variables ?

```
int main() {
   A a;
   { // nouveau bloc
    A &b{a};
    ...
   } // fin de bloc
   ...
   return EXIT_SUCCESS;
}
```

A la fin du bloc, on ne peut pas envisager que 'b' soit détruit par appel à un destructeur qui serait celui de 'a'. La seule chose acceptable est que le rôle joué par 'b' disparaisse, c.a.d que son nom soit simplement oublié.

Donc : non, il n'y a pas d'appel implicite à un destructeur pour les références. ("b" n'est ni construite, ni détruite)

chapitre sur **le type référence** (&)

On peut imaginer :

```
int main() {
   A a,b;
   a.val++; // pour les distinguer
   A * pa{&a};
   A * & pa_bis {pa};
   pa = &b ;
   cout << (pa_bis==pa);
   // ...
   return EXIT_SUCCESS;
}
```

pa_bis est une référence vers pa qui est un pointeur vers un A ...

chapitre sur **le type référence** (&)

On peut essayer ... :

```
int main() {
   A a,b;
   a.val ++; // pour les
distinguer
   A * pa{&a};
   A * & pa_bis {pa};
   pa = &b ;
   cout << (pa_bis==pa);
   A & * px{pa}; // mais ça non !
   return EXIT_SUCCESS;
}
```

non, cela n'aurait pas de sens car on chercherait à exprimer que px serait un pointeur, donc une adresse, vers une référence d'un objet A ... or rien ne distingue un A de sa référence.

Comme le type indiqué est strictement équivalent à A * le compilateur relève une erreur d'écriture

chapitre sur **le type référence** (&)

Y a t'il une notion de tableau de références ? Réfléchissons ...

```
int main() {
   A a,b;
   A c[] {a,b};
   A &d[] {a,b}; // .... bizarre
   return EXIT_SUCCESS;
}
```

Le tableau 'c' est construit en réservant 2 espaces pour des éléments de type A, et en les initialisant à partir de 'a' et de 'b' : ce sont deux **copies** qui sont faites ici.
La syntaxe pour 'd' est refusée. Le programmeur cherche vraisemblablement à éviter les copies.
Mais, il y aurait des incohérence à l'accepter. Rappelons que pour les tableaux :

- c == & (c[0]) ou de manière équivalente *c== c[0]
- les adresses se suivent : c+1 == & (c[1])   le + 1 ajoute l'unité correspondante à la taille de A

or, avec 'd', l'intention serait de dire que &(d[0]) = &a et &(d[1]) = &b alors que 'a' et 'b' ont été construits n'importe où, il y aurait rupture de la continuité d'adressage des tableaux

chapitre sur **le type référence** (&)

Y a t'il une notion de tableau de références ? Réfléchissons ... :

```
int main() {
   A a,b;
   A c[] {a,b};
   A *d[] {&a,&b}; // ça oui
   A &e{c[0]}; // ça aussi
   A f{c[0]};  // pas de pb
}
```

Ces écritures, correctes, illustrent des nuances à bien distinguer :
- 'c' tableaux de As contient des copies de 'a' et de 'b'
- 'd' est un tableau de pointeurs vers des A
- 'e' et c[0] sont la même variable (par alias)
- f est une copie de c[0]

Rq : pour des raisons similaires, il n'y a pas de vecteurs de références : si vous voulez faire ce genre de modélisation utilisez les pointeurs.

chapitre sur **le type référence** (&)

On peut aussi essayer ... :

```
int main() {
    A a;
    A & b{a};
    A & c{a};
    A & d{c};
    A & & e {c}; // non
    return EXIT_SUCCESS;
}
```

- aucun pb pour déclarer plusieurs alias vers 'a' (c'est le cas pour 'b' et 'c')
- ni même pour le faire indirectement (avec 'd' ici qui est un alias vers 'a' déclaré via 'c')
- mais, même si on peut comprendre ce que le programmeur aurait voulu dire avec 'e', cela n'a pas de sens de le typer aussi précisément : un alias est un alias, il n'y a pas de notion de distance à l'original

chapitre sur **le type référence** (&)

Que peut-on référencer, avec ce type ?

Nous avons compris que :
- un type référence ne peut pas changer d'objet référencé
- il doit (donc) être immédiatement initialisé à la déclaration (pour s'assurer qu'il référence qq chose d'existant)

Il reste à se poser une question, examinons :

```
int main() {
   string s{"bonjour"};
   string &x {s+ " toi "};
   // interdit
   return EXIT_SUCCESS;
}
```

chapitre sur **le type référence** (&)

Que peut-on référencer, avec ce type ?

Nous avons compris que :
- un type référence ne peut pas changer d'objet référencé
- il doit (donc) être immédiatement initialisé à la déclaration (pour s'assurer qu'il référence qq chose d'existant)

Il reste à se poser une question, examinons :

```
int main() {
  string s{"bonjour"};
  string &x {s+ " toi "};
  // interdit
  return EXIT_SUCCESS;
}
```

Le résultat de l'expression s+" toi" est temporaire, avec une durée de vie précaire, il est stocké dans une zone mémoire différente de celle des variables usuelles. On peut s'interroger sur le bien fondé de créer un alias vers cette zone, anonyme par nature, il aurait été plus naturel de créer une variable normale explicitement (comme avec "bonjour"). Etendons toutefois cet exemple ...

chapitre sur **le type référence** (&)

Que peut-on référencer, avec ce type ?

Nous avons compris que :
- un type référence ne peut pas changer d'objet référencé
- il doit (donc) être immédiatement initialisé à la déclaration (pour s'assurer qu'il référence qq chose d'existant)

Il reste à se poser une question, examinons :

```
int main() {
  string s{"bonjour"};
  string &x {s+ " toi "};
  // interdit
  return EXIT_SUCCESS;
}
```

```
void affiche (string &s) {
  cout << s;
}
int main() {
  string s{"bonjour"};
  affiche (s+ " toi ");
  // exemple comparable
  return EXIT_SUCCESS;
}
```

Le type référence (dans affiche) permet d'éviter de construire une copie, il est donc utile.
Pour rendre les choses possible, en tenant compte du fait que la zone où se situe l'expression anonyme est particulière, c++ accepte son référencement dès lors que le type se déclare constant.

chapitre sur **le type référence** (&)

Que peut-on référencer, avec ce type ?

Nous avons compris que :
- un type référence ne peut pas changer d'objet référencé
- il doit (donc) être immédiatement initialisé à la déclaration (pour s'assurer qu'il référence qq chose d'existant)

Il reste à se poser une question, examinons :

```
int main() {
  string s{"bonjour"};
  const string &x {s+ " toi "};
  // ok
  return EXIT_SUCCESS;
}
```

```
void affiche (const string &s) {
  cout << s;
}
int main() {
  string s{"bonjour"};
  affiche (s+ " toi ");
  // exemple comparable
  return EXIT_SUCCESS;
}
```

Le type référence (dans affiche) permet d'éviter de construire une copie, il est donc utile.
Pour rendre les choses possible, en tenant compte du fait que la zone où se situe l'expression anonyme est particulière, c++ accepte son référencement dès lors que le type se déclare constant.

chapitre sur **le type référence** (&)

Que peut-on référencer, avec ce type ?

Pour aller un tout petit peu plus loin (cela peut vous être utile pour décrypter les messages d'erreur du compilateur), c++ distingue ce qu'il appelle des LVALUE et des RVALUE au moment d'une affectation de la forme expression_1 = expression_2 :

- la partie gauche est une zone mémoire destinée à identifier une zone mémoire qui stockera des valeurs et qui seront ré-utilisées
- la partie droite peut être le résultat d'un calcul, peut être temporaire, elle n'est pas forcément destinée à être modifiée

Ces distinctions on conduit les concepteurs de c++ à introduire deux types de références :
- celles que nous avons vues, destinées aux l-values, et dénotées par &
- des références droites, destinées aux r-values, et dénotées par && (nous ne les utiliserons pas)

Retenez simplement que les références & forment des alias :
- avec des zones que l'on peut déjà nommer facilement
- avec des zones temporaires uniquement si l'alias est déclaré constant

Exemple 1 avec **le type référence** (&)

Le constructeur de copie

était déjà une justification solide de l’existence des références :
- on ne pouvait pas se contenter du type A seul,
- et, si on ne pouvait pas faire de copie implicitement, alors on ne pourrait pas passer un objet en argument autrement qu'en utilisant son adresse A* (remarquez que c'est le choix fait par java)

```
class A {
public:
 A(A const & original);
};
```

Exemple 2 avec **le type référence** (&)

Reprenons, et améliorons, l'exemple déjà rencontré suivant :

```
class A {
  private
    public:
     int valeur;
     A(int);
     A min(A other);
};
```

```
A::A(int x):valeur{x}

A A::min(A other) {
   if (other.valeur < valeur) return other;
   else return *this;
}
```

```
int main() {
  A a{1}, b{2};
  cout << a.min(b);
};
```

l'appel à min provoquait :
- une copie de b dans other
- un retour d'une copie de a (puisqu'ici c'est lui le plus petit)

On peut éviter ces copies grâce au type référence ...

Exemple 2 avec **le type référence** (&)

Reprenons, et améliorons, l'exemple déjà rencontré suivant :

```
class A {
  private
    public:
     int valeur;
     A(int);
     A& min(A &other);
};
```

```
A::A(int x):valeur{x}

A& A::min(A &other) {
   if (other.valeur < valeur) return other;
   else return *this;
}
```

```
int main() {
  A a{1}, b{2};
  cout << a.min(b);
};
```

plus aucune copie n’est faite grace à l’aliasing des références

Exemple 2 avec **le type référence** (&)

Reprenons, et améliorons, l'exemple déjà rencontré suivant :

```
class A {
  private
    public:
     int valeur;
     A(int);
     A& min(A &other);
};
```

```
A::A(int x):valeur{x}

A& A::min(A &other) {
   if (other.valeur < valeur) return other;
   else return *this;
}
```

```
int main() {
  A a{1}, b{2};
  cout << a.min(b);
};
```

plus aucune copie n’est faite grace à l’aliasing des références

Profitons encore de cet exemple pour expliquer ce qui se passe avec cout

L'affichage en c++, cout et <<

```
#include <iostream>
using namespace std;
int main() {
 int nb{3};
 string fruit{" pommes"};
 cout << "j'ai " << nb << fruit << endl;
}
```

L'affichage en c++, cout et <<

<< est un opérateur binaire. Il prend les arguments à sa gauche et à sa droite :

- un de type ostream
- l’autre est soit int, soit string, ...

Comme toute fonction on peut le surcharger
(l'intérêt est de le faire sur son second argument)

```
#include <iostream>
using namespace std;
int main() {
 int nb{3};
 string fruit{" pommes"};
 cout << "j'ai " << nb << fruit << endl;
}
```

L'affichage en c++, cout et <<

<< est un opérateur binaire. Il prend les arguments à sa gauche et à sa droite :

- un de type ostream
- l’autre est soit int, soit string, ...

Comme toute fonction on peut le surcharger
(l'intérêt est de le faire sur son second argument)

```
#include <iostream>
using namespace std;
int main() {
 int nb{3};
 string fruit{" pommes"};
 cout << "j'ai " << nb << fruit << endl;
}
```

```
ostream & operator<<( ostream &out , const int &x );
ostream & operator<<( ostream &out , const string &x );
```

ce sont ces deux redéfinitions, qui existent déjà, qui rendent possible les utilisations précédentes. Regardez ces signatures ... voilà notre 3ème d'utilisation des références !

Exemple 3 avec **le type référence** (&)
L'affichage en c++, cout et <<

Si on veux définir l’affichage pour les objets A, on procède ainsi (on ré-expliquera tout ensuite)

Dans A.hpp :

```
#include <iostream>
using namespace std;
class A {
public:
 string name;
 ... puis constructeur etc ...
};
ostream& operator<<( ostream &out , const A &x );
```

Notez que la déclaration se fait à l'extérieur de celle de la classe : ce n'est pas une fonction membre, l'opérateur est "dans le langage", mais il est naturel de déclarer sa surcharge dans A.hpp puisqu’elle concerne les objets A

Puis, dans A.cpp :

```
ostream& operator<<(ostream &out ,const A&x ) {
  out << x.name ;
  return out ;
}
```

Notez encore que l'on ne préfixe pas par A::
(toujours parce que ce n'est pas une fonction membre)

Exemple 3 avec **le type référence** (&)

L'affichage en c++, cout et <<

Alors il sera possible de faire simplement :

```
int main () {
   A a {"test"}, b{"truc"};
   cout << a << b << endl;
}
```

Décryptage :

- lors de l'appel cout << x où x est un A : la constante cout sera transmise en 1er paramètre de l'opérateur <<.
- cout ne sera pas copié : le passage par référence (indiqué par &) permet la création d'un alias. Il est naturel de faire l'économie d'une copie de cout (qui est probablement un objet complexe, et/ou peut être qu'il doit rester unique)
- le second argument est de type A
- lui aussi est transmis tel quel, sans copie
- on précise même ici qu'il ne sera pas modifié (const) car seule une lecture est utile. (On peut choisir que non)
- dans le bloc de code on utilise une autre surcharge déjà existante de operator<< : celle dont le second argument est une string
- le type retour est l'ostream d'entrée : cela permet, par associativité à gauche, d'enchaîner cout << truc << bidule;
- remarquez qu'on ne met pas const devant l'ostream, car le flux est modifié

```
ostream& operator<<(ostream &out ,const A&x ) {
  out << x.name ;
  return out ;
}
```

Avant dernier Transparent

pourquoi rien ne marche dans h ?

```
void f(A &x) { cout << "f" ;}
class A {
   void g() {cout << "g"}
};
h(const A &a) { f(a); a.g();}
```

Avant dernier Transparent

pourquoi rien ne marche dans h ?

```
void f(A &x) { cout << "f" ;}
class A {
   void g() {cout << "g"}
};
h(const A &a) { f(a); a.g();}
```

« a » est censé être constant.

Or son appel dans « f » est typé par un alias déclaré non constant : c’est à dire que potentiellement « f » peut modifier « x » (donc « a »).

C’est vrai qu’ici « f » ne touche pas à « x », mais le compilateur détecte un risque.

Avant dernier Transparent

pourquoi rien ne marche dans h ?

```
void f(const A &x) { cout << "f" ;}
class A {
   void g() {cout << "g"}
};
h(const A &a) { f(a); a.g();}
```

« a » est censé être constant.

Or son appel dans « f » est typé par un alias déclaré non constant : c’est à dire que potentiellement « f » peut modifier « x » (donc « a »).

C’est vrai qu’ici « f » ne touche pas à « x », mais le compilateur détecte un risque.

Avant dernier Transparent

pourquoi rien ne marche dans h ?

```
void f(const A &x) { cout << "f" ;}
class A {
   void g() {cout << "g"}
};
h(const A &a) { f(a); a.g();}
```

Il reste une erreur sur « a.g() »

Là aussi elle provient du fait que « a » est déclaré invariable.

L’appel à « g() » peut potentiellement modifier les attributs de « a ».

Si on ne veut pas restreindre son utilisation il vaut donc mieux l’avoir déclaré avec le type le plus adéquat.

Avant dernier Transparent

pourquoi rien ne marche dans h ?

```
void f(const A &x) { cout << "f" ;}
class A {
   void g()const {cout << "g"}
};
h(const A &a) { f(a); a.g();}
```

Il reste une erreur sur « a.g() »

Là aussi elle provient du fait que « a » est déclaré invariable.

L’appel à « g() » peut potentiellement modifier les attributs de « a ».

Si on ne veut pas restreindre son utilisation il vaut donc mieux l’avoir déclaré avec le type le plus adéquat.

Dernier Transparent

quel est le problème ici ?

```
A* f() {
   A a ;
   return &a ;
}
int main() {
   A* p {f()} ;
   // ici ...
}
```

Dernier Transparent

quel est le problème ici ?

```
A* f() {
   A a ;   // a est créé localement
   return &a ; // il sera détruit à la sortie
}
int main() {
   A* p {f()} ;
   // ici ... p pointe sur un objet détruit !
}
```

Dernier Transparent +1 !

quel est le « problème » ici ?

```
A f(A &x) {
   return x ;
}
int main() {
   A a, *p {&f(a)} ;
   cout << (&a == p) ; // affiche 0 (false)
}
```

Dernier Transparent +1 !

quel est le « problème » ici ?

```
A f(A &x) {
   return x ;
}
int main() {
   A a, *p {&f(a)} ;
   cout << (&a == p) ; // affiche 0 (false)
}
```

Même si « a » est passé à « f » en utilisant une référence, avec ce type retour une copie est faite.

Dernier Transparent +1 !

quel est le « problème » ici ?

```
A & f(A &x) {
   return x ;
}
int main() {
   A a, *p {&f(a)} ;
   cout << (&a == p) ; // affiche 1 (true)
}
```

Par contre si le type retour est une référence, on a bien le même objet